Academic Report Cover Page

CIS 657

OS Project 3

Ameya Kale
SU ID: 304967094
Email Address: akkale@syr.edu

November 7, 2022

Learning Objectives

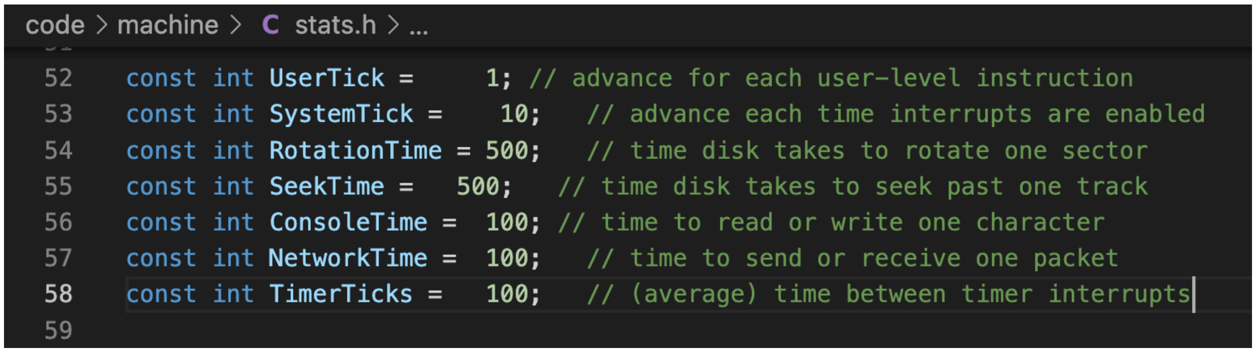
After completing this project assignment, students can

1. describe how timer device is emulated in Nachos, (i.e., timer.[cc|h]),
2. describe how timer interrupt service routine is called and how it is implemented,
3. describe Round-Robin Scheduling with Quantum,
4. implement a Round-Robin Scheduling with Quantum in Nachos,
5. describe how a Timer Interrupt Service routine works and
6. modify a Timer Interrupt Service routine as a part of Round-Robin Scheduling with Quantum.

Question 1:

Ans:

- The value of TimerTicks is initialized in the stats.h file as 100.
- Therefore, a timer interrupt will occur after every 100 timer ticks.
- However, if the value of the variable 'randomize' is TRUE, the next timer interrupt is calculated with using the formula, '1 + (RandomNumber() % (TimerTicks * 2)).'
- Here, the function RandomNumber() just returns a random number of integer type.



code > machine > C stats.h > ...

```
const int UserTick =     1; // advance for each user-level instruction
const int SystemTick =    10;   // advance each time interrupts are enabled
const int RotationTime = 500;   // time disk takes to rotate one sector
const int SeekTime =   500;   // time disk takes to seek past one track
const int ConsoleTime =  100; // time to read or write one character
const int NetworkTime =  100;   // time to send or receive one packet
const int TimerTicks =   100;   // (average) time between timer interrupts
```

Question 2:

Ans

- The type of the variable 'randomSlice' is of boolean type.
- The value of 'randomSlice' in line number 99 of file kernel.cc is TRUE.

Question 3:
Ans:
- The file ‘interrupt.cc' in the ‘machine’ directory implements the function ‘OneTick()’.
- This function updates the values of ‘totalTicks’ and ‘systemTicks’.
- If the ‘status’ is ‘SystemMode’, then the values are updated by adding the value of ‘SystemTick’ to them. Else, they are updated by adding the value of ‘UserTick’.
- In other words, this functions increases the required time for a program with the help of ‘SystemTick’ and ‘UserTick’ variables.
- The Run() function in the Machine class which is in the file mipssim.cc contains the another call to OneTick() when a user program is executed.

Question 4:
Ans:
- The output shows the value of the totalTicks.
- This value increases by 100, i.e. after every timer ticks.
- Also, when the doRandom variable is equal to false, the timeTick() occurs at its set value i.e., 100 timer ticks.

