Project User Programs Design

Group 33

NAME	AUTOGRADER LOGIN	EMAIL
Mingqian Liao	student461	liaomq@berkeley.edu
Junhee Park	student333	junhee20park@berkeley.edu
Hugh Lau	student198	hughlau@berkeley.edu
Chris Liu	student105	chris-liu@berkeley.edu

Argument Passing

Data Structures and Functions

Modification in `**src/userprog/process.h**`
We add a macro to limit the max number of arguments to 100.

```
#define MAX_ARGUMENTS 100
```

Modification in `**src/userprog/process.c**`
We modify the function `load`, by adding the following variables:

```
bool load(const char* file_name, void (**eip)(void), void** esp) {
  int argc = 0; // number of arguments parsed from file_name
  char *argv[MAX_ARGUMENTS]; // string arguments parsed from
file_name
  // Parse arguments from file_name and pass argc and argv to
setup_stack.
  // ...
}
```

We also modify the header of the `setup_stack` function, and add a new variable:

```
static bool setup_stack(void** esp, int argc, char *argv[]) {
  char *argv_addr[argc]; // Keeps pointer to each string argument.
  // Set up stack, following convention.
  // ...
}
```

Algorithms

In the function `load`, we parse the arguments using `strtok_r()` with empty space (“ “) as the delimiter as such:

```
bool load(const char* file_name, void (**eip)(void), void** esp) {
  // ...
  char *token = strtok_r(file_name, " ", saveptr);
  while (token != NULL) {
    // If argc >= MAX_ARGUMENTS, return false (loading not
successful)
    argv[argc++] = token;
    token = strtok_r(NULL, " ", saveptr)
  }
  // ...
}
```

When the `load` function calls the setup_stack function, we will then call `setup_stack(esp, argc, argv)`. function. If we successfully `install_page` in `setup_stack`, then after setting ESP to `PHYS_BASE`, we implement the following:

```
static bool setup_stack(void** esp, int argc, char *argv[]) {
  // ...
  *esp = PHYS_BASE;

  // Place actual argument strings of argv on the stack. Each time we
do so, we also decrement the ESP by 4, and store the memory address
(ESP) into argv_addr[i].
  for i=argc-1 to i=0; *esp -= 4:
    memcpy(**esp, argv[i], strlen(argv[i]));
    argv_addr[i] = *esp;

  // Pad the stack for 16-byte alignment. (argc + 2) to account for
fact that stack must be aligned where argc is stored. Then decrement
esp by 4 to account for null pointer sentinel.
  *esp -= ((argc + 2) % 4) * 4;
  *esp -= 4;

  // Place memory addresses of each argument on the stack, in reverse
order (ie argv[0] goes in the lowest memory address).
  for i=argc-1 to i=0; *esp -= 4:
    **esp = argv_addr[i];

  // Place memory address of argv on stack (argv[0] located at ESP +
4 at this point).
  **esp = *esp + 4;

  // Add argc to the stack, and decrement ESP by 4. Then decrement
ESP by 4 again to account for return address.

  // ...
}
```

At the end of setup_stack, we will then have the following on the stack, from highest to lowest memory address:

- Strings (`char[]` from `argv[argc-1]` to `argv[0]`)
- Padding for 16-byte alignment,: `((argc + 2) % 4) * 4`
- Null pointer sentinel
- Pointers to the string arguments (in reverse order, `char*` from `argv_addr[argc-1]` to `argv[0]`)
- Pointer to `argv` (`char**`)
- `argc` (int)
- Return address (`void*`)

Synchronization

To allow synchronization without error, we use the re-entrant version of `strtok()` function while parsing for the arguments — `strtok_r()` -- rather than `strtok()` function itself. Because the re-entrant version takes an extra argument to store the state between calls, it can be interrupted and re-entered by another thread without crashing the program. If the `strtok()` function were used, multiple threads may call `strtok()` on different strings. Since `strtok()` searches on the last non-NULL string it was passed in, it may result in wrong results by looking at the wrong string. For example, if the execution goes as follows: thread A calls `strtok(``"``hello;world;my``"``, "``;``"``)` → thread B calls `strtok(``"``what.they.mean``"``, "``.``"``)` → thread A calls `strtok(NULL, "``;``"``)`, then thread A may now end up receiving “they.mean” rather than “world”.

