Academic Report
Cover Page

CIS 657

OS Project 2

Ameya Kale
SU ID: 304967094
Email Address: akkale@syr.edu
October 14, 2022

Learning Objective

After completing this project assignment, students can

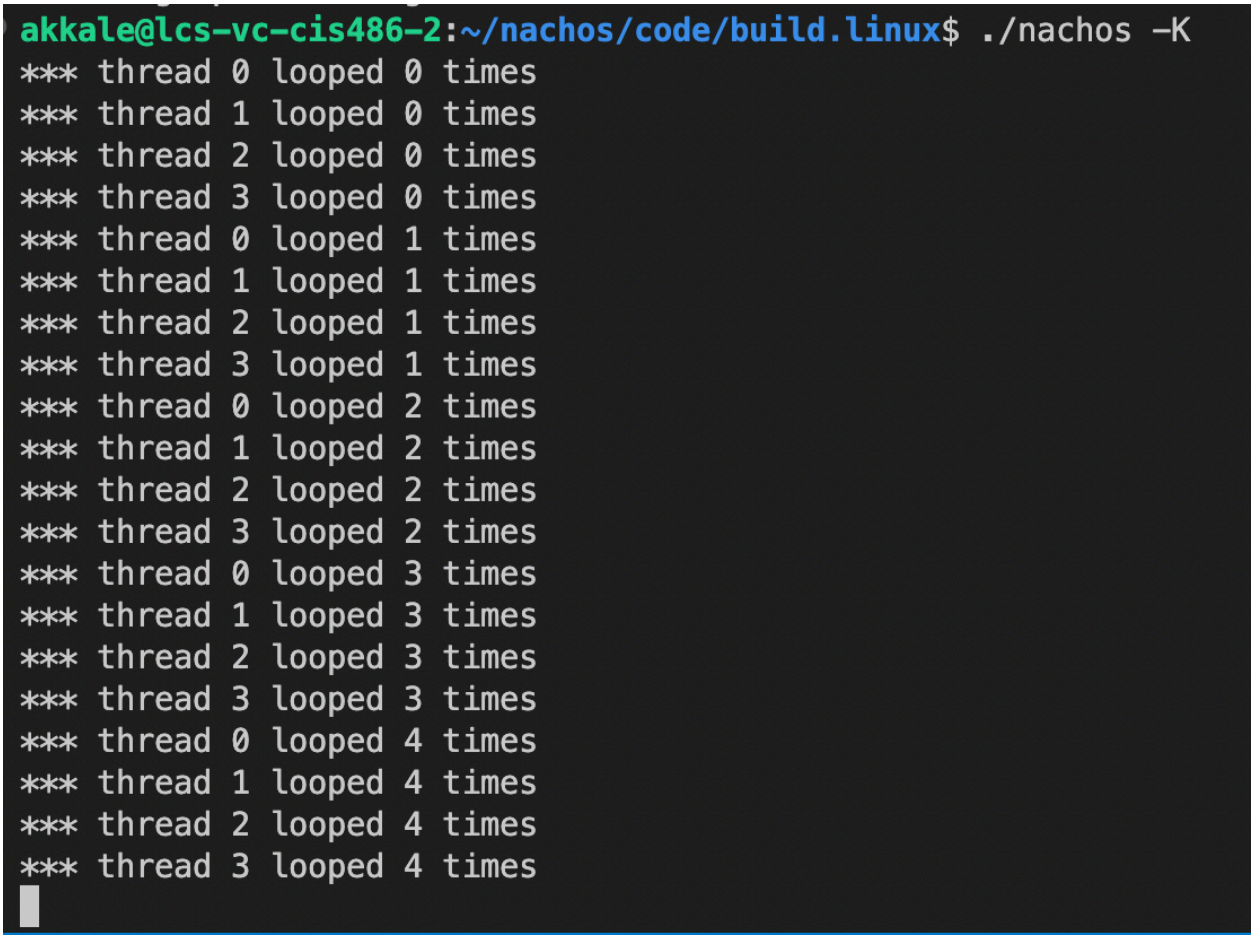
1. describe the difference between a preemptive and a non-preemptive scheduler,
2. describe the role of Yield() function,
3. describe the effect of the timer device in thread execution sequence,
4. eventually implement a non-preemptive multi-programming in Nachos.

Question 1

Output before:

```
akkale@lcs-vc-cis486-2:~/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
```

Output after uncommenting the given line:



1. We can see that before the threads were getting executed one-by-one. After uncommenting the given line, the threads are executing in a round-robin fashion.
2. We uncommented the line “kernel->currentThread->Yield();”. This line stops the current running thread preemptively. This means that the execution of the current thread was stoped or interrupted before the thread actually finished its execution. This thread is then put back into the ready queue and the scheduler selects the next thread to run from the ready queue.
3. This is the reason why we see a different output than the one before. We are preemptively stopping the current thread execution after executing the printf line and putting it back into the ready queue.