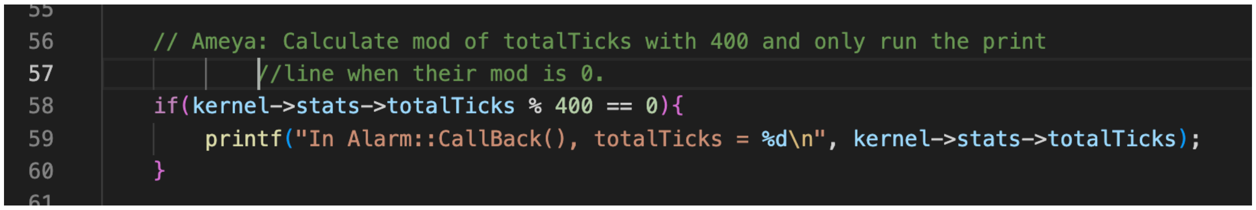
Question 5:
Ans:
1)
  - The value of totalTicks is printed after a regular interval of 100 timer ticks.
  - The value of x is 100, i.e. after every regular interval of 100 timer ticks.

2)
- The stats.h file in the machine directory declares the variable named totalTicks. The value of totalTicks is defined in the interrupt.cc file.

3)
- Pseudo Code:

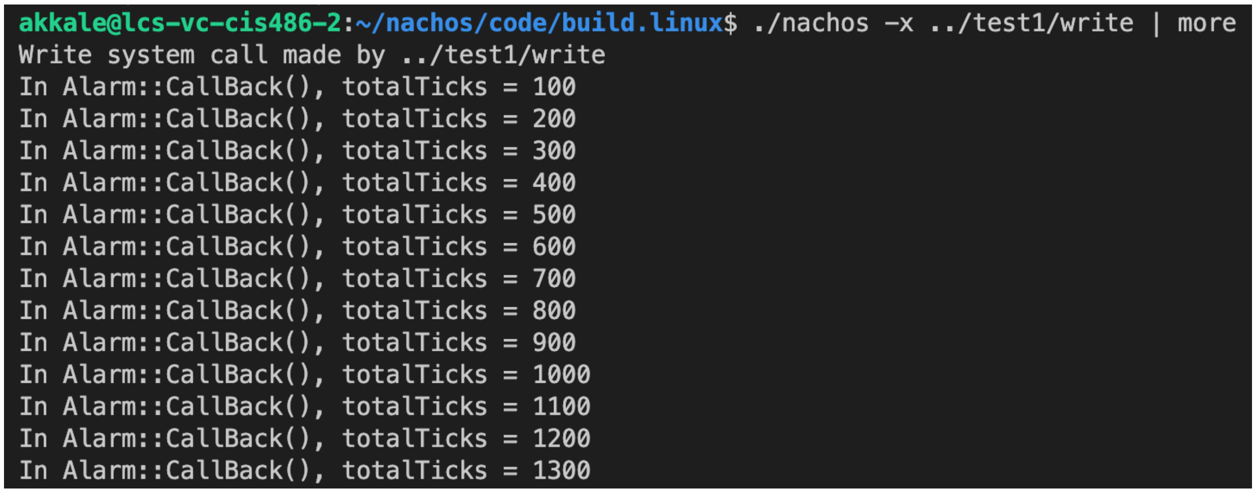
```
    if (remainder of totalTicks divided by 4*100 == 0) {
        print the message "In Alarm::CallBack(), totalTicks = totalTicks”
    }
```

- Implementation in code:



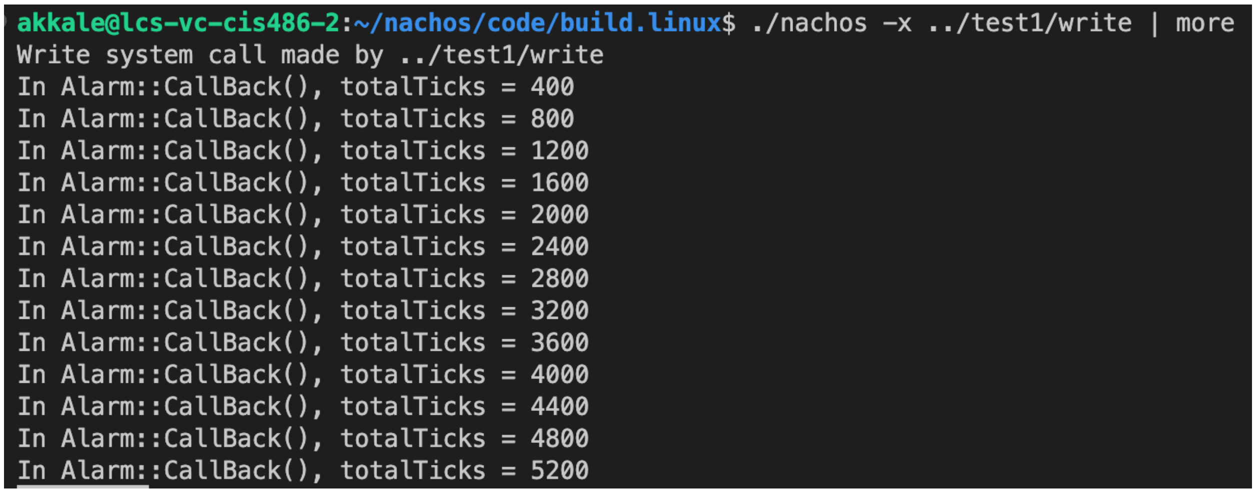
```
    // Ameya: Calculate mod of totalTicks with 400 and only run the print
            //line when their mod is 0.
    if(kernel->stats->totalTicks % 400 == 0){
        printf("In Alarm::CallBack(), totalTicks = %d\n", kernel->stats->totalTicks);
    }
```

- Output before:



```
akkale@lcs-vc-cis486-2:~/nachos/code/build.linux$ ./nachos -x ../test1/write | more
Write system call made by ../test1/write
In Alarm::CallBack(), totalTicks = 100
In Alarm::CallBack(), totalTicks = 200
In Alarm::CallBack(), totalTicks = 300
In Alarm::CallBack(), totalTicks = 400
In Alarm::CallBack(), totalTicks = 500
In Alarm::CallBack(), totalTicks = 600
In Alarm::CallBack(), totalTicks = 700
In Alarm::CallBack(), totalTicks = 800
In Alarm::CallBack(), totalTicks = 900
In Alarm::CallBack(), totalTicks = 1000
In Alarm::CallBack(), totalTicks = 1100
In Alarm::CallBack(), totalTicks = 1200
In Alarm::CallBack(), totalTicks = 1300
```

- Output after:

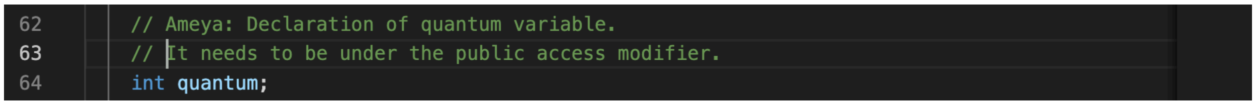


```
akkale@lcs-vc-cis486-2:~/nachos/code/build.linux$ ./nachos -x ../test1/write | more
Write system call made by ../test1/write
In Alarm::CallBack(), totalTicks = 400
In Alarm::CallBack(), totalTicks = 800
In Alarm::CallBack(), totalTicks = 1200
In Alarm::CallBack(), totalTicks = 1600
In Alarm::CallBack(), totalTicks = 2000
In Alarm::CallBack(), totalTicks = 2400
In Alarm::CallBack(), totalTicks = 2800
In Alarm::CallBack(), totalTicks = 3200
In Alarm::CallBack(), totalTicks = 3600
In Alarm::CallBack(), totalTicks = 4000
In Alarm::CallBack(), totalTicks = 4400
In Alarm::CallBack(), totalTicks = 4800
In Alarm::CallBack(), totalTicks = 5200
```

Question 6:

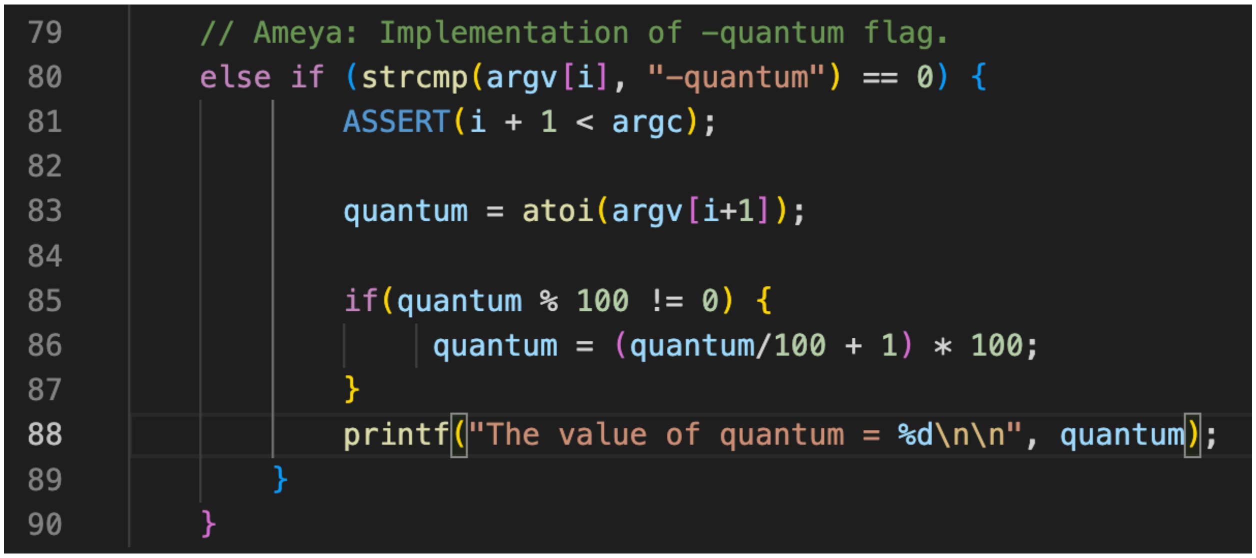
Approach:

1) We need to implement the quantum flag and functionality to it to take the value of the quantum as the input.
2) To store the value of quantum, we can declare the variable quantum in the kernel.h file.



```
// Ameya: Declaration of quantum variable.
// It needs to be under the public access modifier.
int quantum;
```

3) We can modify our kernel.cc file to implement the -quantum flag like this:



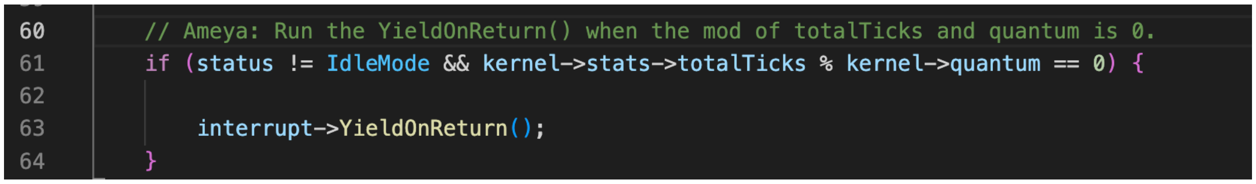
```
    // Ameya: Implementation of -quantum flag.
    else if (strcmp(argv[i], "-quantum") == 0) {
            ASSERT(i + 1 < argc);

            quantum = atoi(argv[i+1]);

            if(quantum % 100 != 0) {
                quantum = (quantum/100 + 1) * 100;
            }
            printf("The value of quantum = %d\n\n", quantum);
        }
    }
```

4) First, we make sure that we are not at the end of argv.
5) If it not, we then store the next argument value into the quantum variable. But before we do that, we multiply the argument by 100, as we need to round to the next multiple of 100 for the given quantum value.
6) Now, we need to only call the YieldOnReturn() function after every quantum.
7) To do this, we can modify our alarm.cc file like this:

8) The YieldOnReturn() function will only be called when the remainder of totalTicks divided by the quantum is equal to zero. In other words, when the quantum has expired.