Rationale

Overall, this design follows the 80x86 calling convention of setting up the arguments (in reverse order) on the stack. A macro `MAX_THREADS` was used to make sure that the max number of arguments allowed does not change, and will always be 100. Since we know the max number of arguments, we could safely use this number to initialize our `char *argv[]`. The parsing of the `file_name` was done in `load`, because this is the function that calls on `setup_stack`. In `setup_stack`, although putting the order of the actual string arguments on the stack did not matter, we chose to keep it in the same reverse order (from `argc-1` to `0`, where `argv[0]` is placed on the lowest memory address) for ease of understanding and keeping it uniform with the pointers of the string arguments which must be in reverse order. We also placed a null pointer sentinel to ensure that `argv[argc]` is a null pointer in accordance with C standards.

Process Control Syscalls

Data Structures and Functions

Modifications** in `**src/userprog/process.h**`**
To support process control syscalls, we'll be making some changes in `process.h` to keep track of relevant information.

PCB Structs
We want to modify the process PCB struct to keep track of information relevant to its parent and child processes. Specifically, we do this by adding a pointer to the parent process PCB struct (to be used when exiting), as well as a boolean representing whether or not the parent process has exited yet. We also record a file pointer to the executable it's running (used when denying/allowing writes to it), alongside information about its child processes (see "Child Info" Structs). Finally, we include a lock to be obtained when we access this struct.

```
struct process {
  // ... existing data
  struct process *parent_process;
  bool parent_exited;
  FILE *executable;

  mutex_t pcb_lock;

  struct list child_info_structs;
};
```

"Child Info" Structs
We want to allow each process to track information about its children, to be used when it waits on a child process or when one of its child processes exits.

For each child, the parent needs to know the following:

- The pid of the child process `pid_t child_pid`, used for identifying which child info struct belongs to which child process.
- A `bool waited`: whether or not this child process has been waited on before, initialized to false.
- The `int exit_status` denoting the exit status of the child process, initialized to -1.
- The `sem_t wait_semaphore`, initialized at zero and used to synchronize a parent process's wait() syscall with this process's execution.

```
struct child_info {
  pid_t child_pid; // process id of corresponding child process
  bool waited; // has this child process already been waited on
  int exit_status; // exit status, init to -1
  struct semaphore wait_semaphore; // semaphore used for scheduling
wait syscalls, init to zero, upped when finished running and a
waiting thread will down this

  struct list_elem elem; // to facilitate the creation of child info
struct lists
}
```

Helper Functions/Abstractions

We'll frequently need to find a child info struct corresponding to a process's pid; creating an abstraction to retrieve a child info struct with a given pid if it exists in a list of child info structs will simplify our code:

```
child_info *find_child_info_struct(struct list *child_info_structs,
pid_t child_pid);
```

If such a child info struct doesn't exist, this function should return a NULL pointer, so return values from this function will need to be checked appropriately.

Algorithms

practice : `int practice (int i)`
Return i + 1.

halt : `void halt (void)`
Call the `shutdown_power_off` function in `devices/shutdown.h`.

