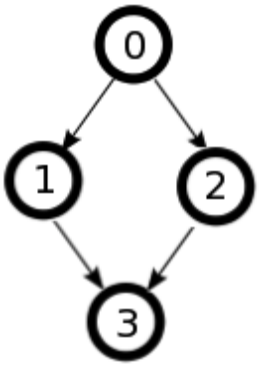
CSCI 4061: Intro. to Operating Systems
Assignment 1: Graph Execution

Due: October 7th, 2015, 11:59 pm. Please work in groups of 2.

Purpose:

To develop a `graphexec` program in C that will execute a graph of user programs in parallel using `fork`, `exec` and `wait` in a controlled fashion. Such graphs are used in compiler analysis and parallel program execution to model control and data dependencies. A control dependency specifies that a program cannot start until its predecessor(s) is finished. A data dependency specifies that a program requires input from its predecessor(s) before it can execute.

Description:



Your main program `graphexec` will be responsible for analyzing a graph of user programs, determining which ones are eligible to run, and running programs that are eligible to run at the moment in time. As programs in the graph finish, your program will determine which other programs in the graph have become eligible to run, execute those programs, and continue this process until all programs are finished. In the example graph to the left, node 0 can be executed first because it's not a child of any other node. After node 0 finishes executing, then nodes 1 and 2 can be executed in parallel (they both should be started immediately without waiting for either 1 or 2 to finish). Only after both 1 and 2 finish can the final node 3 be executed. Note that this is not a *process* tree – your main program can spawn each node as long as nodes run in the order specified by the dependencies.

Each node in the graph represents one program to be executed. Each node will contain:
- The program name with its arguments
- "Pointers" to child nodes
- The input file to be used as standard input for that program
- The output file to be used as standard output for that program

A node becomes eligible for execution once all of its parent nodes (nodes that contain a pointer to this node) have completed their own execution. Your main program will `fork` and `exec` each of the nodes as they become eligible to run. Also, input and output redirection must be used so that each node can get its standard input from a file and also write its output to a file (respectively). This is given to you in a file named `redirect.c`.

Graph File Format:

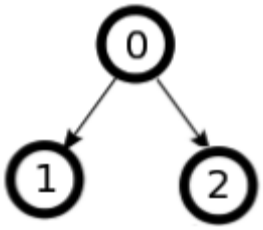
We will run your `graphexec` program as follows:

```
$ ./graphexec some-graph-file.txt
```

A text file will represent the structure of the graph. We have included most of the test cases we will run with your program. Each node will be represented by a line in the following format:

```
program name with arguments:list of children IDs:input file:output file

ls -l:1 2:blank-file.txt:ls-output.txt
cat myfile1 file2:none:blank-file.txt:cat-out.txt
gzip:none:ls-output.txt:ls-output.txt.gz
```



In this example, node 0 (`ls -l`) runs first and outputs `ls-output.txt`; when this completes, both node 1 (`cat`) and node 2 (`gzip`) run simultaneously. Node 1 requires `myfile1` and `file2` to exist and outputs `cat-out.txt`; node 2 requires `ls-output` to exist and zips that into `ls-output.txt.gz`.

Note: If there are no children for a node, it must be specified as "none". The input and output files must be specified; if there is no input, specify a blank text file. This means that your program should just open each file and redirect the file descriptors as long as the files can be opened. For convenience, the nodes will implicitly be numbered from 0 to (n-1) from the order the nodes appear in the text file (where n = total number of nodes in the graph). The children IDs will correspond to this numbering system. You may not assume that the nodes are in execution order from top to bottom based on their ID number. The ID numbers are used mainly to create "pointers" from parent nodes to children nodes.
Note: This means that the first node to be executed isn't necessarily the first node in the file. You may not assume there is only one root node.

