CSE 150 - Operating Systems
Documentation #2

Spring 2018 - Lab 04 - Group 1

Avery Berchek, Aleksandr Brodskiy, David Cabral, Christopher DeSoto, Adiam G-Egzabher, Nanditha Embar, Christian Vernikoff

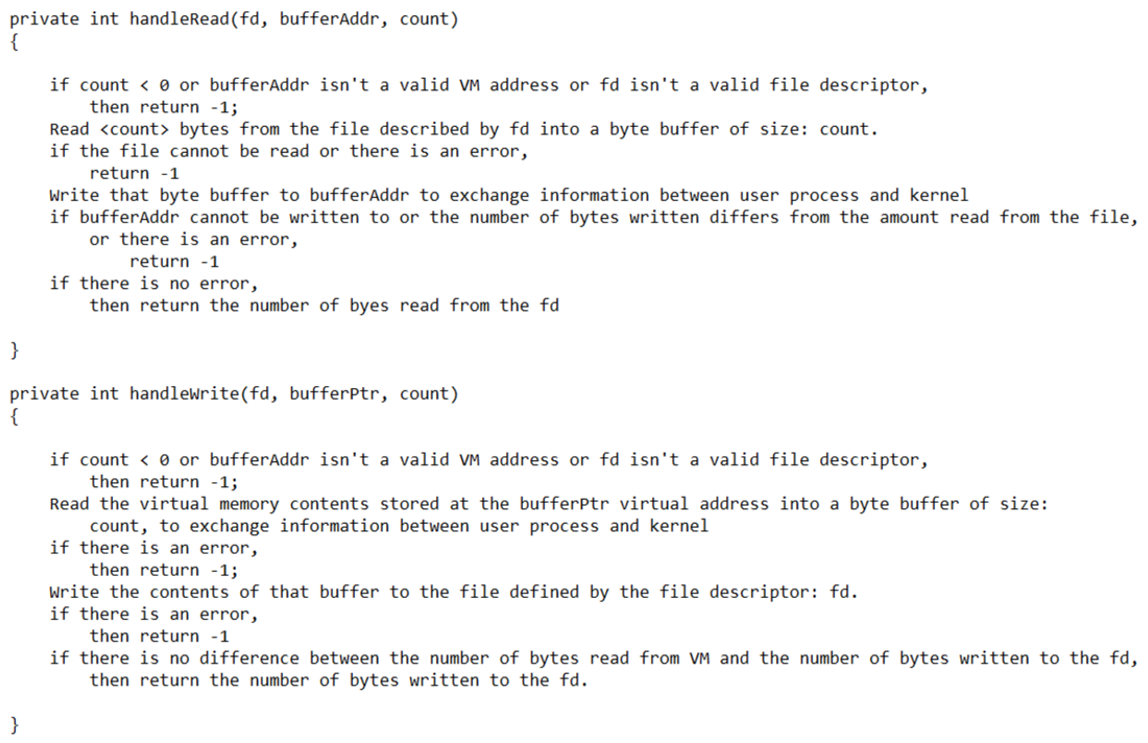
March 31, 2018

Outline

Documentation

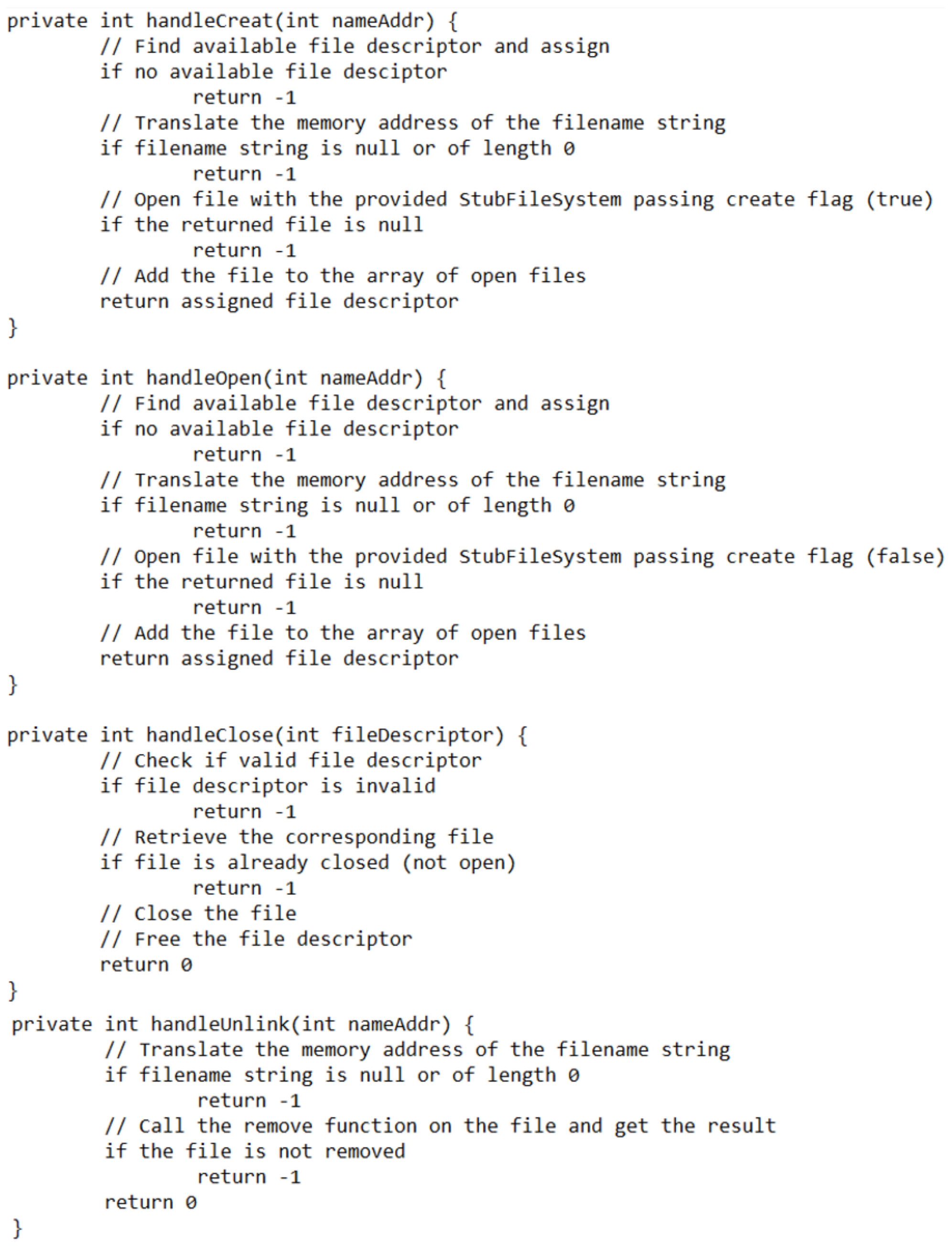
Task I

The pseudo−code amounted thus far for the implementation of a user processes with the ability to access a file system through various SYSCALLs is demonstrated below.



```
private int handleRead(fd, bufferAddr, count)
{

    if count < 0 or bufferAddr isn't a valid VM address or fd isn't a valid file descriptor,
        then return -1;
    Read <count> bytes from the file described by fd into a byte buffer of size: count.
    if the file cannot be read or there is an error,
        return -1
    Write that byte buffer to bufferAddr to exchange information between user process and kernel
    if bufferAddr cannot be written to or the number of bytes written differs from the amount read from the file,
        or there is an error,
            return -1
    if there is no error,
        then return the number of byes read from the fd

}

private int handleWrite(fd, bufferPtr, count)
{

    if count < 0 or bufferAddr isn't a valid VM address or fd isn't a valid file descriptor,
        then return -1;
    Read the virtual memory contents stored at the bufferPtr virtual address into a byte buffer of size:
        count, to exchange information between user process and kernel
    if there is an error,
        then return -1;
    Write the contents of that buffer to the file defined by the file descriptor: fd.
    if there is an error,
        then return -1
    if there is no difference between the number of bytes read from VM and the number of bytes written to the fd,
        then return the number of bytes written to the fd.

}
```

