Project User Programs Design

Group 79

Name	Autograder Login	Email
Jacob Wu	student303	jacobwu@berkeley.edu
Max Ye	student95	maxye@berkeley.edu
Ryan Nader	student277	berkeleybear22ryan@berkeley.edu
Vedansh Malhotra	student39	vedansh@berkeley.edu

Argument Passing

Data Structures and Functions

`char* functionName` // Example: “print”
`char *token, *save_ptr` // Used to facilitate call to strtok_r
`char* argv[128]` // Example: address of each of “print” “hello” “world” “162”
`int argc` // Example: 4
`process_execute` from process.c and `thread_create` from thread.c

```
static void start_process(void* file_name_)
char* argv[128];
```

Algorithms

Call `strtok_r()` Example input: “print hello world 162” → output: “print” “hello” “world” “162”
Call `strlen()` Example input: “print” → output: 5
Call `memcpy()` to push items onto the stack

Pseudocode:

```
pid_t process_execute(const char* file_name) {
  Parse the function name of file_name and pass function name to thread_create
}
...
static void start_process(void* file_name_) { // Example: file_name_ = "print hello world 162"
  ...
  During the call to load, we pass only the function name to load // Example: function_name = "print"
  ...
  Create the argv array and initialize argc to 1
  For each word in file_name:
    Add the word to the stack, and save the address to argv
  Implement word alignment by calculating total size of argc, argv, argv[0] thru argv[argc - 1], and the null sentinel, decreasing the stack pointer by that amount, and then round down to the nearest 16-byte
  Push null sentinel (in the appropriate place in the block allocated)
  Push argv[argc - 1] to argv[0](in the appropriate place in the block allocated)
  Push argv address(in the appropriate place in the block allocated)
  Push argc (in the appropriate place in the block allocated)
  Push return address
  ...
}
```

Note: Each push operation consists of an update to the stack pointer (decremented by the appropriate size) and a memcpy operation (actual transfer of data onto the stack)

Sample:
`file_name` := args-multiple some arguments for you!
Assuming `PHYS_BASE` is 0xc0000000:

```
 Address         Name          Data           Type
0xbfffffc2   argv[0][...]    args-multiple\0 char[14]
0xbfffffed   argv[1][...]    some\0          char[5]
0xbffffff3   argv[2][...]    arguments\0     char[10]
0xbffffffdf  argv[3][...]    for\0           char[4]
```

```
0xbffffffa   argv[4][...]    you!\0          char[5]
0xbffffff0   stack-align       0         uint8_t
0xbffffff4   argv[5]           0         char *
0xbffffff8   argv[4]        0xbffffffa   char *
0xbffffffc   argv[3]        0xbffffffd   char *
0xc0000000   argv[2]        0xbfffffe3   char *
0xc0000004   argv[1]        0xbfffffed   char *
0xc0000008   argv[0]        0xbffffff2   char *
0xc000000c   argv           0xbffffff8   char **
0xc0000010   argc              5         int
0xc0000014   return address    0         void (*) ()
```

Synchronization

- We choose to use `strok_r` which is the thread-safe version of `strok`
- We should obtain a lock once we start changing the stack pointer, and release the lock only after the entire change to stack pointer is complete, to prevent a race condition if two calls to start_process are made at the same time.

Rationale (and Edge Cases)

- These operations must be done after the stack pointer is set up, i.e. after `setup_stack` call in `load`.
- These operations need to be done in a place with access to the stack pointer as well as the full function header, so we do it in `process_execute`.
- We support functions up to 128 arguments, since it's rare for function calls to exceed this count. Any more than 128 will be rejected via a check of the argument count.
- We decrement ESP by the space needed to store argv, argc, and argv[0]...argv[arc-1] and null sentinel, then 16-byte align it, then go back to add the actual values of the corresponding items onto the stack. This makes it easier to debug compared to calculating separately the amount of offset needed for stack alignment, and also allows for a more natural for loop implementation.
- Total line count is about 30.

