CS 342 Design Document
Project 2D: Implementing System Calls

Rajat Kateja (r.kateja@iitg.ernet.in)
Mathematics and Computing
10012338

TA: Lalatendu Behera
Chetti Prasad

----DATA STRUCTURES----
The following data structures were introduced:

In thread.h

Added the following to struct thread

```
struct list fd_list;              /* File Descriptor List */
struct lock child_lock;           /* Lock for the parent to wait on child */
bool child_load_failed;           /* Boolean to store if the child load failed */
struct list dead_child_list;      /* Store a list of dead children of a process */
bool waited_on_by_parent;         /* Did parent thread already wait on this thread */
```

Introduced the following data structure:

Structure fd is the file descriptor which has three fields, first is an integer related to each open file for each thread, next is the file pointer, and last is the list elem so that the instances of this structure can be inserted into the struct list fd_list of each thread.

```
struct fd
 {
        int file_no;
        struct file *file;
        struct list_elem elem;
 };
```

Structure dead_thread stores the dead child thread of a parent thread which may be used by the parent thread to wait on the child thread (i.e. return with the child's exit status) even after the child thread has exited. It has three fields, first is the tid of the thread to identify the thread, next is the exit status of the thread and last is the list_elem so that instances of this struct can be inserted into the dead_child_list of each of the threads.

```
struct dead_thread
 {
        int tid;
        int exit_status;
        struct list_elem elem;
 };
```

----ALGORITHMS----

The following algorithms were used to implement the various system calls:

First of all in the syscall handler, the f->esp pointer is checked to be non NULL, must belong to the user valid address space and must be a valid page in the page table of the user program (checked by lookup_page function). If any of these tests fails, then the thread is exited with a -1 exit status.

Wrote a function **create(int no_of_args, struct frame *f)**. This function checks that the no_of_args arguments in the frame stack are all valid, the three tests mentioned above are applied to each of them, and if any of that fails, it returns false. In each of the system calls, depending on the number of arguments that the system call receives, the function is called along with the frame. If the return is false, **quit** function is called (implemented for this project) which makes the thread exit with exit status -1.

Halt
This system call simply switches off the system by calling power_off() function.

Exit
It takes the exit status of the thread, prints that the thread is exiting with the given exit status in the required format, sets the return value to the exit status, and then it pushes the thread into its parents dead_thread_list, so that it can be waited upon by the parent even after it is dead. Finally thred_exit() is called to make the thread exit.

Exec
This function is used to create new child functions, the basic functionality achieved by calling the process_execute () function with the received argument of the process, which does the required job of creating the new thread and putting it in the ready queue. Then the function puts the parent thread in the child_lock waiting list, and blocks itself. This is used as a synchronisation because it is required that the parent cannot return from this function until the child has successfully loaded. So now the parent goes into block state and is in the child_lock lock's waiting list. From here, the parent is woken up by the child only after its loading is finished (in process.c start_process function). Here only, the requirement that is child loading has failed, the function must return -1 is met. If the loading of the child has failed, in the start_process function the child thread sets the child_load_failed Boolean field in the parent thread to be true. Next when the thread is unblocked from the child_lock waiting list, it checks for this field. If it is set ot be true, the return value is set to be -1. Else the return value is the value returned by process_execute function which is nothing but the exit status of the child thread.

Wait
This system call is used to wait on a child thread by the parent thread. This functionality is basically achieved by calling the process_wait() function in process.c where all the requirements are takem care of. The following is performed in the process_wait function:

- First the tid is checked, in the all_list (where all threads are stored) by using the function get_thread_by_id which was made for the previous implementation as well (project 2B and 2C). If a valid thread is not found in the list, then the calling thread (i.e. parent thread's) dead_child_list is checked. If the thread is found there, the return value is set to the exit status of the thread, the child thread is removed from the list (since it has already been waited upon once), the the function returns. If the thread is not found even in the dead_thread_list, then the function returns with -1 as required.