Question 2

Output:

```
akkale@lcs-vc-cis486-2:~/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 2 looped 1 times
*** thread 3 looped 1 times
*** thread 0 looped 2 times
*** thread 1 looped 1 times
*** thread 2 looped 2 times
*** thread 3 looped 2 times
*** thread 0 looped 3 times
*** thread 1 looped 2 times
*** thread 2 looped 3 times
*** thread 0 looped 4 times
*** thread 1 looped 3 times
*** thread 2 looped 4 times
*** thread 3 looped 3 times
*** thread 1 looped 4 times
*** thread 3 looped 4 times
```

1. When we uncomment line 99 in file kernel.cc, an object of class Alarm is created.
2. The constructor of Alarm class generates interrupts to the CPU at random intervals of time. Therefore, we see a change in the output.
3. When an interrupt is generated, context switch operation is performed and the current thread which was running on the CPU is put back into the ready queue.
4. The CPU then serves the interrupt.
5. Once done, the scheduler picks the next thread to run from the ready queue (in this case, the first one from the ready queue). The CPU then runs this thread.
6. We can also see that at the end only threads 1 and 3 execute as threads 0 and 2 have finished their entire execution.

Question 3

1. At this point, there is only one user program thread (called main thread) created in the Nachos, as every process has one thread in it.

```
156   void
157   RunUserProg(void *filename) {
158       AddrSpace *space = new AddrSpace;
159       ASSERT(space != (AddrSpace *)NULL);
160       if (space->Load((char*)filename)) {  // load the program into the space
161           space->Execute();         // run the program
162       }
163       ASSERTNOTREACHED();
164   }
```

2. The above block of code in main.cc is responsible for creating address space to run a user program.

Question 4

1. The first argument to the function Fork is a pointer to the function for which we want the threads to be created. The forked threads will then execute this function.

2. If we use a pointer to the function RunUserProg() as the first argument of Fork(), then we also need to pass userProgName as the argument.
   - After doing the above, when we call the Fork function, it will create a new thread for each userProgName, i.e. for each program name passed while running nachos.

Output before passing RunUserProg to Fork()

```
akkale@lcs-vc-cis486-2:~/nachos/code/build.linux$ ./nachos -x ../test1/read
Read system call made by main
Read system call made by main
Read system call made by main
Read system call made by main
Read system call made by main
Exit system call made by main
```

Output after passing RunUserProg to Fork()

```
akkale@lcs-vc-cis486-2:~/nachos/code/build.linux$ ./nachos -x ../test1/read
Read system call made by ../test1/read
Read system call made by ../test1/read
Read system call made by ../test1/read
Read system call made by ../test1/read
Read system call made by ../test1/read
Exit system call made by ../test1/read
```

We can see that before the main thread was executing the program. But after passing d RunUserProg to the Fork() function, a new thread was gets created to execute each program specified by userProgName.

Question 5

Approach:

1. From threadtest.cc we can learn how the code for creating a thread is implemented.
2. First, we define a pointer object *t of the class Thread with thread name as “forked thread”.
3. Then we call the Fork function on our object t by passing the function name and the arguments to be executed.
4. Therefore, we can modify our main.cc file with a similar code to create a new thread for our userProgName array.
5. We will create a for loop to iterate over and create a new thread for each program specified in the userProgName array.
6. Inside the for loop, we will create a new pointer object of class Thread and pass in the program name as the name of the thread.
7. We will then call the Fork function by passing in the function RunUserProg as the first argument to execute the programs and then the element userProgName[index] as the second argument as the name of the program to be executed.

```
320     //Ameya: For loop to iterate over each userProgName element and create
321             //respective threads for them
322     if (userProgName != NULL) {
323         for(int index = 0; index < numOfxFlags; index++){
324             Thread *t = new Thread(userProgName[index]);
325             t->Fork((VoidFunctionPtr) RunUserProg, (void *) userProgName[index]);
326         }
327     }
```

Conclusion:

All the questions were answered and all the codes were executed successfully with the desired output. The program can now run multiple user programs one by one.

```
akkale@lcs-vc-cis486-2:~/nachos/code/build.linux$ ./nachos -x ../test1/read -x ../test1/write
Read system call made by ../test1/read
Read system call made by ../test1/read
Read system call made by ../test1/read
Read system call made by ../test1/read
Read system call made by ../test1/read
Exit system call made by ../test1/read
Write system call made by ../test1/write
Write system call made by ../test1/write
Write system call made by ../test1/write
Write system call made by ../test1/write
Write system call made by ../test1/write
Exit system call made by ../test1/write
```