From the pseudo−code it is observable that the various SYSCALLs are implemented through the handler functions which perform *unlinkng*, *creating*, *opening*, *writing*, *reading*, & *closing* manipulations on the respective FILEs and their corresponding memory addresses.



```
private int handleCreat(int nameAddr) {
        // Find available file descriptor and assign
        if no available file desciptor
                return -1
        // Translate the memory address of the filename string
        if filename string is null or of length 0
                return -1
        // Open file with the provided StubFileSystem passing create flag (true)
        if the returned file is null
                return -1
        // Add the file to the array of open files
        return assigned file descriptor
}

private int handleOpen(int nameAddr) {
        // Find available file descriptor and assign
        if no available file descriptor
                return -1
        // Translate the memory address of the filename string
        if filename string is null or of length 0
                return -1
        // Open file with the provided StubFileSystem passing create flag (false)
        if the returned file is null
                return -1
        // Add the file to the array of open files
        return assigned file descriptor
}

private int handleClose(int fileDescriptor) {
        // Check if valid file descriptor
        if file descriptor is invalid
                return -1
        // Retrieve the corresponding file
        if file is already closed (not open)
                return -1
        // Close the file
        // Free the file descriptor
        return 0
}
private int handleUnlink(int nameAddr) {
        // Translate the memory address of the filename string
        if filename string is null or of length 0
                return -1
        // Call the remove function on the file and get the result
        if the file is not removed
                return -1
        return 0
}
```