- If the thread was found in the all_list of thread, then it is checked that the tid must be of one of the parent's child threads (by iteratively checking the parents child_list). If thread is not found in the child_list, function returns with -1. Also if the thread has already been waited upon by the parent (checked by using the bool waited_on_by_parent field of the thread), then also the function returns with -1 status. All theses are as per the requirements specified in the pintos manual.
- If till this stage, the function has not returned, means we need to now wait on the given child thead. First its waited_on_by_parent is set to true (fir future reference), and the parent's waiting_on_child is set to be this child thread. The the parent thread is blocked.
- The parent thread is then woken up by the child thread in it's thread_exit function, by checking that the parent is non-NULL, and it's waitin_on_child field.
- Once the thread wakes up, it checks its dead_child_list and removes this thread from the list. This step is done because before waking up the parent the child thread puts itself in the paren's dead_child_list (in the exit funcion), and this thread has already been waited upon, so it must not be in the dead_child_list.
- Finally the parent thread returns from the function.

Create
This sytem call is used to create a new file in the file system. This is done using the function filesys_create() provided in the filesys.c file using the appropriate arguments as passed to this funciton. The return value is also the same as the one returned by the mentioned function.

Remove
This function is the opposite of create function, used to remove a file from the file system. This is achieved using the filesys_remove function with the appropriate arguments.

Open
This system call opens the file specified in the file name received as the argument. It first preforms the routine checks on the arguments, and returns with exit status -1, if any of the tests fails. Then it opens the file using filesys_open function which returns a file pointer. Next task is to put this file in the file descriptor of the thread. For this, first the fd_list of the thread is checked. If it is empty, means this is the first file that is being opened by the thread. In this case, the file_no of this file descriptor is set to 3 (since 0,1 and 2 are reserved). If the list is non-empty, then the file_no is set to be one larger than the previous file_no. Finally it is checked that the file opened is not the thread's executable or its parent's executable. If either of the above is true, then the file is rendered non-writeable using the fi;e_deny_write() function.

Filesize
This function receices a file number as argument. This file number is transformed to file pointer from the fd_list of the thread using the get_fd_by_no() function (implemented). Then the file_length() function provided in file.c is used to get the sixe of the file, which is returned.

Read
This function also received a file number as argument (along with size and buffer). This number is used to get the file descriptor from the fd_list of the thread using the get_fd_by_no() function. Then if the number is 0, which is the case of stdin, then input_getc() function is used. In other cases file_read function is used by passing the appropriate arguments.

Write
This function is similar to the read function, only change is that it is used to write on the file provided by the number. Like before first the file descriptor is found, then the appropriate function is called, i.e. putbuf in case of stdout and file_write in all other cases.

Seek
This is used to set the read/write position in the file. It is implemented using the file_seek function provided in file.c.

Tell
It is used to get the read/write position in the file, which is found from the file_tell function.

Close
This is used to close the file preciouslt opened. Firstly the file pointer is found from the file descriptor of the file (found using get_fd_by_no function), then the instance of this file is removed from the fd_list of the thread and then the file is closed using the file_close function.

One particular implementation is in the **exception.c** file, where in case of a page fault exception, if the page faule was cause due to a NULL pointer being dereferenced or the pointer not being a valid used addredd, then the thread quits with -1 status by calling the quit function. This is done to handle the cases where a bad read or write is attempted.

----SYNCHRONISATION----

A parent must not return from the exec system call until the child has been loaded (irrespective whether successfully or unsuccessfully). This is achieved by using the child_lock. The parent is inserted in this lock's waiting list and blocked. Then the child acquires the lock in the start of it's start_process and releases the lock only after it has returned from the load function, thereby unblocking the parent thread.

----RATIONALE----

1. The code is perfectly synchronized,so even in case of arbitrary switches the deadlock situation will not occur.
2. Since it does not pass the multi-oom test,it is likely that the in case of overload of memory kernel may panic or crash (since the process may run out of memory) .