Process Control Syscalls

Data Structures and Functions

userprog/syscall.c

```c
// modify
static void syscall_handler(struct intr_frame*);

// add
static bool valid_byte_memory(const void* addr);
static bool valid_user_memory(const void* addr, size_t size);
static bool valid_user_string(void* str_);
static void assert(bool condition);
```

userprog/process.c

```c
// modify
pid_t process_execute(const char* file_name); // for exec syscall
static void start_process(void* file_name_); // for exec syscall
int process_wait(pid_t child_pid UNUSED); // for wait syscall
void process_exit(void); // for wait syscall
```

userprog/exception.c

```c
// modify
static void kill(struct intr_frame*); // update status if we don't exit normally
```

process.h

```c
// add
typedef struct wait_data {
  pthread_mutex_t lock; // protects ref_cnt
  int ref_cnt; // initialize to 2 (represents parent and child access)
  semaphore exit_sema;
  int exit_code;
  semaphore load_sema; // to keep track of load status
  bool loaded;
  list_elem elem; // to put this wait data into the parent's wait_statuses list
  pid_t pid;
} wait_data_t;
```

process.c (implementation along the lines of discussion 3)

```c
// add
void init_wait_data(struct wait_data_t *wd) {
  sema_init(&wd->sema, 0);
  wd->ref_cnt = 2;
  lock_init(&lock, NULL);
  wd->exit_code = NULL;
}

// modify
static void init_process(struct process* p, const char* name, int priority) {
  ...
  list_init(t->wait_statuses);
  init_wait_data(t->wait_data);
  sema_init(t->child_sema, 0);
  t->loaded = false;
  ...
}

// modify
struct process {
  ...
  struct list wait_datas; // list of children's wait_datas
  struct wait_data_t* wait_data; // the process's own wait data
  ...
}
```

Algorithms

userprog/syscall.c

`valid_user_memory`: (add) validate the pointer using functions in `userprog/pagedir.c` and in `threads/vaddr.h`

- Ensure argument is `!NULL`
- Calls helper method `valid_byte_memory`, which:
  - Checks the pointer is not null
  - Checks the pointer maps to valid user memory using `is_user_vaddr`
  - Retrieves the corresponding page using `pagedir_get_page(thread_current()->pcb->pagedir, addr)` and make sure `!NULL`
- Return whether or not all checks pass

`syscall_handler`: (modify) create a big switch statement (or multiple if-else statement blocks) for all the syscalls we need to implement

- Call `valid_user_memory` before making the system calls
  - First, validate all native types (int or unsigned) using valid_user_memory, which will call valid_byte_memory to iterate over all bytes for the size of the int or unsigned
  - valid_byte_memory checks `addr != NULL && is_user_vaddr(addr) && pagedir_get_page(thread_current()->pcb->pagedir, addr)`
  - Then, validate all strings using valid_user_string: ensure the string pointer is valid using valid_user_memory, then check that each character in the string is valid using valid_user_memory
  - All these checks, if failed, will trigger `printf("%s: exit(-1)\n", thread_current()->pcb->process_name);` and `process_exit();`
- Use `args[0]` in a switch statement and call the corresponding function, storing return values in the `eax` register if applicable
- `practice`
  - Set `eax` to `args[1] + 1`
- `halt`
  - Close all open file descriptors
  - Call `shutdown_power_off`
- `exit`
  - Close all open file descriptors
  - Set `eax` to status code
  - Print `%s: exit(%d)`, where process name and exit code respectively subsitute `%s` and `%d`
  - Exit the thread with `thread_exit()` provided in threads/thread.h
- `exec`
  - Call `process_execute(args[1])` and set `eax` to return value of that
- `wait`
  - Call `process_wait(args[1])` and set `eax` to return value of that

`exec`

- userprog/process.c
  - `process_execute`
    - Call `sema_down` on the child process's `load_sema` (to get notified when child loads)
    - Then, (now that we are past `sema_down`, this means child has loaded) check child's `loaded`
      - If `loaded` is `true`, return child's tid, else return -1

    - `start_process`
        - After successful call to `load`, set thread's `loaded` to true
        - Call `sema_up` on process's `load_sema` (to notify parent)
- userprog/exception.c
    - `kill`
        - Set thread's `loaded` to false (it should already be initialized to false)
        - Call `sema_up` on thread's `load_sema` (to notify parent)
- Create a free function to free process's child's variables of list `wait_datas` and `wait_data`
- process.c: `init_process` shown above
- process.h: `struct process` shown above