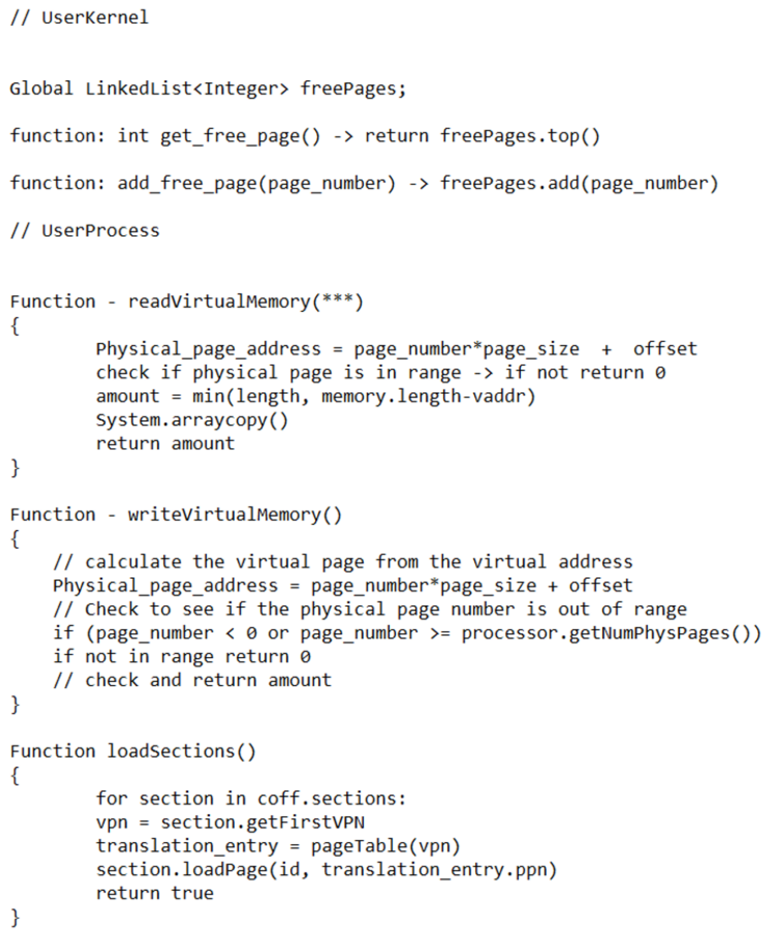
It is observable that the six functions defined are the handleRead, handleWrite, handleCreat, handleOpen, handleClose, & handleUnlink functions which enable user processes to manipulate Readable and Writeable files, respectively.

Within this manipulative capability are the underlining essentials of UNIX/POSIX functional system calls. As such, the control flow ensures that the provides with them the checks for validity of FILE *descriptors* and FILENAMES.

In this manner the functionality of the implemented pseudo−code is relatively straightforward and essentially self−explanatory.

