University of Washington, Bothell
CSS 501: Data Structures and Object-Oriented Programming I

Example Program 5: Lists and Queues and Simulation! Oh, My!

Concurrency Simulator

Programs executed concurrently on a uniprocessor system appear to be executing at the same time, but in reality the single CPU alternates between the programs, executing some number of instructions from each program before switching to the next. You are to simulate the concurrent execution of multiple, simple programs on such a system and determine the output that they will produce.

The program that is currently being executed is said to be *running*, while all programs awaiting execution are said to be *ready*. A program consists of a sequence of *statements*, one per line, followed by an `end` statement. The statements in our simple language only manipulate variables; there is no flow control (programs are just a sequence of operations). The statements available are listed below.

Statement Type	**Syntax**
Assignment	*variable* = *constant*
Output	`print` *variable*
Begin Mutual Exclusion	`lock`
End Mutual Exclusion	`unlock`
Stop Execution	`end`

A *variable* is any single lowercase alphabetic character and a *constant* is an unsigned decimal integer less than 100. There are only 26 variables in the computer system, and they are shared among the programs (this is often called a *shared memory* model of parallel programming. Thus assignments to a variable in one program affect the value that might be printed by a different program. All variables are initially set to zero.

Each statement requires an integral number of *time units* to execute. The running program is permitted to continue executing instructions for a period of time called its *quantum*, a certain number of time units. When a program's time quantum expires, another ready program will be selected to run. Any instruction currently being executed when the time quantum expires will be allowed to complete.

Programs are queued first-in-first-out for execution in a *ready queue*. The initial order of the ready queue corresponds to the original order of the programs in the input file. This order can change, however, as a result of the execution of `lock` and `unlock` statements.

The `lock` and `unlock` statements are used whenever a program wishes to claim mutually exclusive access to the variables it is manipulating. These statements always occur in pairs, bracketing one or more other statements. A `lock` will always precede an `unlock`, and these statements will never be nested. Once a program successfully executes a `lock` statement, no other program may successfully execute a `lock` statement until the locking program runs and executes the corresponding `unlock` statement. Should a running program attempt to execute a `lock` while another one is already in effect, this program will be placed at the end of the *blocked queue*. Programs blocked in this fashion lose any of their current time quantum remaining. When an `unlock` is executed, any program at the head of the *blocked queue* is moved to the end of the *ready queue*. The first statement this program will execute when it runs will be the `lock` statement that previously failed. Note that it is up to the programs involved to enforce the mutual exclusion protocol through correct usage of `lock` and `unlock` statements. (A renegade program with no `lock`/`unlock` pair could alter any variables it wished, despite the proper use of `lock`/`unlock` by the other programs.)

Input and Output Format

The first line of the input consists of seven integers separated by spaces. These integers specify (in order): the number of programs which follow, the unit execution times for each of the five statement types (in the order given above), and the number of time units comprising the time quantum. The remainder of the input consists of the programs, which are correctly formed from statements according to the rules described above.

All program statements begin in the first column of a line, and there will only be one statement per line. White space appearing in a statement should be ignored. Associated with each program is an identification number based upon its location in the input data (the first program has ID = 1, the second has ID = 2, etc.).

Your output will contain of the output generated by the `print` statements as they occur during the simulation. When a `print` statement is executed, your program should display the program ID, a colon, a space, and the value of the selected variable. Output from separate `print` statements should appear on separate lines. A sample input and correct output are shown below.

Sample Input

```
3 1 1 1 1 1 1
a = 4
print a
lock
b = 9
print b
unlock
print b
end
a = 3
print a
```

```
lock
b = 8
print b
unlock
print b
end
b = 5
a = 17
print a
print b
lock
b = 21
print b
unlock
print b
end
```

Sample Output

```
1: 3
2: 3
3: 17
3: 9
1: 9
1: 9
2: 8
2: 8
3: 21
3: 21
```

Statement of Work

Develop your concurrency simulator so that it reads its input from `cin` and writes its output to `cout`. You will need to implement a **queue** and a **list** class for this assignment; they should be linked implementations. You may refer to the textbook's implementations of these, but you will need to adapt them to your purposes. Do *not* use the textbook's classes verbatim; develop the classes specific to our purposes here.

This program is specifically designed to be too big for you to solve "all at one shot". You will need to very carefully adhere to a systematic development process to be successful. Nevertheless, you should be able to successfully complete the assignment without trouble. The key is to *not* try to solve the whole thing, but rather to break the task down into manageable pieces. An outline of the steps you might take are:

1. Get to work right away and work on this every day.
2. Don't try to write code yet. Instead, develop a clear understanding of *what* this program is supposed to do. While you do this, write down your understanding. This is part of your design. If there is anything you don't understand, ask a question on the class discussion forum to get an answer.
3. Now that you understand what the program should do, *don't write code yet either*. It is still too early. Instead, now start thinking of *how* the program will work. Identify the major data types/classes/instances you will need to create. These will include things like the *ready queue*, the *blocked queue*, a **program**, the **variable store**, etc. Which of these are classes and which are instances? Is it possible that some of these are multiple instances of the same class? Do any of your classes contain data types/structures that should be classes in their own right? If a class is a container, what sort of thing does it contain?
4. It's still not time to write code. For each identified class, take a first stab at listing the operations the class must provide. For each operation, write a brief synopsis, determine and document its parameters and return type, and document any restrictions on its input, output, conditions that must be true before it can be called, error handling (if any), etc.
5. Don't write code just yet! Now think about how you will test your program. I suggest that you consider each class separately and develop a plan for each *separately*. So, for instance, if you have a **queue** class that holds programs, you first might develop a **queue** class that holds ints and get that working. Then you might start on the **program** class. If it contains a **list** of statements, then you would plan to first implement and test the **statement** class, then a **list** of ints, then a **list** of **statements**, then the **program** class, and then finally a **queue** of **program**s. When it comes to the entire program, you might plan on first getting a version working that doesn't support locking, and once that's fine, implement the locking mechanism.
6. Everything you've done so far is your design. Go for a run or bike ride.
7. Take your class definitions and make two (electronic) copies of each; these are the boilerplate comments for your `.h` and `.cpp` files.
8. You are now ready to start coding. Follow your implementation and test plan and get each component working separately and then working integrated together. At any particular point in time, you should have a partially working program. As time progresses, the functionality of what you have will gradually approach the final desired functionality.