`wait`

- userprog/process.c
    - `process_wait`
        - Iterate through `wait_datas` to find the matching pid that was passed in
        - If no matches, return -1; otherwise, call `sema_down` on the child (p) with `p->wait_data->exit_sema`
        - After we are past `sema_down`, save the child's `exit_code` locally
        - Acquire the lock of `p->wait_data->lock`
        - Decrease `ref_cnt` by 1 and if `ref_cnt` is now 0, release lock, free `p->wait_data` and remove it from the current process's `wait_datas` list; otherwise, only release lock
        - Return the child's `exit_code`
    - `start_process`
        - After successful call to `load`, call `init_wait_data` and add the result to the current process's `wait_datas` list
    - `process_exit`
        - Call sema_up on `curr_proc->wait_data->exit_sema`
        - Acquire the lock of `curr_proc->wait_data->lock`
        - Decrease `ref_cnt` by 1
        - If the current thread's `ref_cnt` is 0, first free the lock and then free the process's `wait_data`; otherwise, only free lock
        - Iterate through `wait_datas`
            - Decrement `ref_cnt` of the children, acquire and releasing the lock as appropriate
            - If child's `ref_cnt` is 0, first release the lock and then free the child's `wait_data` and delete the `elem` from the list; otherwise, only free lock
    - userprog/exception.c
        - `kill`
            - Call sema_up on `curr_process->wait_data->exit_sema`
- process.c: `init_wait_data` shown above

Synchronization

- `exec`
    - Since "parent process cannot return from a call to `exec` until it knows whether the child process successfully loaded its executable", we use a semaphore.
    - The parent calls `sema_down` (in syscall.h) to force it to wait up the child calls `sema_up`.
- `wait`
    - Similar to `exec`, we use a semaphore to make sure the parent waits on the child.
    - We use `ref_cnt` to determine how many references are pointing to the `wait_data`.

Rationale

We implement variable memory checks to ensure the arguments passed in are valid
Memory checks are done in multiple, modularized functions depending on granularity. check_byte_memory is the most granular, and is called by check_user_memory where the input could be an int (4 bytes). At times, we use check_string_memory for char arrays, and this check calls check_user_memory for each character.

File Operation Syscalls

Data Structures and Functions

Create and maintain the per-process file descriptor table (HashMap where key is index number and value is struct file object in the Global File Descriptor Table.)

```
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 list. */

  /* 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 */
#endif

  /* Owned by thread.c. */
  unsigned magic; /* Detects stack overflow. */

  /* Group's proposed changes (Sep 16) */
  struct file* fdt[128]; // File Descriptor Table, fixed size of 128 based on @https://cs162.org/static/proj/proj-userprog/docs/faq/syscalls/

  int next_fd; // Integer to the next available file descriptor slot
  /* End: Group's proposed changes (Sep 16) */
};
```

Algorithms

- SYS_CREATE
  - Validate arguments (See previous section)
  - Call `filesys_create` and store the return valid in f->eax
- SYS_REMOVE
  - Validate arguments (See previous section)
  - Call `filesys_remove` and store the return valid in f->eax
- SYS_OPEN
  - Validate arguments (See previous section)
  - Get the File Descriptor Table and retrieve `next_fd` from thread
  - Ensure `next_fd < 2 || 127 < next_fd || file_descriptor_table[next_fd] != NULL`, if not we perform linear search to find an open and available slot
    - If this fails, then we have too many files open (we can support up to 126 files)
  - Call `filesys_open` and update the file descriptor table, and return next_fd++ in f->eax
    - Also, perform a check to ensure opened_file != null, otherwise return with -1 in f->eax
- SYS_FILESIZE
  - Validate arguments (See previous section)
  - Get the file descriptor table
  - Call `filesys_remove` and store the return valid in f->eax
  - Validate file descriptor: `assert(1 < fd && fd < 128 && file_descriptor_table[fd] != NULL);`
  - Call `file_length(file_descriptor_table[fd]);` and return the results in f->eax
- SYS_READ
  - Validate arguments (See previous section)
  - Get the file descriptor table
  - If we are reading from system input, call `for (unsigned i = 0; i < size; i++) byte_ptr[i] = input_getc();`
  - Validate file descriptor: `assert(1 < fd && fd < 128 && file_descriptor_table[fd] != NULL);`
  - Call `file_read` and return the results
- SYS_WRITE
  - Validate arguments (See previous section)
  - Get the file descriptor table
  - If we are writing to system output, call `putbuf(buffer, size);`
  - Validate file descriptor: `assert(1 < fd && fd < 128 && file_descriptor_table[fd] != NULL);`
  - Call `file_write` and return the results
- SYS_SEEK
  - Validate arguments (See previous section)
  - Get the file descriptor table
  - Validate file descriptor: `assert(1 < fd && fd < 128 && file_descriptor_table[fd] != NULL);`
  - Call `file_seek` and return the results
- SYS_TELL
  - Validate arguments (See previous section)
  - Get the file descriptor table
  - Validate file descriptor: `assert(1 < fd && fd < 128 && file_descriptor_table[fd] != NULL);`
  - Call `file_tell` and return the results
- SYS_CLOSE
  - Validate arguments (See previous section)
  - Get the file descriptor table
  - Validate file descriptor: `assert(1 < fd && fd < 128 && file_descriptor_table[fd] != NULL);`
  - Call `file_tell` and remove the corresponding fdt entry

