Nachos Report

1605031
1605041
1605044

Part - 1

Task - 1:

__Location of change in source code__
Changes were made in **'nachos/threads/KThread.java'** in the **join()** and **finish()** methods.

__Data structures used__
A **'ThreadQueue'** was used to maintain the queue of threads that have joined and are waiting on the current thread to exit.

__Basic implementation idea__
Every time a thread calls join() on a target thread (where the caller is not the target thread itself), if the target thread has not finished execution yet, the caller is added to the **ThreadQueue** named **waitingForThisThread**. This is used to maintain a list of all threads that have joined with the target thread. After adding the caller thread to the ThreadQueue, the caller thread is sent to sleep.

When this target thread finishes execution, before going to sleep permanently with a status of **statusFinished**, it keeps fetching threads from its ThreadQueue of threads that are waiting for it to exit and adds all of them to the ready ThreadQueue, so that they are once again queued to receive CPU.

__Tests__
The **'selfTest()'** method inside the **KThread.java** file tests the functionality of join() by first attempting to join a thread to itself and ensuring that the join has failed since a thread should not be able to join itself. After that it forks a new thread from the current thread and joins the current thread to that new thread.

Task - 2:

Location of change in source code
Changes were made by creating the file **‘nachos/threads/Condition2.java’**.

Data structures used
A **‘ArrayList<KThread>’** object was used to maintain the list of threads that are currently waiting on this particular condition variable.
A **‘Lock’** object was used to protect access to the aforementioned list of threads.

Basic implementation idea
Upon calling the **sleep()** method on a Condition2 object, the caller thread is added to the **ArrayList** named **threadArrayList** and then sent to sleep. This entire mechanism is protected by acquiring and releasing the aforementioned lock.

When a thread calls **wake()** on a Condition2 object, the first thread in the **ArrayList** named **threadArrayList** is woken up and put on the ready queue. This is also protected by the acquiring and releasing of the aforementioned lock.

When a thread calls **wakeAll()** on a Condition2 object, it calls **wake()** on all threads in the **ArrayList** named **threadArrayList**.

Tests
The **‘selfTest()’** method inside the **Condition2.java** file tests the functionality of the Condition2 class by running two tests.
Firstly, a single thread is sent to sleep on a condition variable and then a second thread wakes it up.
After that, multiple threads are consecutively sent to sleep on a condition variable and a different thread wakes all of them up.

Task - 3:

Location of change in source code
Changes were made in the file **‘nachos/threads/Alarm.java’** by updating the **timerInterrupt()** and **waitUntil(long x)** methods.

Data structures used
A **‘ArrayList<KThread>’** object named **sleepQueue** is used to maintain the list of threads that are currently sleeping for a fixed duration.
A **‘HashMap<String, Long>’** object named **sleepTimerMap** is used to map the unique identifiers of sleeping threads to the duration for which they are supposed to sleep.

Basic implementation idea
The **timerInterrupt()** method is called by the Machine after every 500 ticks by default.

When a thread calls **waitUntil(long x)**, it is added to the **sleepQueue**, while the duration of its sleep is mapped within the **sleepTimerMap** against its unique identifier. It is then put to sleep.

When the **timerInterrupt()** is called, it checks whether any of the threads currently sleeping have slept for a duration equal to or greater than the duration they are supposed to sleep for, as indicated by the **sleepTimerMap**. If yes, they are woken up and added to the ready queue.

Tests
The **‘selfTest()’** method inside the **Alarm.java** file tests the functionality of the Alarm class.
It creates five threads and sends them to sleep by calling **waitUntil(long x)** with randomized values of **x** and then ensuring that they wake up at a time equal to or greater than the time when they were supposed to wake up.

Task - 4:

Location of change in source code
Changes were made in the file **'nachos/threads/Communicator.java'** by updating the constructor, **speak(int word)** and **listen()** methods.

Data structures used
Two **Condition2** objects are used to synchronize listening and speaking between threads. A **Lock** object is used to protect access to an atomic task of listening or speaking. A **boolean** is used to keep track of whether any thread has spoken a word through this **Communicator** object and the spoken word is stored in an **int**.