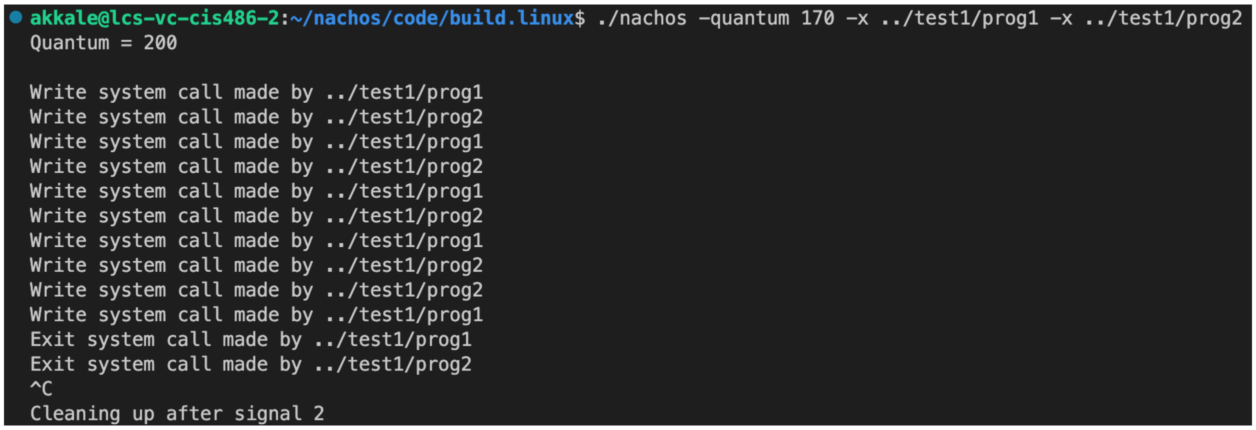


```
    // Ameya: Run the YieldOnReturn() when the mod of totalTicks and quantum is 0.
    if (status != IdleMode && kernel->stats->totalTicks % kernel->quantum == 0) {

        interrupt->YieldOnReturn();
    }
```

Output:

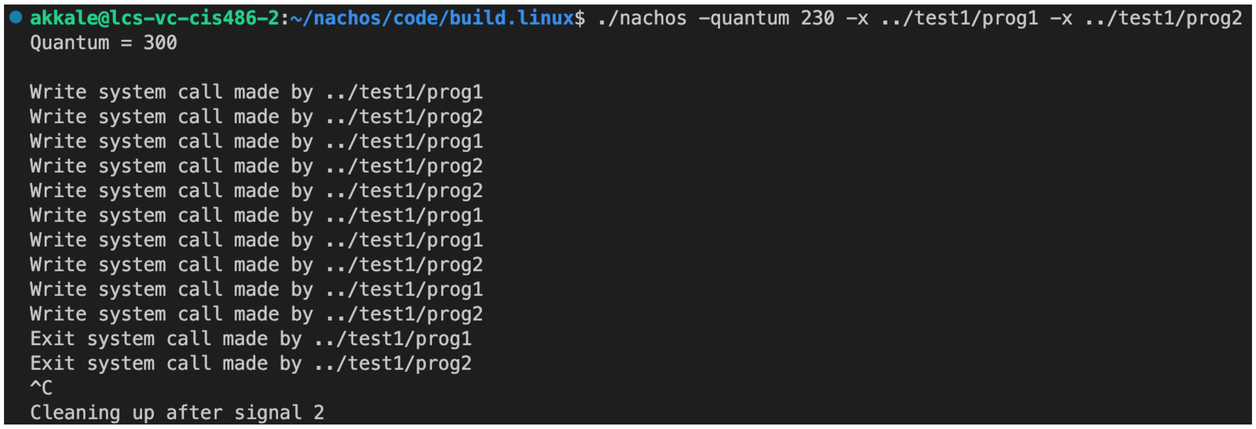
- When quantum = 170



```
akkale@lcs-vc-cis486-2:~/nachos/code/build.linux$ ./nachos -quantum 170 -x ../test1/prog1 -x ../test1/prog2
Quantum = 200

Write system call made by ../test1/prog1
Write system call made by ../test1/prog2
Write system call made by ../test1/prog1
Write system call made by ../test1/prog2
Write system call made by ../test1/prog1
Write system call made by ../test1/prog2
Write system call made by ../test1/prog1
Write system call made by ../test1/prog2
Write system call made by ../test1/prog2
Write system call made by ../test1/prog1
Exit system call made by ../test1/prog1
Exit system call made by ../test1/prog2
^C
Cleaning up after signal 2
```