exit : `void exit (int status)`

1. **Error checking**:
    a. *Parent process has already exited*: since we'll be accessing the parent process's PCB to update information, we need to make sure that the parent process hasn't already exited (causing its PCB block to be freed). We can do this by checking if the boolean `parent_exited` within our current PCB is true (this will be updated when the parent exits): if so, the parent's PCB block has been freed and thus there is no struct to update information in → skip to step 6.
2. Assuming none of the above errors are found, access this process's parent PCB block through the `*parent_process` address found in our current PCB block. Obtain the lock `pcb_lock` on the parent PCB struct.
3. Within its parent PCB, access its list of `child_info_structs` and find the child info struct that corresponds to the exiting process using the helper function `find_child_info_struct()`.
4. Edit the `status_code` of this child info struct to equal `status`
5. Up the `wait_semaphore` to allow its parent to move forward with a potential wait syscall.
6. Print the exit status in the format `%s: exit(%d)` followed by a newline, with the process name and exit status replacing `%s` and `%d`, respectively. We get the process name from the PCB struct.
7. Call `file_close()` on the file pointer `executable` stored in the exiting process's PCB, and free the PCB struct of the exiting process.

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

1. Obtain the lock `pcb_lock` for the parent process's PCB struct.

2. **Attempt to have the child process load the program**. The parent process will run `process_execute()`, passing in the executable:
    a. We will modify `process_execute()` such that it has the responsibility of not returning until we've fully attempted to load the executable. To do this, we allocate space on the heap for a semaphore `load_semaphore` initialized to zero, and alter `start_process()` in `process.c` to take in the address of this semaphore; it will up it after it has succeeded/failed in loading the executable.
    b. Allocate space on the heap for a boolean `load_success`, and alter `start_process()` to additionally take in the address of this boolean; it will be used to communicate if loading the executable was successful.
    c. Attempt to load the program in a new thread running `start_process()`, and immediately down `load_semaphore`. The new thread running `start_process()` will:
        i. Create a PCB struct for the child process (separate from the child info struct to be stored in the parent's PCB).
            i. Set `parent_process` to the address of the parent's PCB struct.
            ii. `parent_exited` is set to false.
            iii. The file pointer `executable` will be set inside the upcoming call to `load()`.
            iv. Initialize the lock `pcb_lock`.
            v. Initialize `child_info_structs` to be an empty list.
        ii. Open the executable file in `load()`:
            i. Call `file_deny_write()` on the executable to ensure no write operations can be done on it.
            ii. If we succeed in loading the file, return true. Do not `file_close()` the executable, as closing the executable file will occur when the user process exits via the exit syscall. Access the process's PCB by calling `t->pcb` and set `executable` to the pointer of the file just loaded before returning true.
            iii. Otherwise, `file_close()` the executable and return false.

iii. If the call to `load()` returns true, access the boolean `load_success` on the heap via the provided address and set it to true. Otherwise, set it to false.

iv. Up the semaphore `load_semaphore` on the heap via the provided address.

d. Once the thread running `process_execute()` is able to down the semaphore, observe the value of the boolean `load_success`. If the value is false, loading the file has failed → be sure the PCB created for it is freed before we return -1. Otherwise, return the tid given by the `thread_create()` call. Before returning, free the space allocated to both `load_success` and `load_semaphore`.

3. If the call to `process_execute()` returns -1, return -1. Otherwise, the return value is the pid of the new child process; save this to return at the end of the syscall.
4. We know at this point that loading the executable was successful; add a child info struct to the parent PCB's `child_info_structs`.
    a. Set `child_pid` to the value returned by `thread_create()`.
    b. Set `waited` to false.
    c. Set `exit_status` to -1.
    d. Initialize the semaphore `wait_semaphore` to zero.
5. Release the lock `pcb_lock` obtained at the beginning of this syscall, and return the child process's pid.

wait : `int wait (pid_t pid)`

1. **Error checking**:
    a. `*pid*` *doesn't correspond to a direct child*: we can check for this by iterating over this process's list of child info structs, stored in its PCB struct. For each child struct, check to see if its `child_pid` equals the `pid` we're given. If none of the child info structs have a matching `child_pid`, the `pid` we're given doesn't correspond to a direct child of this process and is therefore invalid → return -1.
    b. *Process pid has already been waited on*: assuming the first error doesn't appear, we have access to the child info struct with `child_pid` = `pid`. Check its `waited` attribute and see if it's true: if so, it's already been waited before and therefore can't be waited on again → return -1.