Basic implementation idea
The **Condition2** objects are initialized using the same **Lock** object to protect access.

When a thread calls **speak(int word)** on a **Communicator** object, it firstly acquires the lock, then repeatedly checks whether a word has been spoken, in a loop. If a word has indeed been spoken, it wakes up all threads sleeping on the **Condition2** object named **listenCondition** and then goes to sleep on the **Condition2** object named **speakCondition**. If no word has been spoken, it writes its spoken word into the **int** and makes the **boolean** marker true to indicate that a word has been spoken. After that it wakes up all threads sleeping on the **Condition2** object named **listenCondition** and then goes to sleep on the **Condition2** object named **speakCondition**. Upon being woken up from the sleep, it releases its lock and returns.

When a thread calls **listen()** on a **Communicator** object, it firstly acquires the lock, then repeatedly checks whether a word has been spoken, in a loop. It keeps going to sleep on the **Condition2** object named **listenCondition** until a word has been spoken. When a word has been spoken, it listens to the spoken word and then wakes up all threads sleeping on the **Condition2** object named **speakerCondition**. Finally it releases the lock and returns the heard word.

Tests
The **'selfTest()'** method inside the **Communicator.java** file tests the functionality of the Communicator class.
It creates five threads and calls **listen()** on the same **Communicator** object from them.
It creates a sixth thread and calls **speak()** five times on the same **Communicator** object as before, ensuring that a random distribution of the listening threads hear the words spoken by it.

Part - 2

Task - 1:

__Location of change in source code__
Changes were made in **‘nachos/userprog/UserProcess.java’** in the **handleRead(...)**, **handleWrite(...)**, **handleHalt(...)** and **handleSyscall(...)** methods.

__Data structures used__
Two **‘OpenFile’** objects are used to store the references to the input and output streams. A **static integer processCounter** is used to enforce a unique **integer processID** for each **UserProcess** instance. Two static final integers are used to store the identity of the root process and the maximum permissible length for strings. A **Lock** object is used to protect access to the **static integer processCounter** since it must be assigned and then incremented whenever a new **UserProcess** object is instantiated.

__Basic implementation idea__
The **handleHalt()** method enforces that the halting process must be the root process.
The **handleRead()** method firstly sanitizes the file descriptor to read from, the virtual address to store the read data in and the number of bytes to be read. After that it attempts to read the requested number of bytes from the intended source and validates whether the read was successful. If successful, it stores the read data at the specified virtual address. It returns the status code accordingly.
The **handleWrite()** method firstly sanitizes the file descriptor to write to, the virtual address referencing the data that is to be written and the number of bytes to be written. It then attempts to write the requested number of bytes from the address provided to the file indicated. After that it evaluates whether the write to the file was successful and returns status codes accordingly.

__Tests__
The C program **“nachos/test/myprog.c”** tests the functionality of the read, write and halt system calls.
It reads input from the console to test **handleRead(...)**, writes output to the console to test **handleWrite(...)** and attempts to halt from a non-root process to test **handleHalt()**.

Task - 2:

Location of change in source code

Changes were made in **‘nachos/userprog/UserKernel.java’** creating and/or modifying the **initialize(...)**, **addPhysicalPage(...)** and **fetchPhysicalPage()** methods and in **‘nachos/userprog/UserProcess.java’** creating and/or modifying the **tryAllocate(...)**, **load(...)**, **loadSections(...)**, **unloadSections(...)**, **readVirtualMemory(...), writeVirtualMemory(...)** and **handleExit(...)** methods.

Data structures used

A **LinkedList<Integer>** named **physicalPageList** is used to store the page numbers of available physical pages. A **Lock** object is used to protect access to this **physicalPageList**. A **static integer numberOfPhysicalPages** is used to store the total number of physical pages in our Machine.

Basic implementation idea

Inside UserKernel.java

The **initialize(...)** method initializes the list of available physical pages with the total number of pages in the system.

The **addPhysicalPage(...)** method adds a page with the page number passed to it, to the list of available physical pages. If the provided page number is invalid, it turns false. Otherwise, it returns true. Access protection is maintained by acquiring and releasing the lock.