`next_fd` serves as a counter to provide a unique file descriptor for the next file that the thread or process opens. Every time a file is opened, the value of **`next_fd`** is assigned as the file descriptor for that file, and then **`next_fd`** is incremented.
If **`next_fd`** exceeds a certain number (array indices start from 0), the file descriptor table is full. The operating system must handle this scenario, either by returning an error when trying to open a new file or by implementing mechanisms to expand the file descriptor table.

File Operation System Calls	Pintos Implementations
create	*`SYS_CREATE`*

remove	`SYS_REMOVE`
open	`SYS_OPEN`
filesize	`SYS_FILESIZE`
read	`SYS_READ`
write (most testing uses this)	`SYS_WRITE`
seek	`SYS_SEEK`
tell	`SYS_TELL`
close	`SYS_CLOSE`

Document Functions	Description (Specification)
`file_deny_write`	Used to help with idea that while a user process is running, you must ensure that nobody can modify its executable on disk
`file_allow_write`	Used to help with idea that while a user process is running, you must ensure that nobody can modify its executable on disk

Synchronization

We implement a global lock to ensure only one file operation system call can be processed at a time.

Rationale

We make each function call separate so it's easier to maintain and support. In the meantime, it also makes it possible to call other functions (e.g. exit may need to call file close). This design approach should require minimal coding, say 5-30 lines per system call.

Floating Point Operations

Data Structures and Functions + Files

```
/* switch_thread()'s stack frame. */
struct switch_threads_frame {
  uint32_t edi;        /*  0: Saved %edi. */
  uint32_t esi;        /*  4: Saved %esi. */
  uint32_t ebp;        /*  8: Saved %ebp. */
  uint32_t ebx;        /* 12: Saved %ebx. */
  void (*eip)(void);   /* 16: Return address. */
  struct thread* cur;  /* 20: switch_threads()'s CUR argument. */
  struct thread* next; /* 24: switch_threads()'s NEXT argument. */
};
```