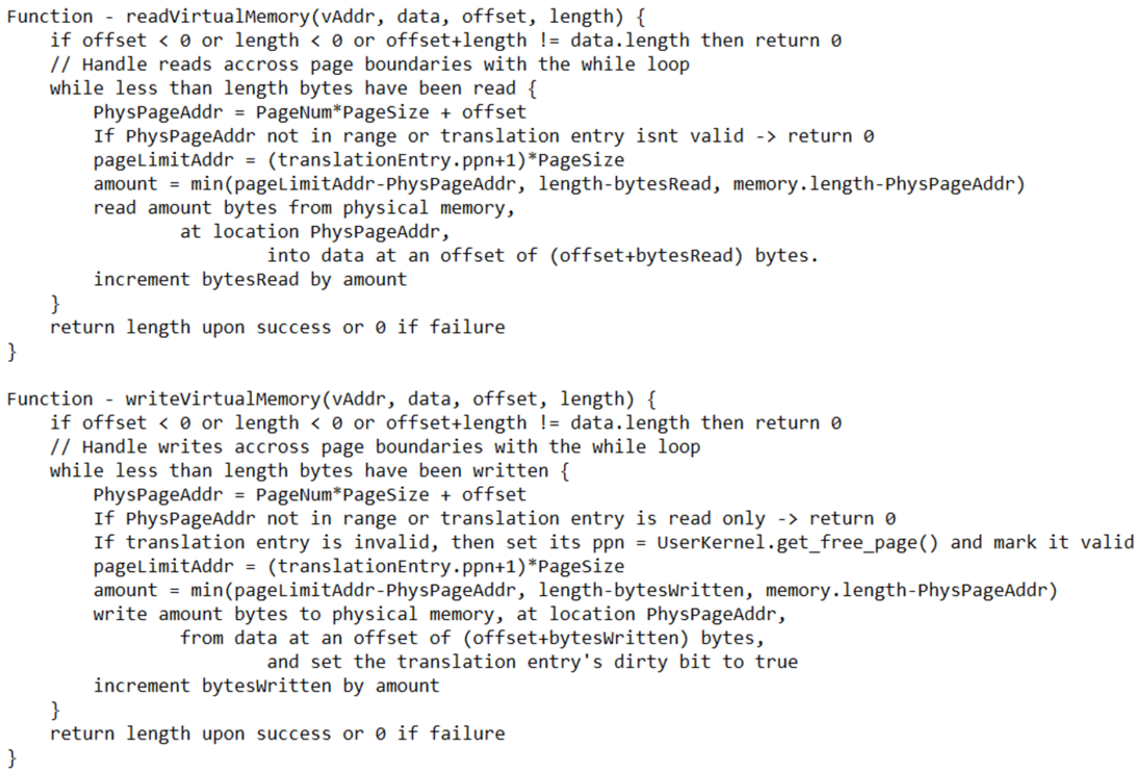
Task II

The initial implementation of user processes for the support of multi−programming paradigms is demonstrated below, through the following definitions of the LOAD-SECTION and READ/WRITE pseudo−code functions to the *pages* of *virtual memory*, which were eventually changed to accommodate for certain discrepancies which will be reviewed in greater detail in the **Desing Decision** section.



```
// UserKernel


Global LinkedList<Integer> freePages;

function: int get_free_page() -> return freePages.top()

function: add_free_page(page_number) -> freePages.add(page_number)

// UserProcess


Function - readVirtualMemory(***)
{
        Physical_page_address = page_number*page_size  +  offset
        check if physical page is in range -> if not return 0
        amount = min(length, memory.length-vaddr)
        System.arraycopy()
        return amount
}

Function - writeVirtualMemory()
{
    // calculate the virtual page from the virtual address
    Physical_page_address = page_number*page_size + offset
    // Check to see if the physical page number is out of range
    if (page_number < 0 or page_number >= processor.getNumPhysPages())
    if not in range return 0
    // check and return amount
}

Function loadSections()
{
        for section in coff.sections:
        vpn = section.getFirstVPN
        translation_entry = pageTable(vpn)
        section.loadPage(id, translation_entry.ppn)
        return true
}
```

This implementation entails the maintenance of a global linked list of physical *free*−pages for the allocation of memory by utilizing pages for new user processes. With the exclusion of the LOADSECTION function however, the reformatted *read* & *write* functions have been amended to ameliorate the handle reading and writing across page boundaries. The page boundary issue was mitigated with the utmost attention to potential error as the READ & WRITE VIRTUALMEMORY functions check that the Virtual address is indeed valid to avoid reading an erroneous and unnecessarily large buffer size. This was done by setting the physical page address equal to the offset added to the product of the page number and the page size.



```
Function - readVirtualMemory(vAddr, data, offset, length) {
    if offset < 0 or length < 0 or offset+length != data.length then return 0
    // Handle reads accross page boundaries with the while loop
    while less than length bytes have been read {
        PhysPageAddr = PageNum*PageSize + offset
        If PhysPageAddr not in range or translation entry isnt valid -> return 0
        pageLimitAddr = (translationEntry.ppn+1)*PageSize
        amount = min(pageLimitAddr-PhysPageAddr, length-bytesRead, memory.length-PhysPageAddr)
        read amount bytes from physical memory,
                at location PhysPageAddr,
                        into data at an offset of (offset+bytesRead) bytes.
        increment bytesRead by amount
    }
    return length upon success or 0 if failure
}

Function - writeVirtualMemory(vAddr, data, offset, length) {
    if offset < 0 or length < 0 or offset+length != data.length then return 0
    // Handle writes accross page boundaries with the while loop
    while less than length bytes have been written {
        PhysPageAddr = PageNum*PageSize + offset
        If PhysPageAddr not in range or translation entry is read only -> return 0
        If translation entry is invalid, then set its ppn = UserKernel.get_free_page() and mark it valid
        pageLimitAddr = (translationEntry.ppn+1)*PageSize
        amount = min(pageLimitAddr-PhysPageAddr, length-bytesWritten, memory.length-PhysPageAddr)
        write amount bytes to physical memory, at location PhysPageAddr,
                from data at an offset of (offset+bytesWritten) bytes,
                        and set the translation entry's dirty bit to true
        increment bytesWritten by amount
    }
    return length upon success or 0 if failure
}
```