The **fetchPhysicalPage()** method checks if any physical pages are available to be allocated to the calling process and returns the first available page. If none are available, returns -1. Access protection is maintained by acquiring and releasing the lock.

Inside UserProcess.java

The **load(...)** method . It uses **tryAllocate(...)** to allocate individual pages during the resource allocation phase. If it fails to allocate necessary resources for the executable, it calls **unloadSections(...)**. If resources have been successfully allocated, it loads the necessary data onto the allocated pages by calling **loadSections(...)**.

The **tryAllocate(...)** method attempts to allocate the requested number of pages to a section of the executable. Returns a boolean indicating success or failure.

The **unloadSections()** method deallocates all previously allocated resources for this **UserProcess** instance and adds the freed physical pages to the **UserKernel**’s list of available physical pages.

The **loadSections(...)** method allocates memory for this process, and loads the COFF sections into memory.

The **readVirtualMemory(...)** method takes the virtual address to read from, the buffer to store read data in, the offset to start reading from and the number of bytes to read. Firstly, it sanitizes the input arguments for invalid values. After that it resolves physical addresses from the virtual address and offset. Finally it iterates through the resolved physical pages, reads the requested data into the buffer and returns the number of bytes successfully read.
The **writeVirtualMemory(...)** method takes the virtual address to write to, the buffer containing the data to be written, the offset to start reading from the buffer and the number of bytes to write. Firstly, it sanitizes the input arguments for invalid values. After that it resolves physical addresses from the virtual address and offset. Finally it iterates through the resolved physical pages, writes the provided data from the buffer into the physical pages and returns the number of bytes successfully written.
The **handleExit(...)** method enforces the deallocation of previously allocated resources by calling **unloadSections()** before exiting the current process.

Tests
The C program **“nachos/test/myprog.c”** tests the functionality of the read, write and halt system calls.
The program executes other .coff files with stream inputs and outputs to test **readVirtualMemory(...)** and **writeVirtualMemory(...)** methods. Successful execution of these .coff files also tests the allocation and deallocation of resources involving the **addPhysicalPage(...)** and **fetchPhysicalPage()** methods in **UserKernel.java** as well as the **tryAllocate(...)**, **load(...)**, **loadSections(...)**, **unloadSections(...)** and **handleExit(...)** methods.

Task - 3:

Location of change in source code

Changes were made in **‘nachos/userprog/UserProcess.java’** in the **handleExec(...)**, **handleJoin(...)**, **handleExit(...)** and **handleSyscall(...)** methods.

Data structures used

An **ArrayList<UserProcess>** named **childProcesses** is used to maintain a list of all child processes that have been forked by this process. A **UserProcess** named **parentProcess** is used to store a reference to the parent process that instantiated this process. A **HashMap<Integer, Integer>** named **childProcessExitStatus** is used to map the processIDs child processes forked from this process to their exit statues. This **HashMap** is protected by a **Lock** object named **exitStatusLock**. A **UThread** object named **processThread** is used to form this process from its parent process’s thread.

Basic implementation idea

The **handleJoin(...)** method firstly sanitizes the input arguments. Next, it fetches the child process to join to, from its list of child processes. Upon successful fetching of the child process, it sets the **parentProcess** of the child process to be null, removes the child process from its own list of child processes and then, protected by acquiring and releasing the **exitStatusLock**, fetches the exit status of the child process. Finally it attempts to write the exit status of the child process to the provided virtual memory address for later access and returns a status code indicating success or failure.
The **handleExec(...)** method firstly sanitizes the provided arguments. After that it attempts to fetch the name of the executable file from the provided virtual address. Upon successful fetching of the file name, it fetches the provided arguments. Having fetched all of this, it instantiates a new **UserProcess** object and assigns it to execute the executable. It also adds the newly instantiated process to its list of child processes and finally returns the processID of the child process.
The **handleExit(...)** method ensures that before the current process exits, it removes its own reference from the **parentProcess** of all of its children and also empties out its list of child processes.

Tests

The C program **“nachos/test/myprog.c”** tests the functionality of the join, exec and exit system calls.
It calls the exec system call on other .coff files and then joins the new processes. This ensures that the calling process does not exit before the children, the children successfully execute the passed executables and that the parent, upon exit, removes its reference from the children.