PA-2

Name: Raghunandan Gajanan Bhat

SUNetID: rgbhat@syr.edu

✪ **[R-1]:** `./nachos -K`

```
rgbhat@lcs-vc-cis486-2:~/PA/pa2/student/nachos/code/build.linux$ ./nachos -K
*** thread 0 looped 0 times
*** thread 0 looped 1 times
*** thread 0 looped 2 times
*** thread 0 looped 3 times
*** thread 0 looped 4 times
*** thread 1 looped 0 times
*** thread 1 looped 1 times
*** thread 1 looped 2 times
*** thread 1 looped 3 times
*** thread 1 looped 4 times
*** thread 2 looped 0 times
*** thread 2 looped 1 times
*** thread 2 looped 2 times
*** thread 2 looped 3 times
*** thread 2 looped 4 times
*** thread 3 looped 0 times
*** thread 3 looped 1 times
*** thread 3 looped 2 times
*** thread 3 looped 3 times
*** thread 3 looped 4 times
^C
Cleaning up after signal 2
rgbhat@lcs-vc-cis486-2:~/PA/pa2/student/nachos/code/build.linux$
```

When `ThreadTest()` is called, it creates 3 new threads `thread-1, thread-2 and thread-3`. It calls `Thread::Fork()` method which allocates the execution stack for `SimpleThread()` function for all three threads and puts them on the ready queue, so that when `SWITCH`(context switch) is called, `SimpleThread()` will be called for each of the threads.

Then `SimpleThread(0)` is called, which prints `'*** thread 0 looped 0 times'` 5 times for `thread-0`. When `thread-0` is done, there will be 3 threads waiting to be executed in ready queue. These threads are executed in the order in which they are created, which is first come first serve basis.

Here nachos' scheduler is behaving like First In First Out (FIFO) also known as First Come First Serve(FCFS).

✪ **[R-2]:** `./nahcos -K`

When `kernel->currentThread->Yield()` is uncommented, currently running thread gives up the CPU for other threads which are in ready queue(if any). Here every thread is allowed to execute one print

statement at once and gives up CPU for another thread in the ready queue by calling `Yield()` system call. This is also called as co-operative approach for switching between processes. Therefore, there is a difference between the output compared to [R-1].

Here nachos scheduler behaves like pre-emptive scheduler (like Round Robin but no time slices).

```
rgbhat@lcs-vc-cis486-2:~/PA/pa2/student/nachos/code/build.linux$ ./nachos -K
*** thread 0 looped 0 times
*** thread 1 looped 0 times
*** thread 2 looped 0 times
*** thread 3 looped 0 times
*** thread 0 looped 1 times
*** thread 1 looped 1 times
*** thread 2 looped 1 times
*** thread 3 looped 1 times
*** thread 0 looped 2 times
*** thread 1 looped 2 times
*** thread 2 looped 2 times
*** thread 3 looped 2 times
*** thread 0 looped 3 times
*** thread 1 looped 3 times
*** thread 2 looped 3 times
*** thread 3 looped 3 times
*** thread 0 looped 4 times
*** thread 1 looped 4 times
*** thread 2 looped 4 times
*** thread 3 looped 4 times
^C
Cleaning up after signal 2
rgbhat@lcs-vc-cis486-2:~/PA/pa2/student/nachos/code/build.linux$
```

✪ **[R-2]:** `./nachos -K` with active timer

When device timer is activated, it starts a software clock. This timer raises interrupts every so many milliseconds. When interrupt is raised, the currently running process is stopped. The state of the process is saved by nachos. Then the next process is executed for certain time slice until the next time interrupt occurs and it is replaced by some other process in the queue.

