NWEN 301 Project 3 Report
Kandice McLean 300138073

Data Structures

No changes were made to any data structures in the projects.

Algorithms

process.c

tid_t process_execute (const char *file_name)
Memory was allocated to a char pointer file_copy using the pre-defined function palloc_get_page. file_copy is used to make a copy of the passed in file_name, so changes can be made without affecting the arguments in the file name, which are needed for other functions. file_name is copied to file_copy using strlcpy, which is a predefined function in string.c in the kernel library, and then the command line arguments are removed by using strtok_r, which is another predefined function in string.c, splitting at the first space so that only the name of the file remains. This is then passed into thread_create, ensuring the threads file_name is only the name, and the original file_name string is not affected in any way, so can be used for its arguments later. thread_create returns a thread id, which is checked for error, then a semaphore placed on it while it is loaded to prevent interactions while it is loading. If the load is successful, the thread is placed on the list of the current threads children, and the tid is returned to whichever function called process_execute.

static void start_process (void *exec_)
If the load initiated in the function is successful, the wait_status must be properly initialised. memory is allocated to the current thread to store wait_status, and the wait_status assigned to the exec_info exec. This was all done on one line to save writing extra code. The lock variable in wait_status was initialised using lock_init from synch.c. The ref_cnt is initialised to 2, as at this point in the process both the parent and child processes are alive. The tid is allocated the current threads id, and the dead semaphore in the wait_status is iniated using sema_init from synch.c . Once this is completed execs success is set to true, and execs load_done semaphore is up'd using sema_up to prevent the parent waiting for the child to finish.

static void release_child (struct wait_status *cs)
A lock is acquired for the child wait_status passed in, as its ref count needs to be changed and it is important no processes access this one while this is occurring. ref_cnt is decremented, then the lock is released. If decrementing the ref_cnt causes it to equal 0, this means both child and parent and child are dead and so the childs wait status memory is freed up for other processes to use.

int process_wait (tid_t child_tid)
The current threads list of children is retrieved and then iterated through using a while loop that completes once the end of the list is reached. If the current child process has a matching tid to the one passed into the function, it is the one that needs to be waited for. Waiting is down using sema_down, the exit code is retrieved from the child processes wait status, the child is removed form the list as it is now dead, release_child is called, and its exit code is returned. If a child matching the tid cannot be found, the function returns error code -1.

void process_exit (void)
This is called when a process exits, so any of its resources need to be freed for use by other resources. sema_up is called on the wait_status's semaphore to stop other processes from waiting. release_child is called on the current threads wait_status to change the ref_cnt. If there are children remaining on exit, their wait_status's are removed from the exiting threads children's list one at a time, and they are all released using release_child.

static void * push (uint8_t *kpage, size_t *ofs, const void *buf, size_t size)
push pushes size bytes in the buffer onto the stack in the kernel page. The pointer in the kernel page is set to ofs when this function is called. First, the size is rounded up to the nearest multiple of 32 bits, which is the size of a word, and word aligned processes are faster to access than unaligned ones, so this step increases efficiency of the operating system. Note that every variable used in push is treated as a number of bytes, to ensure it is placed in the correct part of the kernel memory. If the ofs, which stands for offset, is smaller than the padded_size, the push cannot be completed correctly, so NULL is returned. If it is not, padded_size is deducted from the offset. memcpy, which is a predefined function in string .c which copies n number of bytes from source to destination is called. The number of bytes to copy is the size variable, the destination is the kernel pages pointer address*offset+(padded_size-size), and the source is the buffer. A pointer is then returned to the newly pushed item in memory.

static bool init_cmd_line (uint8_t *kpage, uint8_t *upage, const char *cmd_line, void **esp)
This is where the arguments in the command line are arranged in the correct manner in the kernel page stack so that they can be accessed by applications. The first thing that needs to be done is to push the words. Order does not matter, as their addresses will be pushed afterwards in the correct order and can be used to find them. As a sentence is really just multiple char arrays, the whole line is pushed onto the stack using the push function, passing in the kernel page pointer, the offset, which is set to the size of the kernel page to begin, the cmd_line, and the number of bytes to push, which is the size of the cmd_line. A constant char set to NULL, which will ensure argv[argc] is a null pointer, as per C standards. After this, the arguments are counted, and each one pushed into the stack separately, this is done by using strtok_r splitting at " " and assigning each arguments address to a pointer user_arg, which is retrieved by adding the user page pointer to the pointer attained from strtok_r minus the kpage pointer. This is then pushed to the stack in the same order as the cmd line was. This is the wrong order for argv, so a reverse function is called, which reverses the order of the pointers in argv after they are assigned to argv by giving it the current user pointer and adding the offset. After the addresses are reversed into the correct order on the stack, argv is pushed, then argc, and finally, another null address to complete the stack frame properly. This last null is a fake address that never gets called. If this is all successfully executed, true is returned.

static void reverse(int argc, charargv)**
This is a simple reverse function to reverse the order of the data in argv.