Task III

The algorithmic functionality of the

USERPROCESS.JAVA

EXIT, EXEC, JOIN $\longrightarrow 1, 2, 3$

The pseudo−code implementations of the EXIT, EXEC, and JOIN system calls is demonstrated below:

```
    HashMap <Integer, UserProcess> children
    int pid, parentPID, statusOnExit
    Semaphore signaller = new Semaphore(0)
    int numProcs = nextPID = 0;

handleExit(status):
    Close any open files in the fd table
    // Remove each child processes' pointer to the this proc
    For each child in children:
        child.parentPID = -1
    unloadSections()
    statusOnExit = status
    UserKernel.numProcs--;
    if(UserKernel.numProcs == 0):
        Terminate the kernel
    //Signal to the parent that this process called exit
    signaller.V();
    Finish the main thread of this process

handleExec(fnamePtr, argc, argvPtrPtr):
    filename = read the string pointed to by fnamePtr. Upon error, return -1
    if argc < 0 or filename is null or filename doesnt end with .coff then return -1
    read argc integers from memory location argvPtrPtr,
        then read the strings pointed to by those integers to retrieve the argv contents to pass to the child process,
                upon error, return -1
    UserProcess child = newUserProcess()
    child.pid = UserKernel.newPID()
    child.parentPID = pid;
    children.put(chile.pid, child)
    set file descriptor entries 0 and 1 to the console's Input and Output
    make copies of the processes' file descriptors for the child
    If the child successfully executes the program defined by filename with parameters argv,
        then return child.pid,
                otherwise reverse the child setup and return -1

handleJoin(procID, statusPtr):
    // Verify this process is calling join on a pid it is the parent of
    if(!children.contains(procID)) -> return -1
    child = children.get(procID)
    child.signaller.P() // Sleep while waiting for the child process to exit
    status = child.statusOnExit
    write status to the virtual memory address defined by statusPtr, upon error, return -1
    children.remove(procID)
    if status indicates child exited abnormally, then return 0, otherwise,
    return 1
```

Where the data structures of User Process and User Kernel are HashMap<Integer, UserProcess> *children*, int pid, parentPID, statusOnExit, Semaphore *signaller* = *new* Semaphore(0) and int numProcs, int nextPID respectively.

The handleExit function introduces child process IDs, open file maintenance data structures, and status variables in conjunction with the de−allocation of the file descriptor objects. This is done in order to successfully an exit and close all left open files. In this manner the iteration of the child processes to succeed in the removal of parent pointer establishes the unloadSection dependence which provides the successful exiting of the processes.

Within the handleExec function are manipulations of the argv pointer array to pass arguments to the new executible. This ensures the that the exec handler would obtain the pointers to the argv **char** buffers and read from VIRTUAL MEMORY to achieve execution capabilities. The return is of type **int** which signifies either an error or under succesful circumstances returns a process ID.

.

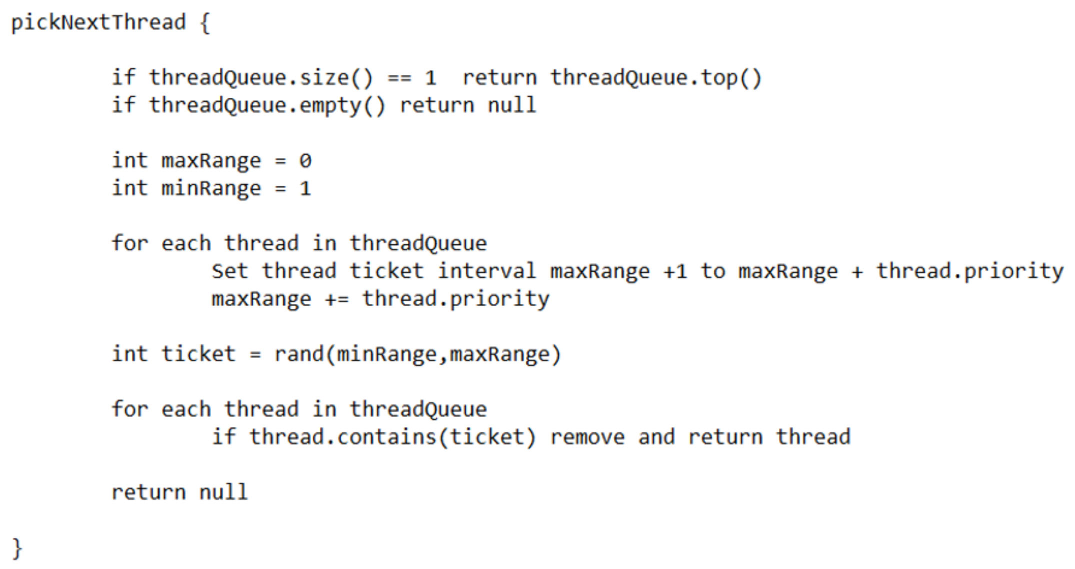
Task IV

Initially, the preliminary idea behind developing a sound and complete implementation of a lottery−based scheduler was to change the maximum and minimum priority value*d* variables. As such, the lottery scheduler can incorporate its algorithmic functionality from the priority queue, respectively. In this manner the lottery scheduler would basically be a priority queue with the maximum value assigned as the highest INTEGER value available in JAVA and the minimum value being initialized to one representing every thread that receives at least one ticket, therefore the priority will be changed to being reliant on tickets.