2. Assuming none of the above errors are found, the parent process should down the `wait_semaphore` in the child info struct with the corresponding `child_pid` = `pid`.
3. Once the down succeeds, we know the child process must be finished; set `waited` within the child info struct to true.
4. Return the exit status found in `exit_status` within the child info struct.

Synchronization

Why do we use `**wait_semaphore**`?
In order for a parent process to wait on a child process, we need to know that the child process has finished execution. Since the wait syscall attempts to down the `wait_semaphore` specific to the desired child process, it won't be able to continue until it is upped (due to it being initialized to zero). Since the only time `wait_semaphore` is upped is after the respective child process uses the exit syscall, the wait syscall must therefore only be able to continue after the child process has exited. If the child process has exited before the parent process waits on it, then the parent process will be able to immediately down the semaphore without delay.

Why do we use `**load_semaphore**`?
We can't allow the parent process to exit from a call to exec until it knows whether the child process successfully loaded its executable. We push this responsibility onto the function `process_execute()` such that it won't return a pid (or -1) until the child process has attempted to load the executable. Similarly to above, since `process_execute()` attempts to down `load_semaphore` before observing `load_success` on the heap and returning a pid, it won't be able to do so until it is upped. Since the only time `load_semaphore` is upped in `start_process()` after `load()` has returned and `load_success` has been set, `process_execute()` must therefore only be able to read `load_success` and continue execution after the child process attempts to load the executable and `load_success` is set appropriately.

Why do we have a `**pcb_lock**`? Why is it only used in exec/exit syscalls but not for wait syscalls?

- We need to ensure the child info struct has had a chance to be created before the child process exits and attempts to modify values on it. This could happen if the child process finishes execution while the original exec syscall is still in

progress.

- If this were the case, since the `parent_exited` value on the child's PCB is initialized to false, the exit syscall would attempt to access its child info struct within the parent PCB. Without this check, it's possible that such a child info struct would not exist by the time this happens, which would prevent the child process from updating values such as its exit status.
- With the lock, since the exec syscall obtains it before starting the new process, in the event that the child process finishes before the exec syscall, it would have to wait for exec to release the lock (at which point, the child info struct has been created).
- If the wait syscall also used `pcb_lock`, there would be deadlock. If we obtain said lock in the wait syscall before the child process exits, its exit syscall will be unable to obtain the lock that wait holds, since wait won't release it until exit runs.
- The wait syscall doesn't need a lock because 1. the parent process must finish the exec syscall before waiting on the child process, so the child struct must exist and 2. once it finds the child info struct, it doesn't modify any values until the child process ups the semaphore—at which point no more edits will ever be made to that child info struct (besides during this wait execution).

Rationale

Why do we initialize `**exit_status**` to -1?
This is a rather arbitrary decision; `exit_status` will only ever be read in the wait syscall, but the wait syscall only gets to this point after `wait_semaphore` is upped by the child process's exit syscall. However, by the time `wait_semaphore` is upped by the child process, it has already overwritten the value of `exit_status` to be its actual exit status. Therefore, the initial value we set `exit_status` to is never relevant.

Why do we keep information about the child processes in the parent? In other words, why use a child info struct?
We do this since processes free their PCB structs when they exit; if we stored information like exit statuses on their respective PCB struct, they would be immediately lost after the struct is freed. Parent processes trying to access these values via a wait syscall would therefore try to access freed memory addresses to find the values of exit statuses, which is unacceptable. Since the parent is the one accessing this information about the child, keeping this info in the parent's PCB ensures that it sticks around even after the child exits and frees its PCB.

What does the `**parent_process**` pointer do?
This allows a process to set the necessary values (exit statuses, upping the `wait_semaphore`, etc.) in their respective child info struct, since the child info struct is located in the parent's PCB.