setup_stack (const char *cmd_line, void **esp)
All that needed to be done here was call the function init_cmd_line to set up the stack, and assign the boolean it returns to the success variable and return it. The bulk of the work is done in init_cmd_line.

syscall.c

static int sys_exit (int exit_code)
The exit_code is changed in the wait status of the current thread. thread_exit is called, then 0 is returned to indicate the exit is complete.

static int sys_exec (const char *ufile)
If the ufile passed in is NULL, there is no file to execute, so thread_exit is called so no resources are wasted. process_execute function accesses files, so needs to be treated as a critical section, so a lock is acquired. A copy is made of the file, then passed into process_execute function, which completes the process of executing, and returns a process id. Then palloc_free page is called to free the page used by the kernel_file copy. When all this is completed, the lock is removed for other processes to use, and the process id is returned.

static int sys_wait (tid_t child)
All that needs to be done here is to call process_wait and return the int provided by it.

static int sys_create (const char *ufile, unsigned initial_size)
The lock is acquired as there is a need to access the file system in this function, and the file system does not support concurrent access, so this is a critical section. After the lock is placed the ufile is copied over to a kernel file variable, then passed into filesys_create to create a file system with the kernel files name and initial_size size, and this returns an int advising if it was successful. The kernel file memory is then freed, and the lock rleased, and the value returned by filesys is returned.

static int sys_remove (const char *ufile)
This is exactly the same as sys_create, except it calls filesys_remove instead of create. Because of access to the file system, this function is treated as a critical section.

static int sys_open (const char *ufile)
A lock is required here, as this function accesses the file system. A copy of the file is made, then a new file created using filesys_open and the file name copy is freed from memory. If the new file returns NULL, the lock is released and the function exits, returning error code -1. If the new file is successfully opened, a file descriptor is created, and the values initialised. file_descriptor file is assigned the new file, and the handle is retrieved from the current threads handle variable, which is incremented at the same time. The new file_descriptor is then added to the file descriptor list, fds, in the current thread. The lock is then released, and the handle of the new file_descriptor is returned.

struct file_descriptor * get_file_by_handle(int handle)
This function iterates through the list of file descriptors in the current thread and returns the one that matches the handle. If it cannot find the file descriptor with the specific handle, returns NULL.

static int sys_filesize (int handle)
The lock is acquired, as this function works with files. Calls get file by handle to retrieve the file, checking to make sure that neither the file or file descriptor are NULL. Gets the size of the file by calling file_length, releases the lock and returns the size.

static int sys_read (int handle, void *udst_, unsigned size)
Will use the file system, so lock is acquired. Finds the file desciptor using find_file_by_handle and checks that neither the file descriptor nor the file are NULL. If they are, the lock is released and the process exits. If they are not NULL, a while loop is used to continue reading until all bytes are read. Within the loop, the amount of space left in the current page is found by using PGSIZE minus the offset of the page in the destination. If the sz of bytes to read is more than the remaining space in the page, will read bytes up until the amount left in page, otherwise will read all the bytes using file_read, which returns the number of bytes read. This amount is deducted from size, and added to the user page pointer and total_read variables. If the page was not big enough the first time around, there will be remaining bytes to read, so will continue looping until all bytes are read. The lock is then released, and the total bytes read is returned.

static int sys_write (int handle, void *usrc_, unsigned size)
Lock is acquired, as acesses file sytem. Uses file write if the handle isn't STDOUT_FILENO, which returns a value which is added to the total bytes_written variable. The lock is released only when the loop is complete, and bytes written is returned.

static int sys_seek (int handle, unsigned position)
Gets the file descriptor, puts the lock on, then uses file_seek function. Once complete, the lock is released, and 0 returned to indicate was successful.

static int sys_tell (int handle)
Gets the file descriptor, puts the lock on, then uses file_tell function, which returns an int giving the place of the next byte to be read. Once the lock is released, the next byte value is returned.

static int sys_close (int handle)
Contains access to file systems so lock is placed. The file descriptor and the file are retrieved through the handle. file_close is called, then the file descriptor is removed from the list in the current thread. The memory the file descriptor was using is freed, the lock is released, and 0 is returned indicating that the close was successful.

What steps are taken to minimize the amount of time spent executing your kernel code?

Lists: when iterating through lists, all loops break off as soon as the desired element is found, so time is not spent going through the whole list needlessly.

NULL checks: Whenever a file descriptor, file name or file is retrieved from somewhere, they are checked if NULL before proceeding in the algorithms, to ensure time is not wasted on performing functions on broken files. This also makes the OS more stable.

Other Information

Process exit contains a few steps which are identical to release_child, except it is called on the current_thread wait_status. Instead of writing duplicate code, that function is called, even though it was created for releasing children, not the parent.

NULL checks mean extra lines of code however an operating system needs to be reliable, checking for errors before performing functions on files is an easy way to avoid unwanted behaviour.

Locks were placed everywhere the file system was accessed, as it is a critical section due to not being able to handle concurrent processes accessing it.

In process execute, I used a char pointer that was allocated a free page rather than the predefined array, as the pointers are more dynamic.