CS 3013 Operating Systems — WPI, B Term 2016
Craig E. Wills — Project 3 (30 pts)
Assigned: Friday, September 16, 2016 — Due: Friday, September 30, 2016

Important: This project is to be done by each individual student. This is NOT a group project. The project may be done on the Linux systems of the CCC or on your virtual machine. It will be graded on the CCC Linux systems.

Introduction

This assignment is intended to help you put into practice the concepts of thread coordination and resource sharing by building and executing a distributed computation graph. You will use the facilities of the *pthreads()* library for thread and synchronization routines. Your program must be coded in C or C++.

Problem

The basic idea of this assignment is to create a number of threads where each thread corresponds to a node in a distributed computation. Figure 1 shows a sample distributed computation with four nodes (A-D). The graph shows that Node A executes first. Once it is done then Nodes B and C can execute in parallel. Once *both* B and C are done then Node D can execute.

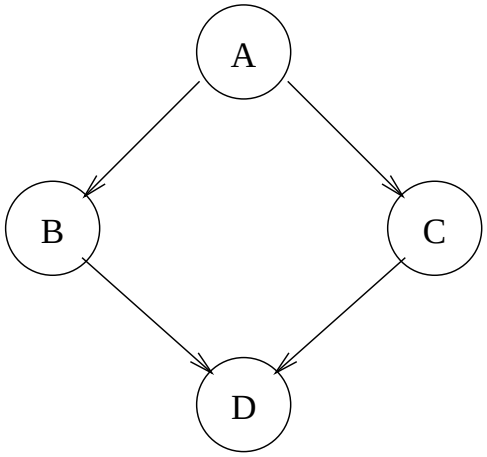

Figure 1: Sample Distributed Computation

Each node label consists of a single capital letter (A-Z) so there can be no more than 26 nodes in any distributed computation. Your code may subtract the ASCII value of the letter 'A' to convert the node label to a numeric identifier between 0 and 25.

The main thread in your program will create a thread for each of the computation nodes using the routine *pthread_create()*. The numeric identifier of the thread is passed as the argument. In creating the threads, your program will need to remember the thread id stored in the first argument to *pthread_create()* for later use. You might call the routine executed by each child thread *Compute()* because it will "compute" a value for the node when it executes.

In the initial part of the project, each node will have a fixed integer value that it "computes" and a fixed number of seconds it will take to compute this value. Information about a graph including nodes, edges, computed value and duration are specified in a textual configuration file that your program needs to read. For example, assume that each node in Figure 1 takes one second to compute a value and that the computed value for A is 1, for B is 2, for C is 3 and for D is 4 then the configuration file would look like:

```
A 1 1
B 2 1 A
C 3 1 A
D 4 1 B C
```

where the first field of each input line is the node label. The second field (fields are space separated) is the value of the node (an integer) and the third field is the computation time of the node (in seconds). Any subsequent fields on a line indicate the nodes that this node is dependent on before it can begin execution. The example configuration file shows that Node A is not dependent on any other node meaning that it can begin execution immediately. Nodes B and C depend on the completion of Node A before they can begin. Node D depends on the completion of both Nodes B and C before it can begin.

In terms of error checking, you need to verify that the first field is a single capitalized letter, the second and third fields are integers and any subsequent fields are letters. One or more spaces may separate each field. One approach is to read an entire line and then use a Linux library routine such as *strtok()* to break a line into string tokens for processing. You can use the Linux library routine *atoi()* to convert a string to its numeric equivalent. See man pages for more details on these routines. If a configuration file is not in the correct format then your program should report an error and terminate. Your program does not need to check for cycles in the computation graph.

You want to create a thread for each node and a semaphore for each edge. In terms of storing information about threads, you may want to have an array of records for each node where you use the numeric identifier of each thread to look up information. This table needs to be global for access by all threads. In terms of storing information about semaphores, there are many ways in which to maintain this information with one possibility being a two-dimensional array storing the semaphore id corresponding to an edge between two nodes.

Once each compute thread begins execution, its actions are summarized in Figure 2. All threads follow the same code, although the specifics for each node will vary. For example, Node A in the example does not wait for any dependent nodes in Step 1. For this part of

the project Step 2 is simply looking up the fixed value of the node. Step 3 is accomplished with the Linux library call *sleep()*. In Step 4 all nodes accumulate their computed value in a shared (amongst all nodes) global variable. This shared variable is initialized to zero. Because the variable is shared, access to it must be protected so that only one thread can do so at a time. You will need to create a semaphore or lock to protect access. Once all (if any) dependent nodes are signaled in Step 5 then the thread should terminate execution.

```
1. Wait for completion of dependent nodes (using semaphores)
2. Compute value (for now just look up fixed value)
3. Sleep for computation time.
4. Accumulate computed value in shared global variable.
5. Signal completion for all dependent nodes (using semaphores)
```

Figure 2: Compute Thread Pseudocode

Your main program is responsible for reading the configuration file and creating a table of node information as well as creating threads and synchronization primitives.

Your main program should initially determine the current time (in seconds) using the Unix library routine *time()*, found in Section 3 of the man pages, which maintains a running count of the time in seconds since January 1, 1970. Your program should record the current time in a global variable when it starts execution so that all time is reported relative to the beginning of your program execution.

Basic Objective

The name of your executable should be *graph* and it should take a single command-line argument, which is the name of the configuration file that your program needs to read.

After your main program has created all of the semaphores and computation threads it waits for all of the threads to terminate using *pthread_join()*. Just before termination, each thread should print its completion time in seconds (relative to the start of the program). After all threads have completed, the main program should print the result of the distributed computation (value of the shared variable) and the total time to complete the computation (relative to the start of the program). Before termination, your main program should destroy all semaphores that it created.