We need to add an attribute for the `uint8_t floating_point_save_state[108]` in the `struct switch_threads_frame` so that when we save the stack frame when we have the FPU data.

```
/* Interrupt stack frame. */
struct intr_frame {
  /* Pushed by intr_entry in intr-stubs.S.
     These are the interrupted task's saved registers. */
  uint32_t edi;       /* Saved EDI. */
  uint32_t esi;       /* Saved ESI. */
  uint32_t ebp;       /* Saved EBP. */
  uint32_t esp_dummy; /* Not used. */
  uint32_t ebx;       /* Saved EBX. */
  uint32_t edx;       /* Saved EDX. */
  uint32_t ecx;       /* Saved ECX. */
  uint32_t eax;       /* Saved EAX. */
  uint16_t gs, : 16;  /* Saved GS segment register. */
  uint16_t fs, : 16;  /* Saved FS segment register. */
  uint16_t es, : 16;  /* Saved ES segment register. */
  uint16_t ds, : 16;  /* Saved DS segment register. */
  /* Pushed by intrNN_stub in intr-stubs.S. */
  uint32_t vec_no; /* Interrupt vector number. */
  /* Sometimes pushed by the CPU,
     otherwise for consistency pushed as 0 by intrNN_stub.
     The CPU puts it just under `eip', but we move it here. */
  uint32_t error_code; /* Error code. */
```

```
  /* Pushed by intrNN_stub in intr-stubs.S.
     This frame pointer eases interpretation of backtraces. */
  void* frame_pointer; /* Saved EBP (frame pointer). */
  /* Pushed by the CPU.
     These are the interrupted task's saved registers. */
  void (*eip)(void); /* Next instruction to execute. */
  uint16_t cs, : 16; /* Code segment for eip. */
  uint32_t eflags;   /* Saved CPU flags. */
  void* esp;         /* Saved stack pointer. */
  uint16_t ss, : 16; /* Data segment for esp. */
};
```

We need to add an attribute for the `uint8_t floating_point_save_state[108]` in the `struct intr_frame`, so that we can capture the entire FPU state during an floating point interrupt.

Struct	Description
`struct intr_frame`	This stores the metadata of the stack frame for the floating point unit for each task. This lives in the kernel stack. This is in file: `src/threads/interrupt.h`
`struct switch_threads_frame`	The role of this is when performing a context switch you will need to save the stack frame of `switch_thread()`. This is in file: `src/threads/switch.h`

File	Changed?	Description
`threads/interrupt.h`		interrupt handling code for context switches
`threads/intr-stubs.S`	Y	interrupt handling code for context switches *we need to change `intr_entry` and `intr_exit` in `intr_entry` this is the main interrupt entry point and we need to use `FSAVE` here to save the state before switching in `intr_exit` this is where we exit the interrupt and need to use `FRSTOR` to restore fp value the thread we are starting back up
`threads/switch.h`		interrupt handling code for thread switching
`threads/switch.S`	Y	interrupt handling code for thread switching this file is x86 and has the mechanics for context switching it saves the state of the currently running thread and restores the state if the next thread onto the CPU *will need to add some code here to handle adding the 108 byte `floating_point_save_state` during a context switch i.e. using the `FSAVE` *and also will use `FRSTOR` in this file for the 108 byte `floating_point_save_state`
`lib/stdio.c`		used to print floating point values, only allows for `<whole>` or `<decimal>` which are both 32-bit size, anything larger will error
`src/threads/loader.S`		loads the kernel
`src/threads/start.S`	Y	this is the kernel startup code *changes need to be made here in the `start` block using `FNINIT` in order to initialize the FPU at the startup
`src/userprog/process.c`	Y	`start_process` function which is a thread function that loads a user process and starts it
`src/userprog/thread.c`	Y	`thread_create` function which is responsible for creating a new kernel thread
`src/userprog/syscall.c`	Y	look at `syscall_handler` and modify in order to handle the floating point system calls

Method	Description
`struct thread* switch_threads(struct thread* cur, struct thread* next);`	Switches from CUR, which must be the running thread, to NEXT, which must also be running switch_threads(), returning CUR in NEXT's context.
`intr_entry`	Main interrupt entry point.

	An internal or external interrupt starts in one of the intrNN_stub routines, which push the `struct intr_frame` frame_pointer, error_code, and vec_no members on the stack, then jump here. We save the rest of the `struct intr_frame` members to the stack, set up some registers as needed by the kernel, and then call intr_handler(), which actually handles the interrupt. We "fall through" to intr_exit to return from the interrupt.
`intr_exit`	Interrupt exit. Restores the caller's registers, discards extra data on the stack, and returns to the caller. This is a separate function because it is called directly when we launch a new user process (see start_process() in userprog/process.c).

Table 3-18 Control Instructions (Floating-Point)
The floating-point control instructions operate on the floating-point register stack and save and restore the floating-point state.