Why do we keep track if parent process exited or not?
It's possible that the parent process exits before the child process exits, meaning the parent's PCB will already be freed by the time the exit syscall is made. Without this check, the child process would attempt to access a bad memory address in the hopes of updating its information on a nonexistent child info struct, which is unacceptable.

Why do we store the file pointer to the executable in each process's PCB?
As we want to make sure that no modifications are made to the executable while we run the user process, we store the file pointer to the executable in the PCB. If we did not do so, we would be calling `file_deny_write` on the executable during loading without ever closing the executable, or we would be closing the executable before the process had exited. By keeping a file pointer in the PCB, we can easily `fclose` the executable when the process exits, ensuring that the executable is always denying writes while the process is running, and allowing modifications once again (by being closed) only when the process is exited.

File Operation Syscalls

Data Structures and Functions

Modification in `**src/userprog/process.h**`
Since every process has a unique mapping from a file descriptor to a file structure, our design is that every process has a mapping structure to achieve the mapping issue; specifically, each file descriptor is linked to its corresponding file structure for the process. It's important to note that all threads within this process will be referencing this table. We can access the file table by referring to the `pcb`.

```
struct file_descriptor_entry{
 int fd; // file descriptor
 struct file *fl; // address to file struct
 struct list_elem elem; // list entry for the file table to track
each file
} // mapping between an fd and file struct

struct process {
  // ... existing data
  struct list file_table; /* list of file_descriptor_entry elements
for each process*/
};
```

Modification in `**src/userprog/syscall.c**`

Basically, we will implement our file system call according to the system call constant by calling the appropriate function in the library provided in `src/filesys/file.c` and `src/filesys/filesys.c`.

At the beginning of `static void syscall_handler(struct intr_frame* f UNUSED)` function, we add a `lock_acquire(&file_lock)`, and at the end of the function, we add a `lock_release(&file_lock)` to guarantee each operation is atomic.

For the mapping of file descriptor to the file structure, we will also design a mapping function to do this, which will access the file table in the process and return the corresponding file structure.

```
// for file descriptor to file* mapping
struct file* fd2file(int fd);
```

We also create a static lock (to be shared across all threads) to synchronize access to the file system.

```
static struct lock file_lock; // used for synchornization
```

We need to add the initialization of `lock_init(&file_lock);` in `void thread_init(void)` for the initialization of `file_lock`.

Modification in `**src/userprog**``**/process.c**`
We want to create a function that will return an available file descriptor whenever it is called. Each time it's called the function will increment the value of a global variable "counter" and return its value. The process is responsible for managing the file descriptor table, so it makes sense for it to be responsible for generating new file descriptors. The incrementation of `highest_file_descriptor` is done in a critical region in the event that multiple file descriptors are created at the same time.

```
// 0, 1, and 2 are already in use by stdin/stdout/stderr
int highest_file_descriptor = 3; // so we just start by 3.

int generate_available_file_descriptor() {
  // acquire lock
  int temp = highest_file_descriptor++;
  // release lock
  return temp;
}
```

Algorithms

Firstly, we need to track the open file of each thread.
For this part, we will mainly work in `src/userprog/process.h` and `src/userprog/process.c`.
In `struct file_descriptor_entry`, the `fd` represents the file descriptor of the `struct file`, we will construct a mapping relationship in the structure. And in order to open and close files more conveniently, we will use the list structure `file_table` defined in `struct process` to manage the relationship of each file descriptor entry.
In a more specific scenario, when the current thread opens a file by file name, it will create a `struct file_descriptor_entry` for the file, and push it to the back of the file table of the process. By this means, we can manage and track every open file by iterating the file table list, which is convenient.

```
//...
if (args[0] == SYS_OPEN){
  // acquire global filesystem lock here

  int fd = open_file_helper(file_name);

  // do error checking here to check that file was successfully
opened

  f->eax = fd; //return the open file descriptor

  // release the lock here
}
//...
```

```
int open_file_helper(char* file_name){
  // do error checking here to check if file_name isn't null and
isn't too long
  struct thread* cur_thread = thread_current();
  struct file_descriptor_entry* new_entry =
(file_descriptor_entry*)calloc(1, sizeof(file_discriptor_entry));
  int fd = cur_thread->pcb->generate_available_file_descriptor(); //
get the process descriptor for the open file
  new_entry->fd = fd; // mapping the descriptor to the file
  new_entry->file = filesys_open(file_name);
  list_push_back(&(cur_thread->pcb->file_table), new_entry->elem);
  return fd;
}
```