From this description it can be gathered that in order to select the thread that *wins* the lottery, first it would go through all threads in the queue and add up all the tickets to find total number of tickets in queue (this may be inefficient however, because this would have to be done every time the NEXTTHREAD function is called). Once this number is found, a random number generator will be used to simulate the lottery with an interval of possibilities from the aforementioned number to the sum of the queues tickets. Once this random number is found it would perform iterative computations through the thread queue and distinguish which thread contains that ticket. However To be able to have unique tickets, support threads containing a large amount of tickets, and not keep track of the status of, as a plausible assumption might be, a plethora of tickets (which must consequently be allocated a lot of memory) a methodology of creating unique intervals while finding the sum of queue tickets must be used.

This process is best explained through exemplification. For instance, suppose two threads are in a queue and they each have five tickets; the random number generator would iterate between $1 \longrightarrow 10$. Assuming the number 7 is produced from the random number generator, each thread would be assigned an interval and the process would begin from top of the queue to the end. Ergo, the top thread would be assigned the interval 1⟶5 to represent it's 5 tickets. In this manner the other thread would be assigned an interval of 6⟶10. Therefore, since the second thread has the random number within it's interval, it is the thread that gets selected and removed from the queue.

The picknextThread described below is the new function that would be defined in order for the priority queue to override it in the lottery scheduler and not the nextThread function as defined in the priority scheduler.



```
pickNextThread {

        if threadQueue.size() == 1  return threadQueue.top()
        if threadQueue.empty() return null

        int maxRange = 0
        int minRange = 1

        for each thread in threadQueue
                Set thread ticket interval maxRange +1 to maxRange + thread.priority
                maxRange += thread.priority

        int ticket = rand(minRange,maxRange)

        for each thread in threadQueue
                if thread.contains(ticket) remove and return thread

        return null

}
```

Design Decisions

The primary factors of significant importance to the design considerations for the development of a multi−programming file system were the scalability, robustness, and efficiency of the system in accordance with the resources associated with its support. In this manner the utilization of data structures, primitive types included, such as priority queues was preeminently regnant to the functionality of the objects defined. For example classes such as ticketInterval were omitted from utilized for the LotteryScheduler class in order to hold a small lottery between two threads and then allow the thread that wins that lottery to go on to have a lottery with the next thread in the queue. Although the ticketInterval class was provided its omission enabled the more efficient functionality of the LotteryScheduler algorithm.

Another issue of equal importance is the fact that the virtual memory functionalities of **Task 2** were changed from the initially proposed implementation in order to account for a heap error that became eminent during the execution of the implementation against established corner cases. In this manner the virtual memory was detected to act sporadically and began to allocated unprecedented amounts of memory. As such the response to alter the READ & WRITE functions was an integral decision of the design process. Therefore, READVIRTUALMEMORY was provided checks to not only ensure that the virtual address is valid, but if it isn't to attempt to read one byte at a time from the address in question and assume that it is an invalid address otherwise.

Essentially these design decisions ameliorated the **Testing/QA** phase of the development endeavor.

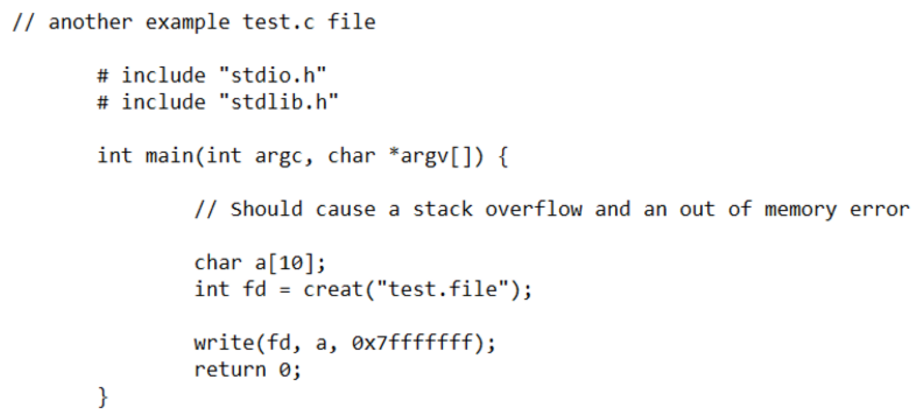
Testing/QA

Task I

In order to establish a testing strategy on the validity of the implementation for the READ & WRITE functions of the first six functions introduced in **Task 1**, a stress test was performed in which a large chunk of memory was dedicated to READ & WRITE by creating an array that maintains a large number of integers. In this manner a for loop was then introduced to fill up the array with values and then another for loop to iterate through the array and assert that the values are correctly maintained in order to test that the corresponding sections and pages are functioning correctly.

