# CSE 314
# Report of Assignment 3 on Nachos

**Student ID: 1605005**

**1605006**

**1605012**

# Part 1: Threading

## Task1:

**Implementation**:

This task is about implementing a join call in class nachos.threads.KThread so that one thread can wait until the completion of another thread. We have made some changes in join() and finish() methods of the KThread class. We have firstly initialized a new variable named joinThread to null which will store the thread which called join first on this thread. In the **join() method**, we have checked whether the status of this thread is statusFinished or not. If this thread was finished, we returned from the function immediately. Otherwise, we have checked if the joinThread variable is null because the result of calling join() a second time on the same thread is undefined. If there was no join() called on this thread before, then we store the currentThread into the joinThread variable and block the currentThread by sleep(). Another thing to be noted is that before sleep(), we have disabled the interrupt and after sleep(), we have restored the interrupt . At last, in **finish() method**, we have checked if the currentThread.joinThread is null or not, if not null, then we make the joinThread ready again so that it can finally wake up.

**Testing:**

We tested our implementation in the JoinTest class, where we fork a PingTest runnable from the main thread and wait for it to finish by calling join() on it. After the PingTest has finished, we print a line from the main thread. The fact that this line only gets printed after all the prints in PingTest confirms that join() is working.

# Task2:

### Implementation:

This task is about implementing condition variables directly without using any semaphores in class nachos.threads.Condition2. For this task, we have used a LinkedList of KThread named waitQueue. In addition to this, we have used a Lock variable named conditionLock for synchronization. We have made some changes in sleep(),wake() and wakeAll() methods of Condition2 class. To provide atomicity, interrupt has been disabled before doing any change in function and enabled at the moment of departure. In **sleep() method**, conditionLock was released which was acquired by currentThread before . Then we have added this currentThread to the list of waitQueue and currentThread gets blocked by KThread.sleep(). Finally, whenever this thread is woken up from sleep state, conditionLock has been reacquired by that thread again and returns. In **wake() method**, we have checked whether the waitQueue is empty or not and if this queue is not empty, then the first thread of this queue gets woken up by the ready() call which slept on this condition variable before. At last, in **wakeAll() method**, until the waitQueue is not empty, we have called wake() to wake all the sleeping threads.

### Testing:

Condition2 was checked in conjunction with the Communicator class. The testing of the communicator class has been explained in the 'Task4' section.

# Task3:

### Implementation:

This task is about completing the implementation of nachos.threads.Alarm class.For this task, we have made changes in waitUntil(long x) and timerInterrupt() methods in the Alarm class. Here, we have used a priorityQueue of a custom class named Entry. Entry stores the waiting threads and their wake time. We have implemented compareTo(Entry other) method in Entry class so that in priorityQueue, all the elements are sorted in ascending order of wakeTime. In the **waitUntil(long x) method**, we have calculated the wakeTime by adding Machine.Timer.getTime() to x. Then a new object of Entry class has been created with this currentThread and wakeTime and also we have pushed this object to the priorityQueue. Finally, currentThread gets blocked by KThread.sleep() and it will be woken up by timerInterrupt() method after passing its wakeTime. In **timerInterrupt() method**, we have checked if the current time has passed the wakeTime of the first entry of the waitingQueue, if so we removed it from the queue, and called ready() on it to wake it up again. We keep doing this check till we found a entry whose wakeTime has still not been passed or the queue has become empty.

### Testing:

We have tested this in the AlarmTest class. Here, we have forked 3 different Kthreads, and called waitUntil on all 3 of those for a fixed time. We printed the time before calling waitUntil, and also printed the time after they woke up. By checking the difference in sleep and wake time for all of them, we are able to confirm that the Alarm class is now working.

# Task4:

### Implementation:

This task is about implementing class nachos.threads.Communicator with operations, void speak(int word) and int listen() so that a speaker sends and a listener receives msg in a synchronized way. For this task, we have used five Condition2 variables, one lock and two boolean variables. Firstly, **in speak(int word) method**, lock is acquired for mutual exclusion. In speak(), a speaker first checks if isSpeaking is true, if so, it means that already a speaker is active, so it sleeps on the transferred condition variable. If not, it sets isSpeaking to true, to prevent further speaker from interrupting. It then calls wake on speakerReady variable which would wake a waiting listener if available. It then sleeps on listenerReady, waiting for a listener to be active. When a listener wakes the speaker, it transfers the word, and calls wake() on the spoken() variable, to let the corresponding listener know that it has finished writing. It then waits on the listened variable. Finally, when it gets waken by the listener, is sets isSpeaking and isListening to false, and calls wake on transferred, to allow a new speaker to be active.