Secondly, we will focus on the implementation of file operation system calls.
For this part, we will mainly work in the `syscall_handler()` method of `src/userprog/syscall.c`.
we have found that the system call constants are defined in `src/lib/syscall-nr.h`.

```
  SYS_CREATE,       /* Create a file. */
  SYS_REMOVE,       /* Delete a file. */
  SYS_OPEN,         /* Open a file. */
  SYS_FILESIZE,     /* Obtain a file's size. */
  SYS_READ,         /* Read from a file. */
  SYS_WRITE,        /* Write to a file. */
  SYS_SEEK,         /* Change position in a file. */
  SYS_TELL,         /* Report current position in a file. */
  SYS_CLOSE,        /* Close a file. */
```

And, we will use `if-else` to implement them in order. It is easy to implement some system calls such as `create()`, `remove()`. But considering the following system call, the argument to represent a file in the function signature is not `file` pointer as the library function, but an integer that represents a file descriptor. Therefore, we need a transition function to map the file descriptor provided by the current thread to the `file` structure.

```
// for file descriptor to file* mapping
struct file* fd2file(int fd){
  // variable declaration
  struct list file_table = thread_current()->pcb->file_table;
  for (file_entry = list_begin (&file_table); file_entry != list_end
(&file_table);
       file_entry = list_next (file_entry))
    {
      struct file_descriptor_entry *fd_entry = list_entry
(file_entry, struct file_entry, elem);
      if (fd_entry->fd == fd) {
        return fd_entry->file;
      }
    }
  return NULL;
}
```

Thirdly, we will concentrate on the synchronization between each thread.
In this part, we mainly work in the `syscall_handler()` method of `src/userprog/syscall.c`.
We will use the `file_lock` defined in each thread to guarantee synchronization.

Specifically, we will initialize our `file_lock` when the thread is initialized. For the file operation syscalls, we will first use `lock_acquire(&file_lock)` to acquire permission to execute the file operation syscall, if there is a thread using the file operation, then the current thread has to wait until the lock is occupied is release, otherwise, it will access the file operation syscall without waiting. At the end of all the file operation syscalls, we will release the lock by `lock_release(&file_lock)`.

```
static void syscall_handle (struct intr_frame* f UNUSED) {
  // Other syscalls

  lock_acquire (&file_lock);
  //...
  if(argv[0] == SYS_CREATE){...}
  // Rest of file operation syscalls
  //...
  lock_release(&file_lock);

  // Other syscalls
}
```

Synchronization

The critical region here is in the `syscall_handler()` method, and the resource will be accessed by a file operation system call invoked by different threads.
To guarantee synchronization, our solution is simple and easily verifiable here. We just add a global `file_lock` at the beginning of the `syscall_handler()` method, so that when there is a thread executing the file operation system call, other threads that want to access the resource will have to wait until the lock is released by the thread. And after the current thread finishes file operation, it will release the lock so that other threads could conduct the file operation.
The `file_lock` is defined in `src/threads/thread.c` using the `struct lock` provided, each thread has a lock to guarantee the atomicity of file operations

Additionally, as mentioned above, the incrementation of `highest_file_descriptor` is done in a critical region in the event that multiple file descriptors are created at the same time.