Task II

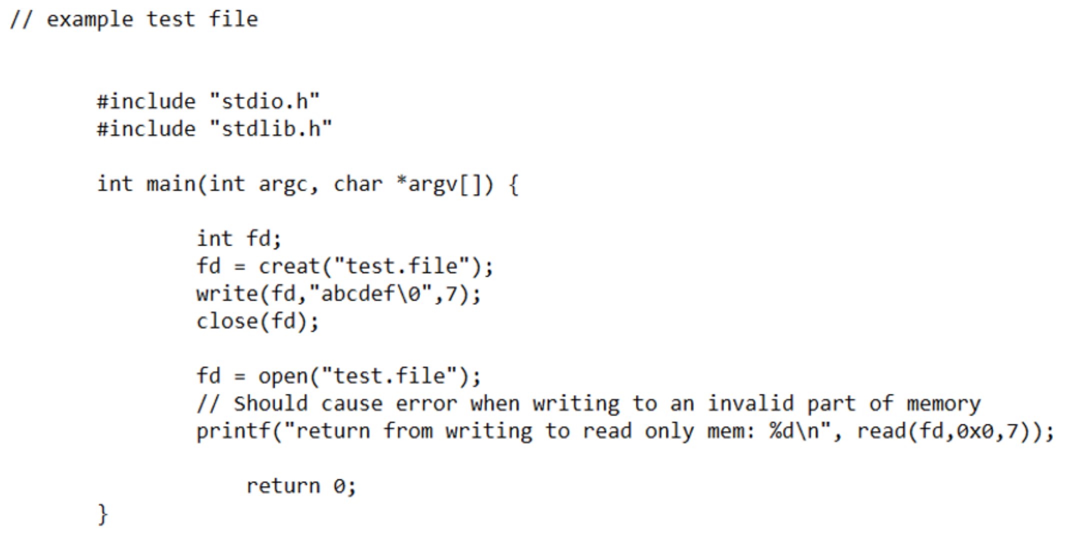
In order to induce an *out of memory* error the following test file was used.



```
// another example test.c file

    # include "stdio.h"
    # include "stdlib.h"

    int main(int argc, char *argv[]) {

        // Should cause a stack overflow and an out of memory error

        char a[10];
        int fd = creat("test.file");

        write(fd, a, 0x7fffffff);
        return 0;
    }
```

Through the execution of this source file a *stack overflow* error was generated and the responsiveness of the code was tested against the induced *out of memory* exceptions in order to solidify its functionality and develop a more efficient virtual memory.

Another source file, exemplified below, was constructed in order to invoke of an invalid chuck of memory during the writing process.

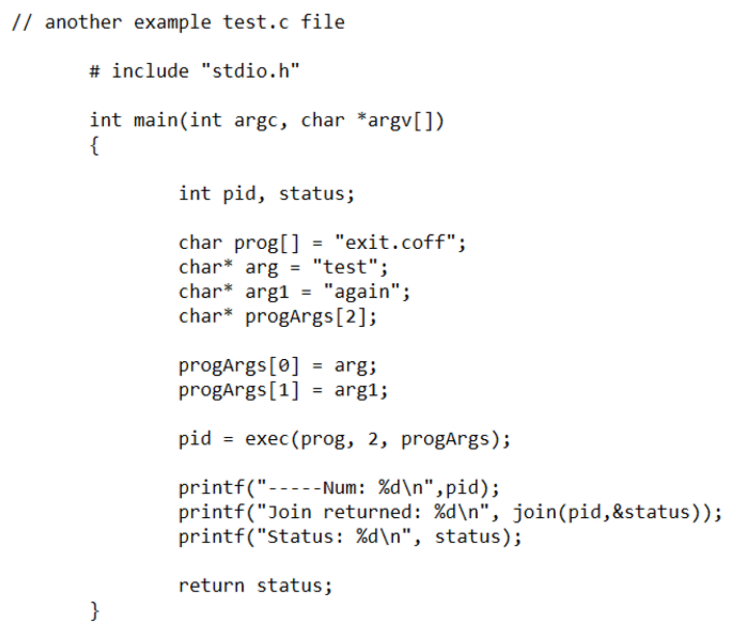


```
// example test file


    #include "stdio.h"
    #include "stdlib.h"

    int main(int argc, char *argv[]) {

        int fd;
        fd = creat("test.file");
        write(fd,"abcdef\0",7);
        close(fd);

        fd = open("test.file");
        // Should cause error when writing to an invalid part of memory
        printf("return from writing to read only mem: %d\n", read(fd,0x0,7));

            return 0;
    }
```