As a starting point it is suggested that you begin your testing with a configuration file containing a single node with no dependencies and then expand to be able to handle more nodes. For example, a simple configuration file would be:

```
A 2 1
```

Sample Execution

Using the configuration for the sample distributed computation in Figure 1, here is a sample execution of the program where the configuration is stored in the file `config.txt`. The order of your node completions may vary.

```
% ./graph config.txt
Node A computed a value of 1 after 1 second.
Node B computed a value of 2 after 2 seconds.
Node C computed a value of 3 after 2 seconds.
Node D computed a value of 4 after 3 seconds.
Total computation resulted in a value of 10 after 3 seconds.
```

Additional Work for Expression Evaluation

Satisfactory completion of the basic objective of this assignment is worth 25 of the 30 points.

For three additional points, your computation nodes will literally compute a value rather than use a fixed value. Once the value is computed it is added to the shared global variable in Step 4 of Figure 2. For this part of the project, each node will continue to add a fixed value to the shared global variable unless the there is an "=" character as a field after the list of dependent nodes (in this case the integer value in the second field is ignored). What follows the equal sign is the specification of a computation using reverse Polish notation (RPN) for the node to compute as its value. RPN (also called postfix notation) specifies operands before operators. You need to implement a stack-based calculator supporting the five *integer-only* operators "+", "-", "*", "/" and "%". All operands will be integers. The "/" operator does integer division and the "%" does modulo arithmetic.

Each operand works by popping off a first and second value from the stack. The second value popped off the stack is the first argument to the operand. For example, "5 3 -" results in a value of 2. The resulting value from an operand is pushed back onto the top of the stack.

The value on the top of the stack at the end of expression execution is the value of the computation and should be added to the shared global variable as shown in Step 4 of Figure 2. If there is an error in the expression (e.g. too few operands on the stack) then an error message should be reported by the thread and a value of zero should be added to the shared global variable.

The following is a new configuration file (`newconfig.txt`) making use of these operators plus two special operands:

1. I: numeric identifier of the node where Node A has identifier of zero, Node B has identifier of 1 and so on.
2. V: current value of the shared global variable. Retrieval needs to be protected as another thread may be attempting to change the variable at the same time.

```
A 1 1
B 0 1 A = 23 2 4 * +
C 0 1 A = V 2 %
D 0 1 B C = I V +
```

Each thread should compute the value of the RPN expression in Step 2 of Figure 2 and not pre-compute the value as could be done for the expression of Node B. The expression for Node C results in a value of zero if the shared variable is even and a value of one if the shared variable is odd. The Node D RPN expression adds the numeric identifier and the retrieved value. Note that this computation has potential non-determinism depending on the relative order of execution of Nodes B and C. Execution of your program (assuming Node B executes before C) will look like:

```
% ./graph newconfig.txt
Node A computed a value of 1 after 1 second.
Node B computed a value of 31 after 2 seconds.
Node C computed a value of 0 after 2 seconds.
Node D computed a value of 35 after 3 seconds.
Total computation resulted in a value of 67 after 3 seconds.
```

Additional Work for NBlock Primitive

For final two points, you need to implement an "nblock" primitive. For this project, an nblock primitive alleviates the need to use a semaphore for each edge of the computation graph. Instead this primitive causes a thread to block until it is signaled n number of times. You need to implement four routines (could also be implemented as a C++ object):

1. *id = CreateNBlock(n)* - Create an nblock primitive that requires it be signaled n times before proceeding.
2. *SignalNBlock(id)* - Signal the given nblock primitive.
3. *WaitNBlock(id)* - Wait on the given nblock primitive until it has been signaled the specified number of times.
4. *DestroyNBlock(id)* - Destroy other synchronization primitives used in creating the nblock primitive.

You may find it useful to maintain a counter of the number of threads that have signaled an nblock primitive, but be careful in your implementation as it is possible to introduce race conditions. Once available, you should create a new version of your project code that uses nblocks. Your code should create one nblock primitive for each node with the given number of dependent nodes as the value of n. In the example of Figure 1, Node A would have an nblock with count zero, Nodes B and C nblocks with counts of one and Node D an nblock

with count of two. If you implement this portion of the project then you should submit two versions of your project—your original version as well as the nblock version. The nblock version should create an executable called *nblock*.

Make

A useful program for maintaining large programs that have been broken into many software modules is *make*. The program uses a file, usually named `Makefile`, that resides in the same directory as the source code. This file describes how to "make" a number of targets specified. The following is a portion of the `Makefile` linked on the course Web page. Typing "`make graph`" causes the executable file *graph* to be created from `graph.C`.

You should copy the sample `Makefile` and modify it for your project. Note the sample `Makefile` is used to compile a C++ version of the sample program. You will need to modify if you are using C.

As a matter of explanation the first three lines below simply define and give values to *make* variables. The remaining text describes how to make the target "graph". WARNING: The operation line(s) for a target MUST begin with a TAB (do not use spaces). See the man page for *make*, *g++* and *gcc* for additional information.

```
LIB=-lpthread
CC=g++

graph: graph.C
        $(PP) graph.C -o graph $(LIB)
```

Submission of Project

Use the web turnin to turn in your project using the assignment name "proj3". You should have a source file for *graph* and have a Makefile that compiles it into an executable with this name. If you do the last part of the project then you should also have a source file for *nblock* and add a dependency to your `Makefile` that compiles it. You must also turnin a script showing test execution of your program(s) for various compute configurations.