# Project 1: User Programs Design Doc

## Group Members

Xinyu Fu (fuxy@berkeley.edu)
James Tang (jamestang@berkeley.edu)
Lanxiang Hu (hlxde1@berkeley.edu)

## Task 1: Argument Passing

### Data Structures and Functions

We will modify the `start_process` function to parse command-line arguments and push them onto the stack for the user process.

#### `userprog/process.c` :

```
static void start_process (void *file_name_);
```

### Algorithms

We plan to use `strtok_r` in `src/lib/string.c` to parse out the first argument (argv[0]) to pass into the `load` function as the `filename` argument as well as the rest of the command-line arguments. If `load` is successfully executed, we start pushing to the stack (by modifying the `intr_frame` struct using `memcpy` ) in this order:

1. command-line arguments
2. stack-alignment space (0's)
3. a null pointer sentinel
4. pointers to the arguments (from right to left)
5. argv pointer
6. argc
7. fake return address (0)

With all necessary elements pushed onto the stack (through the `intr_frame` struct), `start_process` is now able to simulate a return from interrupt and enter a user process with arguments passed correctly.

### Synchronization

We don't have synchronization issues in this task. In `start_process` , we first parse `file_name` , then push the corresponding data onto the stack. Since `fn_copy` is a copy of the `file_name` in the function `process_execute()` , we have no shared data between threads. Also, since each thread has its own stack, when we modify the stack for one thread, other threads won't be affected. Therefore we don't have synchronization issues.

### Rationale

Since we need to process multiple command-line arguments separated by spaces, we need to tokenize the input string. We parse out the first argument of the `argv` to load the executable. Then, the rest of the arguments need to be pushed from right to left onto the stack, since after the function call they will be popped off the stack in the original order. This way of managing the stack makes sure that there will be no page faults as the user process retrieves the arguments.

## Task 2: Process Control Syscalls

### Data Structures and Functions:

#### `userprog/syscall.c`:

Add handlers for all syscalls.

```
static void syscall_handler(struct intr_frame *f UNUSED)
```

Add a function to verify pointers (null pointers, invalid pointers which point to unmapped memory locations, or pointers to the kernel's virtual address space).

```
bool valid_ptr(void *ptr)
```

#### `userprog/process.c`:

Modify this function to initialize `wait_info` of the child thread, insert it into the list of child wait info, then pass it to the child thread through `thread_create()`. Wait for the child thread to load before proceeding.

```
tid_t process_execute (const char *filename);
```

Modify this function to update `loaded` in `wait_info` and up the semaphore.

```
static void start_process (void *file_name_);
```

Modify this function for SYS_WAIT. Find the `wait_info` struct that has a match with the tid in the list of `children_wait_info` and wait for the child to exit. Otherwise, return -1.

```
int process_wait (tid_t child_tid)
```

Modify this function to up the semaphore and unblock the parent process when exiting.

```
void process_exit (void)
```

#### `threads/thread.h`:

Add struct for parent/child thread communication.

```
typedef struct wait_info
{
  struct semaphore sema;     // Semaphore shared between the parent and child thread.
  int ref_cnt;             // keeps track of how many threads have reference to this struct,
   free memory when it is 0.
  struct lock lock;         // allows mutual exclusion when modifying members in the struct
  int exit_code;                // Pointer to the child's exit_code.
  bool loaded;                   // whether the child process have successfully loaded executable
  bool waited;                        // whether the parent is has waited for the child
  tid_t tid;                   // tid of the child thread
  struct list_elem elem;           // for creating a list of children_wait_info
}
```

Add a `wait_info` struct to the thread struct.

```
typedef struct thread
{
  struct wait_info* wait_info;      // wait_info struct for current thread
  struct list children_wait_info;  // list of wait_info for children threads
}
```

**`threads/thread.c`:**

Add code to initialize wait_info to `NULL` of the thread.

```
static void init_thread (struct thread *t, const char *name, int priority)
```

Save exit code to `wait_info`. Decrement `ref_cnt` by 1 for each wait_info and free it if `ref_cnt` is 0.

```
void thread_exit (void)
```

## Algorithms

We first check if the stack pointer and pointers provided by the user process are valid by calling `valid_ptr`. If we find that the pointer is invalid, we unlock the locks, free memory, then exit the thread. Otherwise, we execute the syscall handler based on the syscall number.

### Practice()

Increment integer in `args[1]` by 1, then save it in `f->eax`.

### Halt():

Call `shutdown_power_off()` to terminate.

### Exit():

Save exit code to `wait_info`. Decrement `ref_cnt` in `wait_info` by 1. Decrement `ref_cnt` by 1 in the list of children_wait_info as well, and free them if their `ref_cnt` is 0. Then terminate the thread.

### Exec():

Initializes a struct called `wait_info` defined in `thread.h`. It contains a semaphore, a lock, a reference counter, the child exit code, and a bool to indicate if the child has successfully loaded the executable. The parent adds the new `wait_info` to the children list and downs the semaphore in `process_execute` after creating the child thread. The child changes the bool `loaded` after loading the executable in `start_process` and ups the semaphore in order to signal the waiting parent process. If tid gives an error or if the load is unsuccessful, return -1. Otherwise return pid.

### Wait():

We first check if the pid provided is a direct child of the parent process by iterating through the list of children_wait_info in `wait_info` struct. If pid is not the child of the parent process or the parent has already waited once (`wait_info->waited == 1`), we return -1. Otherwise, we change `waited` to True, use the semaphore in `wait_info` for the parent process to wait for the child process (using `process_wait(tid_t)` ) until the child ups the semaphore and returns the exit code. In the case that the child thread is killed by the kernel, the kernel calls `thread_exit()` that ups the semaphore, then the parent will finish waiting and return the exit code -1.