Rationale

In this task, in order to track all the open files in each thread, at first, we want to use an array of file pointers to achieve this, but this way, we will write a lot of code for the management of the array, some operations such as open a new file or close a file will be exhausting. And we get an idea from project0, in which we get familiar with the list structure of Pintos. Then we use a file list and file descriptor entry to construct the relationship between a file descriptor and file pointer, which is the most reasonable and convenient.
Besides, for the implementation of file operation system calls, we can use the basic file functionalities provided. But here we need to convert the file descriptor to a file by iterating the file table of the current thread, which is also very intuitive.
What's more, or synchronization, we use the global file lock to achieve this. Before the current thread could do file operations, it needs first to acquire a lock, only when the area is not accessed by other threads can the thread go into it. As the document recommended, the global lock is the most convenient way to achieve synchronization.

Floating Point Operations

Data Structures and Functions

Modification in `**src/**``**threads**``**/**``**interrupt**``**.**``**h**` **and** `**src/threads/switch.h**`
Create a 108-byte buffer and add it to the `struct intr_frame` in `interrupt.h` and `struct swit``c``h_treads_frame` in `switch.h`, which tracks the FPU registers.

```
/* Interrupt stack frame. */
struct intr_frame {
  ...
  /* pushed in intr_stub.S to save FPU registers to stack */
  char* fpu_buffer[108];
}
```

```
/* switch_thread()'s stack frame. */
struct switch_threads_frame {
  ...
  char* fpu_buffer[108];
};
```

Modification in `src/**``**threads**``**/**``**thread**``**.**``**c**` and `**src/userprog/process.c**`**

Modify the the `start_process` function in `process.c` and the `thread_create` function in `thread.c` to do FPU state initialization.

```
/* A thread function that loads a user process and starts it
   running. */
static void start_process(void* file_name_) {
  /* initialize fpu registers */
  asm volatile("fninit");
  /* save the state to the memory location of fpu_buffer struct*/
  asm volatile("fsave %0" : : "g"(&if_.fpu_buffer));
  /* before switching to user mode */
  asm volatile("movl %0, %%esp; jmp intr_exit" : : "g"(&if_) :
"memory");

}
```

```
tid_t thread_create(const char* name, int priority, thread_func*
function, void* aux) {
// save FPU state of current thread
  char temp[108];
  asm volatile("fsave %0" : : "g"(&temp));
  // create clean FPU state in registers
  asm volatile("fninit");
  // copy clean state to switch_threads_frame
  asm volatile("fsave %0" : : "g"(&sf->fpu_buffer));
  // Restore FPU state of current thread
  asm volatile("frstor %0" : : "g"(&temp));

  /* Add to run queue. */
```

```
  thread_unblock(t);
  return tid;
}
```

Modification in `**src**``**/threads/intr-stubs.S**`

Modify assembly code in `intr_entry` and `intr_exit` to `fsave()` and `frestore()` fpu registers on interrupts.

```
intr_entry:
        /* Save caller's registers. */
        pushl %ds
        pushl %es
        pushl %fs
        pushl %gs
        pushal

        /* Save FPU registers to the stack */
        subl $108 %esp
        fsave %esp

intr_exit:
        /* Restore FPU registers */

        frstore %esp
        addl  $108 esp

        popal
        popl %gs
        popl %fs
        popl %es
        popl %ds
```

Modification in `**src/**``**threads/switch.S**`

Also modify assembly code in `switch_threads` function in `switch.S` to `fsave()` and `frestore()` fpu registers on thread switches.

```
switch_threads:
```

```
        # Save caller's register state.
        #
        # Note that the SVR4 ABI allows us to destroy %eax, %ecx,
%edx,
        # but requires us to preserve %ebx, %ebp, %esi, %edi.  See
        # [SysV-ABI-386] pages 3-11 and 3-12 for details.
        #
        # This stack frame must match the one set up by
thread_create()
        # in size.
        pushl %ebx
        pushl %ebp
        pushl %esi
        pushl %edi

        # Save FPU registers to the stack
        subl $108 %esp
        fsave %esp
        ...

        # Save FPU registers from the stack
        frstore %esp
        addl  $108 esp
        popl %edi
        popl %esi
        popl %ebp
        popl %ebx
        ret
```

