CS39002 Operating Systems Laboratory
Spring 2025

Lab Assignment: 1
Date: 08–Jan –2025

Multi-process applications

FooBar Inc. manufactures fooware utilities. These utilities consist of n fooware modules (called foodules). Call them foo1, foo2, foo3, . . . , foo$n$. Each foodule is managed by a separate group. The foodules have dependencies among them. In order to rebuild a foodule, all its dependencies need to be rebuilt. Each of these dependencies has its own dependencies that also need to be rebuilt, and so on. Assume that there are no cyclic dependencies, that is, the dependencies can be represented as a DAG. The situation is similar to make except that the rebuilding happens in the reverse direction. In other words, when a component changes, make rebuilds that component and all other components that directly or indirectly depend on that component. On the contrary, when a foodule needs to be rebuilt, that foodule and all other foodules on which that foodule directly or indirectly depends need to be rebuilt.

The FooBar dependency graph is supplied to you as a text file *foodep.txt*. The first line contains the number n of foodules. This is followed by n lines storing the dependencies of the n foodules. Each dependency line is of the following format.

u: v_1 v_2 . . . v_k

This means that foo$u$ depends on foov_1, foov_2, . . . , and foov_k. A code *gendep.c* will be supplied to you. Compile and run it with n as the command-line argument, to get a random DAG on n nodes. Without the command-line argument, the program takes $n = 10$.

The FooBar testing team checks for problems with foodules. Any problematic foodule needs to be rebuilt. Suppose that Foodule u needs to be rebuilt. The group handling that foodule is contacted. Upon receiving the request, the group for Foodule u checks the dependencies; let these be v_1, v_2, . . . , v_k. The group for each v_i is then contacted with rebuilding requests. This procedure continues recursively. Because the dependency graph is a DAG, each dependency v_i of u will be eventually rebuilt. Foodule u is then rebuilt.

In this assignment, you emulate the Foobar Inc.'s work flow by a set of processes. You need to write a program *rebuild.c*(*pp*), and compile it to the binary *rebuild*. A call of *rebuild u* initiates the rebuilding procedure. The process launched by the call emulates the working of the group handling the u-th foodule. Whenever needed, it spawns child processes to emulate the working of the groups handling the foodules v_1, v_2, . . . , v_k. The process for each v_i checks its dependencies, and spawns further processes as needed. Because the dependency graph is a DAG, this creation of new processes will eventually stop.

In this context, note the following.

1. All the dependencies of a foodule have to be rebuilt *before* that foodule is rebuilt.

2. No foodule is rebuilt multiple times. For example, if Foodule u depends on Foodules v and w, and Foodule w also depends on Foodule v, then Foodule v is rebuilt only once, and is used for the rebuilding of both Foodule u and Foodule w.

In essence, you run a recursive DFS traversal in the dependency graph starting at the node *u*. However, each recursive call is replaced by *fork*ing a child process that *exec*'s the same code *rebuild* with a new foodule number that is passed as a command-line argument by the parent process. The parent *wait*'s until the child *exit*'s, and then *fork*'s another child (if needed).

A DFS traversal needs a *visited* array so that a node is not visited multiple times. This array needs to be read and modified by multiple processes. At this moment, you do not know sophisticated inter-process communication mechanisms. Use a file *done.txt* to store the *visited* array. The topmost process launched by your shell command initializes the array to all 0's. When a foodule is rebuilt, the corresponding entry is set to 1. When *rebuild* is called by a parent process, the child must know that the initialization of *visited* is not warranted. This information can be passed from the parent to the child by a second command-line argument (the first argument is the foodule number). What exactly you pass as this second argument does not matter. If *rebuild* sees that it is called with a single command-line argument, it understands that it is the root process, so the onus of initializing *visited* is on it. On the other hand, if *rebuild* is called with two command-line arguments, it skips the initialization of *visited* because it knows that it is not the root process.

Every time a decision is taken to *fork* a child, the *visited* array is to be read from the file *done.txt*, because recursive rebuilding of dependencies may make invalid the old *visited* array read by the parent earlier.

To sum up, the working of *rebuild u* is as follows.

Read *n* and the dependencies of *u* from *foodep.txt* (<u>*do not*</u> store the entire graph).
If this is the root process, initialize the *visited* array to all 0's in the file *done.txt*.
For each dependency *v* of *u*, do:
    Read the *visited* array from *done.txt*.
    If *v* is not yet rebuilt, do:
        Fork a child process, and wait until the child process exits.
        The child process execs *rebuild* with *v* as the first command-line argument.
Read the *visited* array from *done.txt*.
Set *visited*[*u*] = 1.
Write the updated *visited* array back to *done.txt*.

Submit a single file *rebuild.c*(*pp*).

Sample

Input	Output

$ gcc -Wall -o gendep gendep.c
$ ./gendep
$ cat foodep.txt
16
1: 2
2: 15
3: 2 7 10 11 12 13 15 16
4: 15
5: 2 7 8 11 13 15
6: 2 4 7 12 13 15
7: 2
8: 7 14
9: 1 3 4 8 10 12 16
10: 14
11: 1 2 12 13
12:
13: 1 14
14: 1 2
15:
16:
$
	$ gcc -Wall -o rebuild rebuild.c
	$ ./rebuild 9
	foo15 rebuilt
	foo2 rebuilt from foo15
	foo1 rebuilt from foo2
	foo7 rebuilt from foo2
	foo14 rebuilt from foo1, foo2
	foo10 rebuilt from foo14
	foo12 rebuilt
	foo13 rebuilt from foo1, foo14
	foo11 rebuilt from foo1, foo2, foo12, foo13
	foo16 rebuilt
	foo3 rebuilt from foo2, foo7, foo10, foo11, foo12, foo13, foo15, foo16
	foo4 rebuilt from foo15
	foo8 rebuilt from foo7, foo14
	foo9 rebuilt from foo1, foo3, foo4, foo8, foo10, foo12, foo16
	$ cat done.txt
	1111001111111111
	$ ./rebuild 10
	foo15 rebuilt
	foo2 rebuilt from foo15
	foo1 rebuilt from foo2
	foo14 rebuilt from foo1, foo2
	foo10 rebuilt from foo14
	$ cat done.txt
	1100000001000110
	$ ./rebuild 11
	foo15 rebuilt
	foo2 rebuilt from foo15
	foo1 rebuilt from foo2
	foo12 rebuilt
	foo14 rebuilt from foo1, foo2
	foo13 rebuilt from foo1, foo14
	foo11 rebuilt from foo1, foo2, foo12, foo13
	$ cat done.txt
	1100000000111110
	$ ./rebuild 12
	foo12 rebuilt
	$ cat done.txt
	0000000000010000
	$