----CODE SNIPPETS-----
Here I am providing the code snippets of the system calls, and the functions written in the syscall.c file.

```
static void
syscall_handler (struct intr_frame *f)
{
	/**********************************************
	Implementing system calls for project 2B
	*********************************************/
	struct thread *cur = thread_current();
	uint32_t *valid = lookup_page(cur->pagedir, f->esp, false);
	if(f->esp == NULL || !is_user_vaddr (f->esp) || valid == NULL)
		quit(f);

	int syscall = * (int *)(f->esp);

	switch(syscall)
	{
		case(SYS_HALT):
		{
			power_off();
			break;
		}
		case(SYS_EXIT):
		{
			if(!check(1, f))
				quit(f);
			int status = * (int *) (f->esp + 4);
			f->eax = status;
			printf("%s: exit(%d)\n", cur->name, status);
			struct dead_thread *thread_to_die = (struct dead_thread
*)malloc(sizeof(struct dead_thread));
			thread_to_die->exit_status = status;
			thread_to_die->tid = thread_current()->tid;
			list_push_front(&thread_current()->parent->dead_child_list,
&thread_to_die->elem);
			thread_exit();
			break;
		}
		case(SYS_EXEC):
		{
			if(!check(1, f))
				quit(f);
			char *process = * (char **) (f->esp + 4);
			if(process == NULL || !is_user_vaddr (process) || lookup_page(cur-
>pagedir, process, false) == NULL)
				quit(f);
			f->eax = process_execute(process);
			list_push_front(&thread_current()->child_lock.waiters, &thread_current()-
>elem);
			thread_block();
			if(thread_current()->child_load_failed == true)
				f->eax = -1;
			break;
		}
		case(SYS_WAIT):
		{
			if(!check(1, f))
				quit(f);
			int tid = * (int *) (f->esp + 4);
```

```
            f->eax = process_wait(tid);
            break;
        }
        case(SYS_CREATE):
        {
            if(!check(2, f))
                quit(f);
            char *file = * (char **) (f->esp + 4);
            if(file == NULL || !is_user_vaddr (file) || lookup_page(cur->pagedir, file,
false) == NULL)
                quit(f);
            int file_size = * (int *) (f->esp + 8);
            f->eax = filesys_create(file, file_size);
            break;
        }
        case(SYS_REMOVE):
        {
            if(!check(1, f))
                quit(f);
            char *file = * (char **) (f->esp + 4);
            if(file == NULL || !is_user_vaddr (file) || lookup_page(cur->pagedir, file,
false) == NULL)
                quit(f);
            f->eax = filesys_remove(file);
            break;
        }
        case(SYS_OPEN):
        {
            if(!check(1, f))
                quit(f);
            char *file_name = * (char **) (f->esp + 4);
            struct file *file;
            if(file_name == NULL || !is_user_vaddr (file_name) || lookup_page(cur-
>pagedir, file_name, false) == NULL)
                quit(f);
            file = filesys_open(file_name);
            if(file == NULL)
                f->eax = -1;
            else
            {
                if(!list_empty(&cur->fd_list))
                {
                    int prev_fd_no = list_entry(list_back(&cur->fd_list), struct
fd, elem)->file_no;
                    struct fd *new_fd = (struct fd *)malloc(sizeof(struct fd));
                    new_fd->file_no = prev_fd_no+1;
                    new_fd->file = file;
                    list_push_back(&cur->fd_list, &new_fd->elem);
                    f->eax = new_fd->file_no;
                }
                else
```

```
				{
					struct fd *new_fd = (struct fd *)malloc(sizeof(struct fd));
					new_fd->file_no = 3;
					new_fd->file = file;
					list_push_back(&cur->fd_list, &new_fd->elem);
					f->eax = new_fd->file_no;
				}
			}
			if(strcmp(file_name, thread_current()->name) == 0 || strcmp(file_name, 
thread_current()->parent->name) == 0)
			{
				file_deny_write(file);
			}
			break;
		}
		case(SYS_FILESIZE):
		{
			if(!check(1, f))