Algorithms

1. Initializing the buffer
    a. In `start.``s` set the CR0_EM bit and update this in `threads/start.S` to remove the flag and indicate to the processor that the FPU is present.
    b. Add a 108-byte buffer to both the `struct intr_frame` in `interrupt.h`

and the `struct switch_threads_frame` in `switch.h`

c. To do the fpu state initialization, we will need to modify the `thread_create` function in `thread.c` and the `start_process` function in `process.c`
d. Use `finit` to initialize the FPU registers, use `asm_volatile(finit);` in `start_process` in `process.c` and `thread_create` in `thread.c`

2. Modify assembly code `int``r``-stub``s``.S` and `switch.S`
    a. Modify the assembly of the `intr_entry` and `intr_exit` functions in `intr-stubs.S` and the `switch_threads` function in `switch.S` in order to save FPU registers on thread or context switch.
    b. `fsave` copies the entire FPU state (108 bytes large) into a provided memory address. This address can be on the stack or anywhere else. Literally, any address works.
    c. `frestore` restores the saved FPU state registers
3. For syscall, add an if statement (`if args[0] == SYS_COMPUTE_E`) in the `sy``s``call.c` for the `compute_e` syscall. We will use `sys_sum_to_e` from `lib/float.c` to calculate, and store the result in `f→eax` to return.

Synchronization

We don't have to add anything for synchronization in this scenario. It is inherently synchronized because we add assembly code to `fsave()` and `frestore()` on thread switches - this code is run during thread context switches. The code we are adding is to ensure FPU synchronization on thread switches, and it is only called in `switch.S` and `intr-stub.S` which is to ensure saving registers and loading on thread switches or context switches. Only one thread at a time can access the FPU, and because we save and restore on thread switches, it doesn't need synchronization.

Rationale

This solution is optimal because we are not writing a separate assembly file, rather we are appending to the given `intr_stub.S` and `switch.S`. This is inherently handled by the OS when switching threads or context switches. All we are doing is adding to the given struct which saves the register state and changing the assembly code to save the state to the stack before switching threads, and we are initializing the FPU registers state on

`start_process` and `thread_create`.

Concept check

---

1. `sc-bad-sp` test case uses an invalid stack pointer (%esp) when making a syscall. The test uses `asm volatile` to embed an assembly line code within the C code on line 16 of `sc-bad-sp.c` file. Within the assembly line code itself, the `movl $.-(64*1024*1024), %esp` line moves the immediate value `-(64*1024*1024)` into the stack pointer (%esp). This value is 64MB below the code segment, so is an invalid address since it is an unmapped address. Then the `int $0x30` requests a syscall. Because the stack pointer is set to an invalid address, the process must exit with the -1 exit code, instead of crashing.
2. `sc-boundary-3` test case uses a valid stack pointer when making a syscall, but because it is too close to a page boundary, some of the arguments are located in invalid memory. In the test case, `char *p` (system call number pointer) is set to the result of `get_bad_boundary()` which returns an invalid address where the preceding byte is valid. By decrementing `p` pointer by 1, we then set p to be a valid address at the first byte, but the remaining addresses are invalid. So even though we set `*p` or the actual system call number to 100, we will not be able to access it. With the line, `movl %0, %%esp;`, we set the stack pointer to a valid address of 0, and `int $0x30` requests a syscall. But since the system call number pointer that we placed on the stack points to an address where only the first byte is a valid address, but the rest are not, the process should get killed.
3. We think that these test cases are missing: the tests for the correctness of the `tell()` and `seek()` operations. In all the test cases, the seek or tell methods are called and the results (correct or not) are printed directly, without verifying the correctness of the results.
   Our test is designed to set the file pos by seek(), and then print the current pos of the file by `tell()` , and if the result of both prints is the same, from the previous `pos` of the file to the pos we specified, then the `seek()` and `tell()` operation is correct.