Here each thread allowed to execute for certain amount of time(time slice) before being replaced by another process. This process scheduling is same as Round Robin scheduling.

```
rgbhat@lcs-vc-cis486-2:~/PA/pa2/student/nachos/code/build.linux$ ./nachos -K
*** thread 0 looped 0 times
*** thread 1 looped 0 times
*** thread 2 looped 0 times
*** thread 3 looped 0 times
*** thread 0 looped 1 times
*** thread 1 looped 1 times
*** thread 3 looped 1 times
*** thread 0 looped 2 times
*** thread 1 looped 2 times
*** thread 2 looped 1 times
*** thread 3 looped 2 times
*** thread 0 looped 3 times
*** thread 1 looped 3 times
*** thread 2 looped 2 times
*** thread 3 looped 3 times
*** thread 1 looped 4 times
*** thread 2 looped 3 times
*** thread 3 looped 4 times
*** thread 0 looped 4 times
*** thread 2 looped 4 times
^C
Cleaning up after signal 2
rgbhat@lcs-vc-cis486-2:~/PA/pa2/student/nachos/code/build.linux$
```

Task 2: Nachos pre-emptive multiprogramming

Task 2.1: Nachos accepting multiple user program names.

✪ **[R-1]:** The list of user programs must be stored in some data structure, so that they can be executed later. Here use the List template, which accepts any type and creates a list of the given type.

```
184     char *debugArg = "";
185     char *userProgName = NULL;        // default is not to execute a user prog
186     List<char*> *userProgramList = new List<char*>(); //list to store multiple user programs
187     bool threadTestFlag = false;
188     bool consoleTestFlag = false;
```

Use List template to create a list of character pointers where each pointer points to a character array. When a new object of `List<char*>` is created, constructor returns a pointer to that list which is stored in `userProgramList` variable.

```
210         }
211         else if (strcmp(argv[i], "-x") == 0) {
212             ASSERT(i + 1 < argc);
213             //append userProgName at the end of list
214             userProgName = argv[i + 1];
215             userProgramList->Append(userProgName);
216                 //cout << userProgName << " append successful\n";
217             i++;
218         }
```

List template class also defines a method `List<T>::Append()` which appends an item at the end of the list. This method can be called using the `List<char*>` object -`userProgramList` and pass user program name(`userProgName`) as a parameter to the method.

The `List<T>::Append()` method creates an element of type `ListElement` which contains the user program name as value stored in the object. Then `List<T>::Append()` method checks for duplication of user program in the list and appends the given user program to the end of the list. This appending operation happens for all the user programs passed as a command line argument and `userProgramList` variable is used to access those list of user programs.

✪ [R-2]: Testing the user programs

1. Now using `userProgramList`, iterate through the list if it is not empty. `List<T>::IsEmpty()` method check if the list is empty. If list is not empty then using `List<T>::NumInList()` get the number elements stored in the list and iterate all of the elements.
2. Using `List<T>::RemoveFront()` method, remove each element of the list one by one from the front of the list and print the list element

```
310     //iterate the list if not empty list
311     if (!userProgramList->IsEmpty()){
312         //remove from front one by one
313         int userProgramCount = userProgramList->NumInList();
314         for(int i=0; i<userProgramCount; i++){
315                 cout << "Program ["<< i <<"] = " << userProgramList->RemoveFront() <<"\n";
316         }
317     }
318
```

3. **Testcases**
   a. `./nachos -x ../test-pa/prog1 -x ../test-pa/prog2`

```
rgbhat@lcs-vc-cis486-2:~/PA/pa2/student/nachos/code/build.linux$ ./nachos -x ../test-pa/prog1 -x ../test-pa/prog2
Program [0] = ../test-pa/prog1
Program [1] = ../test-pa/prog2
^C
Cleaning up after signal 2
rgbhat@lcs-vc-cis486-2:~/PA/pa2/student/nachos/code/build.linux$
```

   b. `./nachos -x ../test-pa/prog1 -x ../test-pa/prog2 -x ../noprog`

```
rgbhat@lcs-vc-cis486-2:~/PA/pa2/student/nachos/code/build.linux$ ./nachos -x ../test-pa/prog1 -x ../test-pa/prog2 -x ../noprog
Program [0] = ../test-pa/prog1
Program [1] = ../test-pa/prog2
Program [2] = ../noprog
^C
Cleaning up after signal 2
rgbhat@lcs-vc-cis486-2:~/PA/pa2/student/nachos/code/build.linux$
```