				quit(f);
			int fd_no = * (int *) (f->esp + 4);
			struct fd *cur_fd = get_fd_by_no(fd_no);
			if(cur_fd != NULL)
				f->eax = file_length(cur_fd->file);
			break;
		}
		case(SYS_READ):
		{
			if(!check(3, f))
				quit(f);
			int fd_no = * (int *) (f->esp + 4);
			char *buffer = * (char **) (f->esp + 8);
			if(buffer == NULL || !is_user_vaddr (buffer) || lookup_page(cur->pagedir, 
buffer, false) == NULL)
				quit(f);
			int size = * (int *) (f->esp + 12);
			struct fd *cur_fd = get_fd_by_no(fd_no);
			if(cur_fd != NULL)
			{
				f->eax = file_read(cur_fd->file, buffer, size);
			}
			else if(fd_no == 0)
			{
				input_getc();
				f->eax = size;
			}
			break;
		}
		case(SYS_WRITE):
		{
			if(!check(3, f))
				quit(f);
```

```
                    int fd_no = * (int *) (f->esp + 4);
                    char *buffer = * (char **) (f->esp + 8);
                    if(buffer == NULL || !is_user_vaddr (buffer) || lookup_page(cur->pagedir,
buffer, false) == NULL)
                              quit(f);
                    int size = * (int *) (f->esp + 12);
                    struct fd *cur_fd = get_fd_by_no(fd_no);
                    if(cur_fd != NULL)
                    {
                              f->eax = file_write(cur_fd->file, buffer, size);
                    }
                    else if(fd_no == 1)
                    {
                              putbuf(buffer, size);
                              f->eax = size;
                    }

                    break;
          }
          case(SYS_SEEK):
          {
                    if(!check(2, f))
                              quit(f);
                    int fd_no = * (int *) (f->esp + 4);
                    unsigned position = * (unsigned *) (f->esp + 8);
                    struct fd *cur_fd = get_fd_by_no(fd_no);
                    if(cur_fd != NULL)
                              file_seek(cur_fd->file, position);
                    break;
          }
          case(SYS_TELL):
          {
                    if(!check(1, f))
                              quit(f);
                    int fd_no = * (int *) (f->esp + 4);
                    struct fd *cur_fd = get_fd_by_no(fd_no);
                    if(cur_fd != NULL)
                              f->eax = file_tell(cur_fd->file);
                    break;
          }
          case(SYS_CLOSE):
          {
                    if(!check(1, f))
                              quit(f);
                    int fd_no = * (int *) (f->esp + 4);
                    struct fd *cur_fd = get_fd_by_no(fd_no);
                    if(cur_fd != NULL)
                    {
                              list_remove(&cur_fd->elem);
                              file_close(cur_fd->file);
                    }
```

```
                        break;
                }
        }

        //---------added code ends-------------------//

}
struct fd * get_fd_by_no(int no)
{
        struct thread *cur = thread_current();
        struct list_elem *e;
        struct fd *cur_fd;
        for(e=list_begin(&cur->fd_list);e!=list_end(&cur->fd_list);e=list_next(e))
        {
                cur_fd = list_entry(e, struct fd, elem);
                if(cur_fd->file_no == no)
                        break;
        }
        if(e==list_end(&cur->fd_list))
                return NULL;
        else
                return cur_fd;
}
bool check(int no_of_args, struct intr_frame *f)
{
        int i;
        for(i=0;i<no_of_args;i++)
        {
                if(!is_user_vaddr (f->esp+4*(i+1)) || lookup_page(thread_current()->pagedir, f-
>esp+4*(i+1), false) == NULL)
                        return false;
        }
        return true;
}
void quit(struct intr_frame *f)
{
        f->eax = -1;
        printf("%s: exit(%d)\n", thread_current()->name, -1);
        thread_exit();
}
```