Node Structure:

We highly suggest you use this code to help you with the node structure:

```
//for 'status' variable:
#define INELIGIBLE 0
#define READY 1
#define RUNNING 2
#define FINISHED 3
typedef struct node {
      int id; // corresponds to line number in graph text file
      char prog[1024]; // prog + arguments
      char input[1024]; // filename
      char output[1024]; // filename
      int children[10]; // children IDs
      in num_children; // how many children this node has
      int status;
      pid_t pid; // track it when it's running
} node_t;
```

High-level View:

Your main program will first parse the graph file given in the first argument, construct a data structure that models the graph, and then start executing the nodes. Your program should determine which nodes are eligible to run, execute those nodes, `wait` for any node to finish, then repeat this process until all nodes have finished executing.

Useful System Calls & Functions:

It is highly suggested that you make use of the following system calls or library functions:
`open, fopen, fgets, fork, dup2, execvp, wait, strtok, makeargv` - from R&R page 37 - VERY USEFUL; you may copy and cite the code from the text.

Hints:

- It may be useful to implement a `determine_eligible` function in your main program which marks the nodes that are ready to run.
- You should make heavy use of the `makeargv` function when parsing the graph file.

Simplifying Assumptions:

- There will be no more than 50 nodes in any given graph.
- No node will have more than 10 children.
- The graph file will not contain extra whitespace "padding" between the colons and values.
- The graph file will contain single spaces between the children ID numbers for that parameter.
- Each line in the graph file will not exceed 1023 characters.
- There are no cycles in the graph.

Error Handling:

You are expected to check the return value of all system calls that you use in your program to check for error conditions. Also, your main program should check to make sure the proper number of arguments is used when it is executed. If your program encounters an error, a useful error message should be printed to the screen. Your program should be robust; it should try to recover if possible. If the error prevents your program from functioning, then it should exit after printing the error message. (The use of the `perror` function for printing error messages is encouraged.)

Documentation:

You must include a README file, named "README" without extensions, which describes your program. It needs to contain the following:

- The purpose of your program
- How to compile the program

- How to use the program from the shell (syntax)
- What exactly your program does
- Your x500 and the x500 of your partner

The README file does not have to be very long, as long as it properly describes the above points. Proper in this case means that a first-time user will be able to answer the above questions without any confusion. Within your code you should use one or two sentences to describe each function that you write. You do not need to comment every line of your code. However, you might want to comment portions of your code to increase readability.

At the top of your README file and main C source file please include the following comment:

```
/*login: itlabs_login_name
* date: mm/dd/yy
* name: full_name1, full_name2
* id: id_for _first_name, id_for_second_name */
```

Grading:

5% README file
15% Documentation within code, coding, and style (indentations, readability of code, use of defined constants rather than numbers)
80% Test cases (correctness, error handling, meeting the specifications)

- Please make sure to pay attention to documentation and coding style. A perfectly working program will not receive full credit if it is undocumented and very difficult to read. It will not receive full points if your README lacks the correct name ("README") or does not include your (and your partner's) x500.
- You will be given most test cases up front. You are also encouraged to make up your own tests. If there is anything that is not clear to you, you should ask for clarification.
- We will use the GCC version installed on the CSELabs machines to compile your code. Make sure your code compiles and runs on the CSELabs machines.

Deliverables:

- Files containing your code (in a single .zip or .tar or .tar.gz file)
- A README file
- A makefile that will compile your code and produce a program called `graphexec`.
- Do NOT include the executable

Note: this makefile will be used by us to compile your program with the make utility. If your provided makefile doesn't work then you will not receive full credit for the test cases. See the references page on the course site for links on creating makefiles; a sample is also given to you in this assignment. All files should be submitted using Moodle. This is your official submission that we will grade. Please note that future submission under the same assignment title OVERWRITES previous submissions; we can only grade the most recent submission. Make sure you have tested the code you upload to Moodle.

Extra Credit:

For up to 10 additional points, your `graphexec` program should determine if cycles exist in the graph, print an error message (such as "Cycle found in graph.") and exit gracefully if any cycle is found. In the example to the left, node 0 lists its child as node 1, and node 1 lists its child as node 0. Your program should detect the cycle that begins again when node 1 points to node 0 and exit.