### Synchronization

The `wait_info` struct is shared data between the parent and child threads. We designed the struct with a lock so that when a thread needs to modify `wait_info` struct data, it needs to acquire a lock to do so, which prevents race conditions. When one thread is writing to the struct, another thread cannot read from or write to it until the lock is released. Although this synchronization strategy causes some threads to wait for others which may slow down processes, it is required because we need to keep the shared resources consistent for all threads.

### Rationale

We use a struct for communication between threads because this is relatively simple to conceptualize and it exists in the heap for both the parent and child threads to access. Coding should be straightforward, except that we need to handle edge cases in syscalls appropriately as well as to allocate and deallocate memory for the shared data, which could be a mess if we are not careful. Initializing the `wait_info` structs and freeing them should be constant time.

## Task 3: File Operation Syscalls

### Data Structures and Functions

#### `userprog/process.c` :

Modify to deny write to an executable after it is loaded.

```
bool load (const char *file_name, void (**eip), void (**esp))
```

Modify to allow write to the executable after process exits.

```
void process_exit (void)
```

#### `userprog/syscall.c` :

Add a global lock for the file system.

```
static struct lock global_file_lock;
```

Modify to initialize a global lock for file syscalls.

```
void syscall_init (void)
```

Modify to handle file syscalls.

```
static void syscall_handler (struct intr_frame *f)
```

`threads/thread.h`:

Add a file descriptor table to the thread struct.

```
struct thread
{
  ...
  struct file* fd_arr[100];
  ...
}
```

`threads/thread.c`:

Modify to initialize the thread's file descriptor table where the fd 0 and 1 are set to stdin, stdout. The rest should be initialized to NULL.

```
static void init_thread (struct thread* t, const char* name, int priority)
```

Modify to deallocate memory of the file descriptor table when thread exits.

```
void thread_exit (void)
```

## Algorithm

We added a global file system lock. File syscalls will acquire this lock when they need to use file system functions. This way we ensure that file syscalls do not call multiple file system functions concurrently. When a file syscalls is called, `syscall_handler()` execute as specified:

### Create:

Call `filesys_create()` in filesys/filesys.c to create a new file and store its return value to `f->eax`.

### Remove

Call `filesys_remove()` in filesys/filesys.c and store its return value to `f->eax`.

### Open:

Call `filesys_open ()` to open the file and add the file to the file descriptor table `fd_arr` by iterating through it and adding to the first NULL position. Store the fd (its index) to `f->eax`.

### Filesize

Access the file at `fd_arr[fd]` and call `file_length()` in filesys/file.c. Store the return value to `f->eax`.

### Read

If fd is 0, we will call `input_getc()` in a while loop until a null terminator is inputed. If fd is not 0, we access the file at `fd_arr[fd]`. If `fd_arr[fd] == NULL`, store -1 into `f->eax`. Otherwise, call `file_read(*file, *buffer, size)` to read the file and store the return value to `f->eax`.

### Write

If fd is 1, write to stdout using putbuf() and store `size` to `f->eax`. Otherwise, we access the file at `fd_arr[fd]` and call `file_write(*file,*buffer,size)` and store the return value to `f-> eax`.

### Seek

Access the file at `fd_arr[fd]`. Call `file_seek(*file, new_pos)` to set the position.

### Tell

Access the file at `fd_arr[fd]`. Call `file_tell(*file)` and store its return value to `f->eax`.

### Close

Access the file at `fd_arr[fd]`. Call `file_close(file)` and set `fd_arr[fd]` to NULL.

## Synchronization

We handled synchronization using a global lock `global_file_lock`. When one thread makes a syscall to the file system, it acquires this lock and releases it afterwards so that file syscalls don't call multiple file system functions concurrently.
We also make sure that when the user process is running, nobody can modify the executable on disk, by calling `file_deny_write()` when the executable is loaded, and calling `file_allow_write()` when the process exits.

## Rationale

The use of `global_file_lock` is a very coarse locking strategy and certainly slows down our system since we are only allowing one file syscall to execute at a time. Hopefully this could be improved later on in class.
Furthermore, we choose to use an array as the file descriptor table since file descriptors are integers which map nicely to an array, and accessing the table is constant time operation. We have also considered using a list data structure, but the linear access time makes it less favorable. However, arrays are fixed-size and that means there is an upper limit on the number of files we can open per thread. Therefore, we may change it later on if this limitation conflicts with requirements of the operating system.

# Additional Questions

1. In `sc-bad-sp.c` file, on line 16. `movl $.-(64*1024*1024), %esp;` will set stack pointer to be a negative number and therefore pointing to a negative address. As a result, this stack pointer is invalid when making a syscall.

2. In `sc-bad-arg` file, on line 12. `movl $0xbffffffc, %%esp` will push the stack pointer to the boundary of the user space. As a result, the syscall arguments will be in the kernel space, which is invalid memory

3. In the test suite provided, it examines for example whether the process is terminated with -1 exit code with invalid inputs, however, since the project requirements explicitly ask for proper memory management, it is also crucial to make sure that the process doesn't "leak" memory once an invalid pointer which is dereferenced from the user is rejected by the kernel or other running process. We need to test that we have appropriately freed all resources in accessing user memory and there is no subsequent memory leak with gdb and Valgrind.