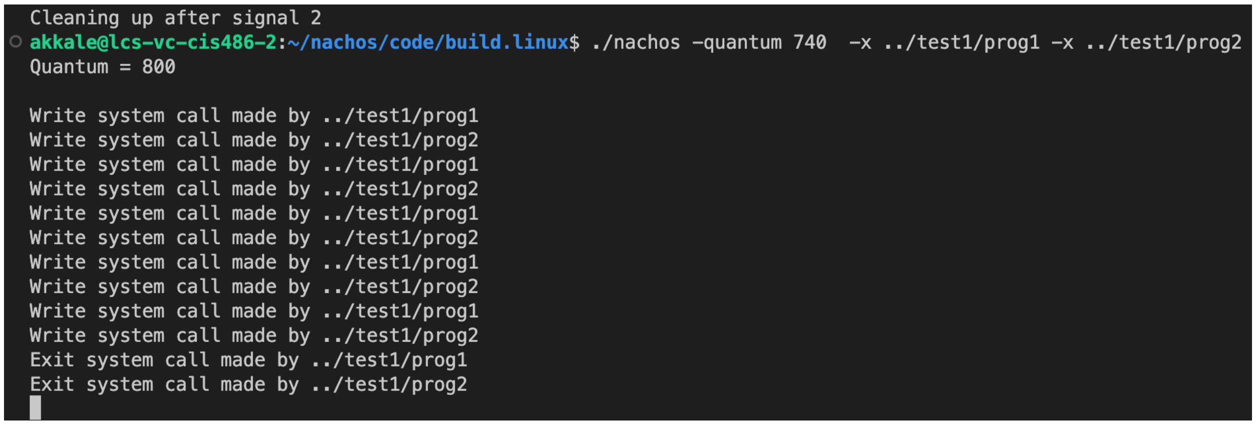
- When quantum = 230



```
akkale@lcs-vc-cis486-2:~/nachos/code/build.linux$ ./nachos -quantum 230 -x ../test1/prog1 -x ../test1/prog2
Quantum = 300

Write system call made by ../test1/prog1
Write system call made by ../test1/prog2
Write system call made by ../test1/prog1
Write system call made by ../test1/prog2
Write system call made by ../test1/prog2
Write system call made by ../test1/prog1
Write system call made by ../test1/prog1
Write system call made by ../test1/prog2
Write system call made by ../test1/prog1
Write system call made by ../test1/prog2
Exit system call made by ../test1/prog1
Exit system call made by ../test1/prog2
^C
Cleaning up after signal 2
```

- When quantum = 740



```
Cleaning up after signal 2
akkale@lcs-vc-cis486-2:~/nachos/code/build.linux$ ./nachos -quantum 740  -x ../test1/prog1 -x ../test1/prog2
Quantum = 800

Write system call made by ../test1/prog1
Write system call made by ../test1/prog2
Write system call made by ../test1/prog1
Write system call made by ../test1/prog2
Write system call made by ../test1/prog1
Write system call made by ../test1/prog2
Write system call made by ../test1/prog1
Write system call made by ../test1/prog2
Write system call made by ../test1/prog1
Write system call made by ../test1/prog2
Exit system call made by ../test1/prog1
Exit system call made by ../test1/prog2
```

<u>Conclusion:</u>

Round-Robin Scheduler was successfully implemented with the help of the timer.