On the other hand, **in listen() method**, a listener checks if isListening is false, if true it means another listener is active, so it sleeps on the speakerReady variable. It also checks if isSpeaking is true, if not, it means no speaker is currently active, in that case too, it waits on the speakerReady variable. If it passes these two checks, it sets isListening to true and calls wake on listenerReady variable, to wake the currently waiting speaker. It then sleeps on the spoke variable and waits for the speaker to speak. After it gets woken up, it stores the spoken int and calls wake on the listened variable to wake the corresponding speaker up. It then returns the received word.

### Testing:

We tested both the Communicator and Condition2 in the CommunicatorTest class. We forked 3 different speakers and 2 different listeners here. Each speaker speaks from 0 to 5. The fact that a speaker begins speaking only after a listener is available and only one speaker can speak at a time and only one listener can hear what a speaker just said indicates that the Communicator and Condition2 are working correctly.

# Part 2: Multiprogramming

## Task1:

**Implementation:**

This task is about implementing read and write system calls in nachos.userprog.UserProcess class so that the user can read the inputs from console and write something in console. For this task, we have added two functions named handleRead(int fileDescriptor, int vaddr, int size) and handleWrite(int fileDescriptor, int vaddr, int size). Here, we have used a Lock variable named ioLock for atomicity of read and write operation. At the start, when UserProcess is created, it initializes its file descriptors by using openForReading() and openForWriting() of the console of the kernel.

In **handleRead(int fileDescriptor, int vaddr, int size) method**, ioLock is acquired and then we have checked if fileDescriptor is not stdin or vaddr is less than zero or size is less than zero and if any condition is true, we return -1 as error. Otherwise, we have declared a buffer array so that number of bytes copied from console can be stored in that array.Then readsize is returned from openFile.read(buffer,0,size) method and if readsize is greater than zero we have written the bytes of  buffer array into virtual memory by calling writeVirtualMemory(vaddr,buffer,0,readsize). Now we have checked if writeSize is equal to readSize and if they are equal then we return writeSize, otherwise we return -1 as error. Finally, ioLock is released by the process.

In **handleWrite(int fileDescriptor, int vaddr, int size) method**, ioLock is acquired and then we have checked if fileDescriptor is not stdout or vaddr is less than zero or size is less than zero and if any condition is true, we return -1 as error. Otherwise, we have declared a buffer array so that number of bytes copied from virtual memory can be stored in that buffer. Then readSize is returned from readVirtualMemory(vaddr,buffer,0,readsize) and if readsize is not equal to size, we return -1 as error. Otherwise, writeSize is returned from openFile.write(buffer,0,size) method which defines the number of bytes written on console. At last, if readSize and writeSize is not equal we return -1 as error and otherwise we return writeSize. Finally, IoLock is released by the process at the departure.

**Testing:**

In our shell code, all sorts of printf are working, which indicates that writing to console is being done properly. So, write system call is working. Also, in the forked echo.coff process, we take an integer, and later a string as input, both of which are getting delivered properly. So, the read system call is also working.

# Task2:

**Implementation:**

This task is about implementing support for multiprogramming so that multiple user processes can run at a time. For this task, we have made changes in nachos.userprog.UserProcess class and nachos.userprog.UserKernel class.

In **UserKernel class**, we have used a global LinkedList of free physical pages and a pageLock for atomicity of operation of the list. In **initialize(string[] args) method**, we have initialized the LinkedList by adding all the free pages which is equal to total physical pages(64).We have also shuffled the list for testing so that all pages of a process always cannot be allocated in contiguous block. Then, we have added two functions in UserKernel class named getNewPage() and freePage(ppn). In **getNewPage()**, we return a free physical page number from the LinkedList and remove it from this list. If there is no free page in the list, then we return -1 that means insufficient memory. In **freePage(ppn)**, we have added the page back to the LinkedList again which means that this page is free for allocation of new process.

In **UserProcess class**, we have made changes in four functions. We have initialized processID as the number of processes and a pageTable array of TranslationEntry in the constructor of this class. A lock variable is used for the update of processID. In **loadSections() method**, we allocate the memory for this process and load the COFF sections into the memory. Every section has multiple pages.So we have used two for loops, one is iterated through number of sections and another loop is iterated through section.getLength().

Virtual page number is retrieved from each section and a new physical page number is returned from UserKernel.getNewPage(). If there is no free physical page, we return false as insufficient free memory and free the already allocated pages. Otherwise, we have added a new TranslationEntry to the pageTable with vpn,free ppn, validBit as true, dirtyBit as false, usedBit as false and readOnlyBit as section.isReadOnly().Then, we have loaded the physical page into memory by calling section.loadPage(sectionPageNumber,physicalPageNumber).In addition to this, stackPages and argumentPage are also added to the PageTable as TranslationEntry.If all the pages of the process are successfully added in the pageTable, we return true which means this process will definitely be run. In **unloadSections() method**, we have just freed the physical pages which were used for this process by calling UserKernel.freePage() and also closed stdin and stdout fileDescriptors for this process.

