Data Structure

process.c

In process_execute:
char *saveptr; // this *char pointer* variable is used internally by **strtok_r**() in order to maintain context between successive calls that parse the same string

in release_child:
int new_ref // this int variable is used to hold on to the reference which is held by both the parent, in its `children' list, and by the child, in its `wait_status'. we keep track of it so we can check to see if its unreferenced (child and parent both dead) in order to free the process.

in process_wait:
struct thread *cur //this thread pointer variable is used to point at the current thread.

struct list_elem *element; //this list_elem pointer is used to point at the children thread that is at the beginning of the list. it is used in the for loop to get the next thread in the list of thread children

struct wait_status *cs //this wait_status pointer is used to track the completion of a process

int exit_num //this int variable is used to hold on to the child exit code if dead

in process_exit:
struct list_elem *element; //this list_elem pointer is used to point at the children thread that is at the beginning of the list. it is used in the for loop to get the next thread in the list of thread children

struct list_elem *next; //this list_elem pointer is used to point at the threads that have been removed from the element pointer variable

struct wait_status *cs //this wait_status pointer is used to track the completion of a process

syscall.c

char *new_file //used to hold on to a copy of a file

bool success //used to hold on to weather the program has been successfully loaded or not

struct file_descriptor *file //used to hold on to the file descriptor

struct thread *cur //this thread pointer variable is used to point at the current thread

struct list_elem *element //this list_elem pointer is used to point at the children thread that is at the beginning of the list. it is used in the for loop to get the next thread
in the list of thread children

int h //used to hold on to the handle parameter

int size //used to hold on to the size(length) of the file

size_t read //used to hold on to the number of bytes to be read

Algorithm

	process.c	
Function	**what the function does**	**steps taken to minimize the time taken to execute**
tid_t process_execute (const char *file_name);	This function starts a new thread running a user program loaded from file_name. copies file_name to thread_name waits for the new user program to be loaded successfully by the start_process function checks whether the execution status is successful, if it is successful it puts the wait_status of the child process into the children list if the execution status is not successful then it initialises tid to TID_ERROR	The steps I took in this function was by using strlcpy() to copy the file_name into the thread_name and strtok_r() so that command-line arguments were not included in the thread_name. using this instead of a while loop minimized the time spent executing my code Each call to **strtok_r**() returns a pointer to a null-terminated string containing the next token. This string does not include the delimiting byte. If no more tokens are found, **strtok_r**() returns NULL The function strlcpy(), copies up to *size* - 1 characters from the NUL-terminated string *src* to *dst*, NUL-terminating the result.
static void start_process (void *exec_);	this function loads a user process and starts it running if loading is successful it locates kernel memory space to hold the wait_status using malloc properly initializes the struct wait_status stop the parent process from waiting for the child process to be loaded	in this function using malloc instead of calloc helped to lower the time taken to execute my code calloc() zero-initializes the buffer while malloc() leaves the memory uninitialized. Zeroing out the memory may take a little time when executing my code

static void release_child (struct wait_status *cs);	this function releases one reference to cs and if it is unreferenced (child and parent both dead) it frees it	the steps I used in this function were using lock_acquire() and lock_release() lock_acquire() Allows the process to lock say for example files in a stream. Locking the file prevents other process from delivering change sets that modify it lock_release() Releases locks for example on a file in a stream that was locked by this process.
int process_wait (tid_t child_tid);	This function waits for thread tid to die and returns its exit status. process_wait goes through each child one by one checking if a child process is the one to be waited for if so it waits for the child process to become dead, gets the exit code from wait_status of the child, releases the child then return the exit code	the step I used was by using semaphore to control access this made my kernel code execute faster
void process_exit (void);	This function frees the current process's resources. It waits for the current process to become dead then releases it. Notifies the parent that this process is dead. then it goes through the child list releasing a reference to the wait_status of each child	the step I used was by using semaphore to control access this made my kernel code execute faster