Solaris Mnemonic	Intel/AMD Mnemonic	Description	Important?	Notes
`fclex`	`FCLEX`	clear floating-point exception flags after checking for error conditions		
`fdecstp`	`FDECSTP`	decrement floating-point register stack pointer		
`ffree`	`FFREE`	free floating-point register		
`fincstp`	`FINCSTP`	increment floating-point register stack pointer		
`finit`	`FINIT`	initialize floating-point unit after checking error conditions	*1	Might be able to use this instead of `FNINIT` not sure whether or not would work but would be used in an identical way, might be better because has error checking.
`fldcw`	`FLDCW`	load floating-point unit control word		
`fldenv`	`FLDENV`	load floating-point unit environment		
`fnclex`	`FNCLEX`	clear floating-point exception flags without checking for error conditions		
`fninit`	`FNINIT`	initialize floating-point unit without checking error conditions	*1 (**use this**)	Used in `start.S` file in `start` to set up the 108-byte `floating_point_save_state`.
`fnop`	`FNOP`	floating-point no operation		
`fnsave`	`FNSAVE`	save floating-point unit state without checking error conditions		Should use `FSAVE` instead because has error checking.
`fnstcw`	`FNSTCW`	store floating-point unit control word without checking error conditions		
`fnstenv`	`FNSTENV`	store floating-point unit environment without checking error conditions		
`fnstsw`	`FNSTSW`	store floating-point unit status word without checking error conditions		
`frstor`	`FRSTOR`	restore floating-point unit state	*	Used in `threads/switch.S` file to restore the state during a context switch i.e. when switching threads in `thread_stack_ofs` you can see in the comments that, “switches from CUR, which must be the running thread, to NEXT, which must also be running switch_threads(), returning CUR in NEXT's context.”
`fsave`	`FSAVE`	save floating-point unit state after checking error conditions	*	Used in the `threads/switch.S` file to save state during a context switch i.e. when switching threads in `switch_threads`.
`fstcw`	`FSTCW`	store floating-point unit control word after checking error conditions		
`fstenv`	`FSTENV`	store floating-point unit environment after checking error conditions		
`fstsw`	`FSTSW`	store floating-point unit status word after checking error conditions		
`fwait`	`FWAIT`	wait for floating-point unit		

`wait`	`WAIT`	wait for floating-point unit		

Algorithms

Consult Table 3-18 Control Instructions for the extra details.

We'll be working with 108-byte FPU save state as a whole.
This save state will be stored as a list i.e. `uint8_t floating_point_save_state[108]` so that we have a byte addressable list to the floating point unit save state and this also simplifies the design because we can abstract away the individual parts of the floating point unit's save state.

Based on the tables above, we can see that the `FNINIT` or `FINIT` is used to initialize the floating point unit and needs to be done at startup in `src/threads/start.S` to set up the 108 bytes properly in the `start` block.

We can also see that we need to use `FSAVE` and `FRSTOR` in the file `threads/switch.S`. (i.e. `FSAVE` in the `switch_threads` block and `FRSTOR` in the `thread_stack_ofs` block.)

`FSAVE` is used to save the entire FPU state and will take in the memory location where this entire FPU state block will be stored.

The `FRSTOR` is used to restore the entire FPU state and will take in the memory location where this entire FPU state block will be stored.

Note that during a context switch you need to use `FSAVE` to create a save state for the FPU for the thread that is being replaced and then you need to `FRSTOR` the thread that is doing the replacing so that you can bring up its previously saved state.

Now specifically for system call `compute_e` (although it is a fake system call like `practice`) we will use `SYS_COMPUTE_E`, it does the following: $\sum_{i=0}^{n} \frac{1}{n!}$ and because the factorial of negative numbers is not defined we need to check that $n \geq 0$ and by x86 convention the return values must be placed on the `eax` register which is an attribute of the `struct intr_frame`.

All this detail is in the table but for clarity, we need to change `threads/intr-stubs.S` in the following ways as well:
We need to change `intr_entry` and `intr_exit`:
- In `intr_entry` this is the main interrupt entry point and we need to use `FSAVE` here to save the state before switching.
- In `intr_exit` this is where we exit the interrupt and need to use `FRSTOR` to restore fp value the thread we are starting back up.

Lastly, whenever creating/switching to a new/different process/thread i.e. if there is a child thread, new thread init or a context switch it is important to note that as described in the manual you need to save "the current FPU registers into a temporary location (e.g. local variable), initializing a clean state of FPU registers, save these FPU registers into the desired destination, and restore the registers from the aforementioned temporary location."