In **readVirtualMemory(int vaddr,byte[] data,int offset,int length) method**, we have checked if vaddr is less than zero or greater than length of main memory and in that case we return 0.Then,maxAmount is the number of maximum bytes which can be read from the virtual memory. We have used a for loop which iterates through maxAmount and in each iteration, we translate the virtual address to physical address by using the page table. Finally, bytes of memory on this physical address have been stored in the data buffer array by incrementing the amount of bytes transferred and make usedBit of that entry as true. At the end of the method, we return the total number of bytes successfully transferred.

In **writeVirtualMemory(int vaddr,byte[] data,int offset,int length) method**, like readVirtualMemory() function we have added same functionalities in this method. The difference is just when we get physical address by translation, then bytes of data buffer array have been transferred to the memory indexed on that physical address by making usedBit and dirtyBit of that entry as true. At the end, we return the total number of of bytes successfully transferred to the memory. Before writing to an address, we also first check if that address belongs to a read only page, if so, we return without writing any further.

**Testing:**

As the read and write system calls use the readVirtualMemory and writeVirtualMemory functions, and the fact that all the reads and writes in our shell program is still working, indicates that the virtual memory allocation system is working. Also, as we are able to create several processes in turns, it means that the exiting processes are also freeing their memory properly.

# Task3:

**Implementation:** This task is about implementing three system calls which are exec, join and exit in nachos.userprog.UserProcess class. For this task, we have added three functions in the class named handleExec(int nameVAddrress,int argc,int argVAddress), handleJoin(int pid,int statusVAddress) and handleExit(int status).
We have added some new data structures such as a List of UserProcess named childProcesses, parentProcessID (int), activeProcessCount (int), exitStatus (int) and didExitNormally (boolean) for implementation of these system calls.

In **handleExec(int nameVAddress,int argc,int argVAddress) method**, if VAddress, argc or argvVAddress is less than zero, we return -1. Otherwise, we have read file name from virtual memory by calling readVirtualMemorystring(nameVAddress,maxFileSize). If this file name does not end with ".coff", we return -1.Otherwise, we have made a argV array of String which contains all the arguments passed to the new process. After that, new userProcess has been initialized and we have checked if new process can execute the coff file. And if there is no error in the execution of that coff file then child process set its parentprocessID as this processID and also we have added the new child process to the childProcessesList of current process. Finally, we return the processID of child process.

In **handleExit(int status) method**, all the children of this process disown its parent by setting parentProcessID as -1.Then, all the sections will be unloaded and descriptors will be closed for this process by calling unloadSections().Now, we decrement the activeProcessCount as this process is no longer be run. If this count is zero that means this is the last exiting process, then we call Kernel.kernel.terminate() directly. Otherwise, the status has been  stored in exitStatus variable which passed to parent process and user thread of this current process has been destroyed by calling UThread.finish().

In **handleJoin(int pid,int statusVAddress) method**, we have iterated through each child process and in each iteration we have checked if pid is equal to childProcessID and if that is true we call child.processThread.join() for joining on child process. Hence,currentProcess gets blocked then and waits until completion of that child process. Whenever parent process is ready to run, we have removed  this child process from the List so that parent process cannot join on the same child process again in future .Now, exitStatus of child process has been passed to parent process and this exitStatusbytes have been written into virtual memory starting at statusVAddress. At last, we have checked the value of didExitNormally. If it is false, it means that exiting of the process occurred due to an unhandled exception and so we return 0.So,in handleException(int cause) method, we call handleExit(-1) in the case of unexpected Exception. Otherwise, if the process exited normally, we return 1 as a successful join. If pid is not equal to any processId of any child, we return -1 as invalid join.

In **handleHalt() method**, we have added extra check whether the current processID is zero or not. Because only the root process can invoke the halt system call. So, if processID is not 0, we return -1. Otherwise, Machine.halt() is called.

## Testing:

In our shell code (myPgr.c), we have used exec to create several child processes. All of those processes are working. We have also called join on these processes, and the parent process waits for the child processes to end. And if we call join on processes with invalid ids, join returns -1. Which means that join is working. Also, the exitStatus of the children is being written properly, which has been verified by printing them after they join. We have also checked for exit of a process due to error by calling exec on “bad.coff”, in which case, the OS does not crash, and the parent process gets -1 as the exitStatus of that process. We have also tried halting from non-root processes, which fail to halt the machine, but the root can halt the machine successfully. So halt is also working correctly.