CS39002 Operating Systems Laboratory
Spring 2024

Lab Assignment:5
Date of submission: 07–Feb –2024

Mutual exclusion and synchronization using semaphores

Mr. Fooman is a top-level boss in a company, having a team of n workers under him. In order to execute a project, he calls his team, and hands over the duty chart to the workers. Each worker is assigned a specific module in the project. The duty chart specifies precedence constraints among the modules, that is, which modules are to be finished before which modules. Assume that there are no cyclic dependencies. As soon as some module is finished, the corresponding worker in charge will report to Fooman. Fooman will finally check whether the modules are carried out according to the precedence plan.

We can visualize Fooman's problem as making a topological listing of the vertices in a directed acyclic graph (DAG) $G = (V, E)$. Each node in G stands for a module (synonymously with a worker), so $|V| = n$. If Module i supplies some prerequisites for Module j, there exists a directed edge (i, j) in E. The graph contains no other types of edges.

In this assignment, you are required to make a multi-process implementation of topological sorting. However, you are *not* supposed to implement any DFS-based algorithm. There will be one process B representing the boss, and n worker processes $W_0, W_1, W_2, \ldots, W_{n-1}$. Unlike in the previous assignment, B does not fork the worker processes. You run B in one terminal. In another terminal, you run the n worker processes concurrently in the background. The different components of this assignment are now elaborated.

The DAG generator

A C program *gengraph.c* is supplied to you. Each run of this program generates a random DAG on n nodes with each (forward) edge present with probability p. The default value of n is 16, and the default value of p is 0.2. You can supply p and n as optional command-line arguments. Redirect the output to a text file *graph.txt* which stores n followed by the $n \times n$ adjacency matrix of the graph.

```
$ gcc -Wall -o gengraph gengraph.c
$ ./gengraph > graph.txt
$ ./gengraph 100 > graph.txt
$ ./gengraph 50 0.1 > graph.txt
```

The boss process *B*

B reads *graph.txt*, and stores the adjacency matrix in a shared memory A with n^2 int entries. B also creates a shared array T of n int entries, for storing the topological listing. Finally, the index in T, call this *idx*, where each worker will write is also created as a shared memory (one int only). The count n of workers is available in the input file *graph.txt*, and does not need to be saved (as a read-only int value) in the shared memory.

B also creates some semaphore sets. Do **not** use pipes or any other synchronization mechanism. The adjacency matrix A is initially populated by B, and later accessed by the workers (and by B too at the very end) in the read-only mode, so accessing A does **not** call for mutual exclusion.

However, *T* and *idx* are shared items. A single semaphore *mtx* is needed for the mutual exclusion of the worker processes trying to update the listing simultaneously. For example, multiple nodes of in-degree 0 may try to write concurrently (or in parallel) to *T* at *idx* = 0. The semaphore *mtx* will guard against race conditions. Use a single semaphore to guard both *T* and *idx*.

For synchronizing the workers as per the precedence plan, a set *sync* of *n* semaphores is needed, one for each worker. (Do **not** create a semaphore for each edge, because there may be too many edges.) The purpose of *sync*[*i*] is to block the worker W_i until all W_j with links (*j*, *i*) write to *T*.

Finally, note that *B* does *not* fork the workers, so it cannot *wait*() for them to terminate. *B* also does not want to do busy waiting (by polling *idx* in a loop), so another semaphore *ntfy* is needed to notify the boss about the end-of-work from each worker process.

B initializes all these semaphores by appropriate values.

After creating and initializing the above shared-memory segments and semaphore sets, *B* waits until it receives notifications from all the workers. After this, *B* prints *T*, and checks the contents of *T* to figure out whether *T* stores a valid topological listing (that is, whether your code worked correctly). It prints an appropriate message (like well done, or these many precedence constrains are violated).

Finally, *B* removes all IPC resources it has created, and exits.

Run a C program *boss.c* to do the task of the boss.

The worker processes

Each worker process W_i (there are *n* of them) should be supplied two command-line arguments: *n*, and the ID *i* in $\{0, 1, 2, \ldots, n - 1\}$ of the node it stands for. Its only task is to append its ID *i* at the index *idx* of *T*, and to update (increment) *idx*. It should attach to the three shared-memory segments created by the boss. Moreover, it should also use the semaphore sets created by the boss. The worker processes are not forked by *B*, so *ftok*() should be used to agree upon the shared-memory and semaphore-set keys. Assume that each worker process starts running strictly after *B* creates all of these resources.

Each W_i waits for the *sync* signals from all incoming links (*j*, *i*) (if any exists). It then writes *i* at the index *idx* of *T*, and increments *idx*. This writing is the critical section for the worker processes, and must be guarded by *mtx*. After this write finishes, W_i sends *sync* signals to all outgoing links (*i*, *k*) (if any exists). Finally, W_i notifies *B* (using *ntfy*) that it is done with the update of *T*, and terminates.

Write a C program *worker.c* to do the above task of a worker. This program is not required to print anything to *stdout*. However, you may print a few diagnostic messages if that comforts you.

Some other files

Compile *boss.c* to the executable *boss*. Also compile *worker.c* to the executable file *worker*. A makefile is supplied to you to do all the compilations. It is painful to run *n* worker processes manually from a terminal. A bash script (*dowork*) is also supplied to you to automate the process. Give execute permission to this script, and call it to launch all the worker processes. This script assumes that the DAG is available as *graph.txt*.

Sample Output

Do this is one shell.

```
$ make
gcc -Wall -o gengraph gengraph.c
gcc -Wall -o boss boss.c
gcc -Wall -o worker worker.c
$ ./gengraph > graph.txt
$ cat graph.txt
16
0 0 1 0 0 0 0 0 0 1 0 0 0 0 0 0
0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 1 1 0 1 0 0 1 0
0 0 0 0 0 0 1 0 1 0 1 0 1 0 0 0
0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 1 0 0 0 0 0 0 1 0 0 0 0 0 1 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 1 0 1 0 0 0 0 0 1 0 0 0
0 1 0 0 0 0 0 0 1 0 0 0 1 0 0 0
1 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0
$ ./boss
+++ Boss: Setup done...
```

The boss waits at this point. Run the shell script `./dowork` in another shell. When all the worker processes finish, the boss prints the following, and terminates.

```
+++ Topological sorting of the vertices
5   3   10  13  4   7   6   15  0   2   9   11  14  1   12  8
+++ Boss: Well done, my team...
```

Use `make clean` to delete all the binary files. Submit the entire directory as a single zip archive.