Synchronization

No synchronization is needed for the following reasons:

For created processes and threads the floating point metadata during initialization should only be using the memory space for that processes or threads and thus should not create a situation where other threads are looking at the initialization data.
In any other situation, when an interrupt takes place and we enter the interrupt handler we disable future interrupts until this one is resolved and because we are assuming as the OS that there is just one FPU and the CPU only has a single core so can only execute one instruction at a time (i.e. there is no parallel execution of code) on that core when in the interrupt handler and thus we will not have competition for the registers and thus no need for synchronization.

Rationale

As described in detail above in order to implement this we will need to modify a combination of .c and .S files, listed in the table above with the files. For this section most of the design is just filling in already existing code so no time/space complexity analysis. The only shortcoming that comes to mind is whether we need to use `FNINIT` or `FINIT` and if any of the other floating point control instructions will need to be used and this will become more clear once further implementation occurs. I imagine that it will be relatively easy to expand on this if needed as we are taking advantage of the modularity of the existing codebase. With regards to the amount of coding based on how much code they said was added to complete the project in the specification it seems that not too many lines will need to be added to the assembly files and perhaps more will need to be added to some of the c files.

Concept check

Q1: Take a look at the Project User Programs test suite in `src/tests/userprog`. Some of the test cases will intentionally provide invalid pointers as syscall arguments, in order to test whether your implementation safely handles the reading and writing of user process memory. Identify a test case that uses an **invalid** stack pointer (`%esp`) when making a syscall. Provide the name of the test and explain how the test works. Your explanation should be very specific (i.e. use line numbers and the actual names of variables when explaining the test case).

A1: tests/userprog/child-bad.c. The instruction on line 10 of movl $0x20101234, %esp; sets %esp to the hex value of 0x20101234 while is equivalent to 537924148 in decimal. This address is not 16-byte stack aligned (it is equal to 4 mod 16) and is therefore an invalid stack pointer when making the syscall. ESP must be 16-byte aligned right before invoking call on int $0x30.

Q2: Please identify a test case that uses a **valid** stack pointer when making a syscall, but the stack pointer is too close to a page boundary leading to some of the syscall arguments being located in invalid memory. Provide the name of the test and explain how the test works. Your explanation should be very specific (i.e. use line numbers and the actual names of variables when explaining the test case).

A2: `src/tests/userprog/exec-bound-3.c` calls `exec` with a string that begins at the end of a valid memory address and extends to an invalid memory address, per the comments. On line 12, we call `get_bad_boundary()` which "returns an address that is invalid, but the preceding bytes are all valid... Used to position information such that the first byte of the

information is valid, but not all the information is valid." We set char* p to this address minus 1 which brings us to an address that is valid by the skin of its teeth. We set the value at this valid address to 'a' and call `exec` on it. `exec` expects a `char* cmd_line` and will continue to read until it hits a null terminator. When `exec` reads past 'a' at the valid address, it goes into invalid address space, which should fail and kill the process.

Q3: Identify **one** part of the project requirements which is **not** fully tested by the existing test suite. Provide the name of the test and explain how the test works. Explain what kind of test needs to be added to the test suite, in order to provide coverage for that part of the project.

A3: What happens if a thread has already received a priority donation and then another thread with even higher priority attempts to acquire the same lock? We should test that the system handles re-donation correctly.

We would call it `priority-donate-redonate`. Here's how it might work:

1. The main thread acquires a lock and thus holds it.
2. A secondary thread (Thread 2) attempts to acquire the same lock that the main thread holds. Since the lock is occupied, Thread 2 should donate its priority to the main thread.
3. A third thread (Thread 3) with a higher priority than Thread 2 also tries to acquire the same lock. This should lead to a "priority re-donation," where the main thread should now operate with this new, higher priority that Thread 3 has.
4. The main thread finally releases the lock. At this point, the priority of the main thread should be restored to its original value, and Thread 3 and then Thread 2 should acquire the lock in that order due to their higher priorities.

   PS: This will, of course, run on the priority-based scheduler.
   I.e. we'd add a statement like: `ASSERT(active_sched_policy == SCHED_PRIO);`