	syscall.c	
Function	**what the function does**	**steps taken to minimize the time taken to execute**
static int sys_halt (void)	this function terminates pintos by calling shutdown_power_off() which is declared in devices/shutdown.h	
static int sys_exit(int exit_code)	This function terminates the current user program and returning the exit value to the kernel. If we get a value of 0 it indicates the termination was successful and anything else as error.	
static int sys_exec (const char *ufile)	This function starts the exacution of a user program and returns a valid pid of the child process if it is successful.	the steps I used in this function were using lock_acquire() and lock_release() lock_acquire() Allows the process to lock say for example files in a stream. Locking the file prevents other process from delivering change sets that modify it lock_release() Releases locks for example on a file in a stream that was locked by this process.
static int sys_wait (tid_t child)	this function waits for a child process and retrieves the child's exit value.	
static int sys_create (const char *ufile, unsigned initial_size)	this function creates a new file and returns true if it is successful and false otherwise.	the steps I used in this function were using lock_acquire() and lock_release() lock_acquire() Allows the process to lock say for example files in a stream. Locking the file prevents other process from delivering change sets that modify it lock_release() Releases locks for example on a file in a stream that was locked by this process.

static int sys_remove (const char *ufile)	this function deletes a file and returns true if it is successful and false otherwise	the steps I used in this function were using lock_acquire() and lock_release() lock_acquire() Allows the process to lock say for example files in a stream. Locking the file prevents other process from delivering change sets that modify it lock_release() Releases locks for example on a file in a stream that was locked by this process.
static int sys_open (const char *ufile)	this function opens a file and returns the corresponding file descriptor using the given integer handle	in this function using malloc instead of calloc helped to lower the time taken to execute my code calloc() zero-initializes the buffer while malloc() leaves the memory uninitialized. Zeroing out the memory may take a little time when executing my code also other steps I used in this function were using lock_acquire() and lock_release() lock_acquire() Allows the process to lock say for example files in a stream. Locking the file prevents other process from delivering change sets that modify it lock_release() Releases locks for example on a file in a stream that was locked by this process.
static struct file_descriptor *lookup_file(int handle)	this function returns the file descriptor associated with the given handle. file descriptor 0 is reserved for standard input and file descriptor 1 is reserved for standard output	
static int sys_filesize (int handle)	this function returns the size of a file	the steps I used in this function were using lock_acquire() and lock_release()

		lock_acquire() Allows the process to lock say for example files in a stream. Locking the file prevents other process from delivering change sets that modify it lock_release() Releases locks for example on a file in a stream that was locked by this process
static int sys_read (int handle, void *udst_, unsigned size)	this function reads a number of bytes from an open file into a buffer in the user program	the steps I used in this function were using lock_acquire() and lock_release() lock_acquire() Allows the process to lock say for example files in a stream. Locking the file prevents other process from delivering change sets that modify it lock_release() Releases locks for example on a file in a stream that was locked by this process
static int sys_write (int handle, void *usrc_, unsigned size)	this function writes a number of bytes to an open file from a buffer in the user program	the steps I used in this function were using lock_acquire() and lock_release() lock_acquire() Allows the process to lock say for example files in a stream. Locking the file prevents other process from delivering change sets that modify it lock_release() Releases locks for example on a file in a stream that was locked by this process
static int sys_seek (int handle, unsigned position)	this function changes the next byte to be read or written in an open file	the steps I used in this function were using lock_acquire() and lock_release() lock_acquire() Allows the process to lock say for example files in a stream. Locking the file prevents other process from delivering change sets that modify it lock_release() Releases locks for example on a file in a stream that was locked by this process

static int sys_tell (int handle)	this function returns the position of the next byte to be read or written in an open file	the steps I used in this function were using lock_acquire() and lock_release() lock_acquire() Allows the process to lock say for example files in a stream. Locking the file prevents other process from delivering change sets that modify it lock_release() Releases locks for example on a file in a stream that was locked by this process
static int sys_close (int handle)	this function closes a file	the steps I used in this function were using lock_acquire() and lock_release() lock_acquire() Allows the process to lock say for example files in a stream. Locking the file prevents other process from delivering change sets that modify it lock_release() Releases locks for example on a file in a stream that was locked by this process

Other information

while working on this project I made many design decisions. I will proceed by explaining my design. one of the design decision I made was to use semaphores. I used semaphores because they are a useful tool in the prevention of race conditions.

when I was implementing the system calls I made sure the file system code was treated as a critical section. this was important because it was essential only one process at a time had access to the file system code so that they didn't interrupt each other. to do this I used a lock_acquire to allow a process to lock the file system code preventing other processes from entering the critical section and making changes to the file. to avoid starvation I used lock_release so that the process would release the lock and give other processes a turn in the critical section.