Within this exemplification, it becomes apparent that there is a copious amount of these lilliputian edge cases/errors. In this manner it becomes paramount to state for the record that this is merely an algorithmic representative of many different test cases used to run test cases against the aforementioned implementations. Therefore the test of accessing invalid segments of memory with the intent of writing to read only has passed and served as an integral part of the testing/QA phase of this project.

Task III

In order to test for the completeness and correctness of the implementation of the EXEC, JOIN, and EXIT syscalls was to run a process demarcated with an identifier and a variable for accounting the process status as follows:



```
// another example test.c file

        # include "stdio.h"

        int main(int argc, char *argv[])
        {

                int pid, status;

                char prog[] = "exit.coff";
                char* arg = "test";
                char* arg1 = "again";
                char* progArgs[2];

                progArgs[0] = arg;
                progArgs[1] = arg1;

                pid = exec(prog, 2, progArgs);

                printf("-----Num: %d\n",pid);
                printf("Join returned: %d\n", join(pid,&status));
                printf("Status: %d\n", status);

                return status;
        }
```

In this manner, the EXEC syscall was tested and made ***bullet–proof*** through the running of the USERSYSCALLTEST.C. Therefore, the JOIN and EXIT were testing, debugged, and adjusted in a likewise manner.

Task IV

Two main testing strategies were utilized to check for the completeness and correctness of the LotteryScheduler implementation. Firstly, testing on multilevel queues and situations where there were threads with a very large number of tickets was the preliminary method. Testing for multilevel queues was done to make certain that priority donation worked correctly with the addition of tickets to signal priority donation. Testing for tickets with a high amount of tickets was done to make sure that a new system could calculate priorities for threads without needing to compute the sum of all tickets in thread queue.

Team Member Work-Log

Avery Berchek
DevOps Engineer

- Designed and outlined the pseudo−code for Task III
- Outlined and participated in the design process associated with Tasks I, II, III, IV
- Implemented the algorithmic functionality for Task III
- Primarily responsible for implementation of corner−case test cases
- Conducted and performed code reviews
- Performed optimizations and ensured the efficiency of the data structures utilized in the source code

Aleksandr Brodskiy
Project Manager

- Organized the *sprints* and weekly meetings in accordance with the *Agile/Scrum* project management methodology.
- Delegated tasks and assigned roles for the Engineering Team as well as conducted the code reviews
- Participated in the design process associated with Tasks I, II, III, IV
- Created and formatted the Design Documentation
- Managed all progress and operations of the Engineering Team in order to provide an efficient, robust, and optimal solution for a timely and submission.

David Cabral
Design Engineer

- Designed and outlined the pseudo−code for Task IV

- Implemented the algorithmic functionality for Tasks IV
- Provided insight and proposed solution for *Lottery Scheduler*
- Proposed solution and outlined source code structure for Task IV

Christopher DeSoto
Principal Engineer

- Set−up the integrated development environment for the entire Engineering Team.
- Outlined and participated in the design process associated with Task I
- Implemented the algorithmic functionality for Task I
- Integrated source code from the *development phase* to the *testing phase* to the *completed phase* to ensure the validity of the solution and provide a general framework for the debugging and testing process for the QA and Software Engineers.

Adiam G-Egzabher
Systems Engineer

- Outlined and participated in the pseudo−code development associated with Task I
- Participated in the design process and Testing/QA phase for Task I
- Provided insight and proposed solution for Task I

Nanditha Embar
QA Engineer

- Ensured functionality of all features
- Provided insight and proposed solution for Task II
- Ensured the highest quality attainable with the time and resources provided for submitted source code

- Participated in the design process and testing/QA phase for Task II

Christian Vernikoff
Software Engineer

- Designed and outlined the pseudo−code for Task II
- Implemented the algorithmic functionality for Tasks II
- Outlined and participated in the design process associated with Task II

Conclusion

As a result of completing an efficient and optimized set of solutions to the aforementioned tasks, foundational components of a POSIX operating system, a multi−programming systemized thread structure, was formed. The developed foundational components of this structure were the various syscalls along with the virtual memory functionality as well as the lottery scheduler algorithm. In this manner, the development of these foundational components supports functionality for higher level client/server applications and protocols.