Reading Material for Processes

As Part of

Operating Systems [CS F372] Course

Semester I, 2020 – 2021

BITS Pilani
K K Birla Goa Campus

In case of any doubt, please contact: [Reach out to us only after putting in a good amount of effort; we can help you with the logic but NOT with the code!!]

PRABHU DIVYA UMESH [2017A7PS0001G]
VINAYAK AGGARWAL [2017A7PS0008G]
ISHAN SANG [2017A7PS0069G]

INDEX

1. PREREQUISITES

★ Process Basics [section ‘A fork() Primer’ of R1]
- fork()
- wait(), waitpid(), exit()
- exec family
- getpid(), getppid()

★ Signal Handling [section ‘Signals’ of R1]
- signal(), kill()
- user-defined signals
- signal handlers

★ Message Passing [section ‘Pipes’ of R1]
- dup(), dup2()
- pipe()

★ Shared Memory [section ‘Shared Memory Segments’ of R1]
- shmget()
- shmat(), shmdt()

★ File Operations
- fopen(), fclose(), fseek(), fscanf(), fprintf()

★ Miscellaneous Commands
- apropos/man -k [can be used to find relevant commands/ C functions]
- ps, pgrep [can be used for debugging purposes]

***R1**: “Beej's Guide to Unix IPC” [You will be able to find it online!!]

NOTE:

man pages (usually available on all Unix-based systems) are excellent resources themselves; try to use them before you search for something on the Internet.

2. WARMUP PROBLEM

Problem Statement:

Write a program to create a process tree as shown in Figure 1 (if the command line argument is 2, 3) and print the process identifiers in reverse order of its creation. The number of processes parent process P has to create is the first command line argument and the number of levels in the tree is the second argument. The program should create simultaneously executing processes in the same level as quickly as possible before child's execution starts. Each process should print its process identifier and its parent process identifier by using the function getpid( ) and getppid( ) respectively. Your program should not create an orphan process while execution. The parent process should also print the exit status of its child processes.

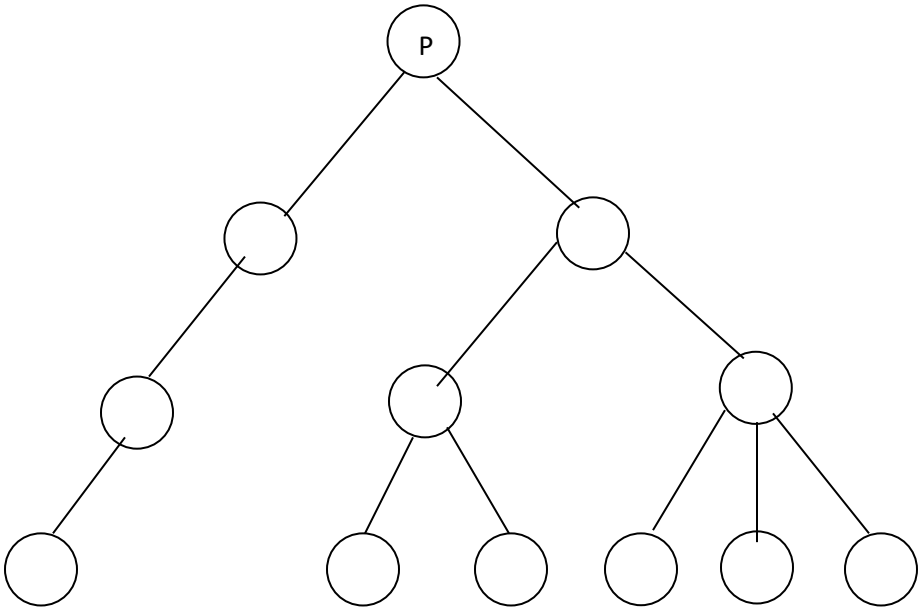


Figure 1 : Process Tree for input 2,3

3. SHORTCUT TO HIGHER MARKS

Problem Statement:

Since you are going to pursue a major in computers, you must be aware of how hacking is done (for your own benefit). We present to you a very sneaky way into "Some one's" user space where you can edit your own marks for your assignment.

Aim: Change your Online #1 marks in "Some one's" user space at your own will without any penalty.

Input Format:

The input must be taken as follows

<filename> <ID No.> <New marks>

This input must be taken as arguments to the main function.

Format of the Marks file:

<ID No.> <\t> <Marks>

Assumptions:

- You must have super user privileges at all times.
- Create two groups 1.) Evaluator, 2.) Student *(please refer to figure2 at the end for the exact hierarchy expected of the directories)*
- There will be two evaluators in the evaluator group.
- Individual evaluator will have its own directories and subdirectories. One of them will contain your marks file.
- You should write your code in the Student's directory and not in the Evaluator's.
- There may be a directory of the same name as that of the file you want to search.
- The code will start searching the marks file from the Evaluator directory, down the hierarchy.
- There will be no space in the name of any file or directory.
- You can use 'chdir' system call as 'cd' command does not work in exec(). You must use exec() series system calls for executing the rest of the commands like 'pwd', 'ls', 'grep', 'rm'.

Procedure to change:

- The code should execute the following command
  ls <options> | grep <options> expression

You are expected to fork two processes (one for ls and the second one for grep taking the input as output of ls command) that will execute the above command.

- After execution of this command, check whether you have found the given marks file.
- If you haven't found the file in the current working directory (cwd), the process should access the shared space to check whether the flag is set. The flag will be set only if some other process already found the file. So this process can exit without proceeding further. If the flag is not set, fork as many children as the number of directories in the cwd.
- It is possible that some of the directories in your cwd have read only permissions. If this is the case, you are expected to change these permissions to read –write – execute mode for group, user and others.
  This can be done using the chmod() system call. After changing the permissions, you will be allowed to traverse down the hierarchy.
- In each child process, change the cwd to the directory for which the child has been forked in the previous step.
- Now, each of the child processes must search for the file in their respective cwd simultaneously.
- At any point, if the file is found,
  - The process will access the shared memory to check if any other process has already found the file and there is a flag set to 1 in the shared space. If the flag is already set, the process should readily terminate without continuing further.
  - If the flag is not set in the shared space, the process will,
    - Execute pwd command to get the path.
    - Write this path in a shared memory.
    - Set the flag to 1.
- A process will terminate normally if it does not find any directory after the execution of the command stated in the first step.
- Till all the children finish their execution, the main process must wait (No orphans should exist in the system).
- After the termination of all child processes, the main process will access the shared memory to change your marks in the file whose path is present.
- First search your ID no. in the given file and then edit your marks corresponding to it. You may need the creation of a temporary file while the editing is being done.
- After your marks have been changed in the file, your main process should terminate.

- The following figure (Figure 2) gives the sample directory hierarchy for your reference.
- The marks file can be in any of the evaluator's directory.
- The figure below shows just a sample example for your understanding and not any concrete directory structure. So, please do not hardcode with reference to the figure below. Your code is expected to be generic.

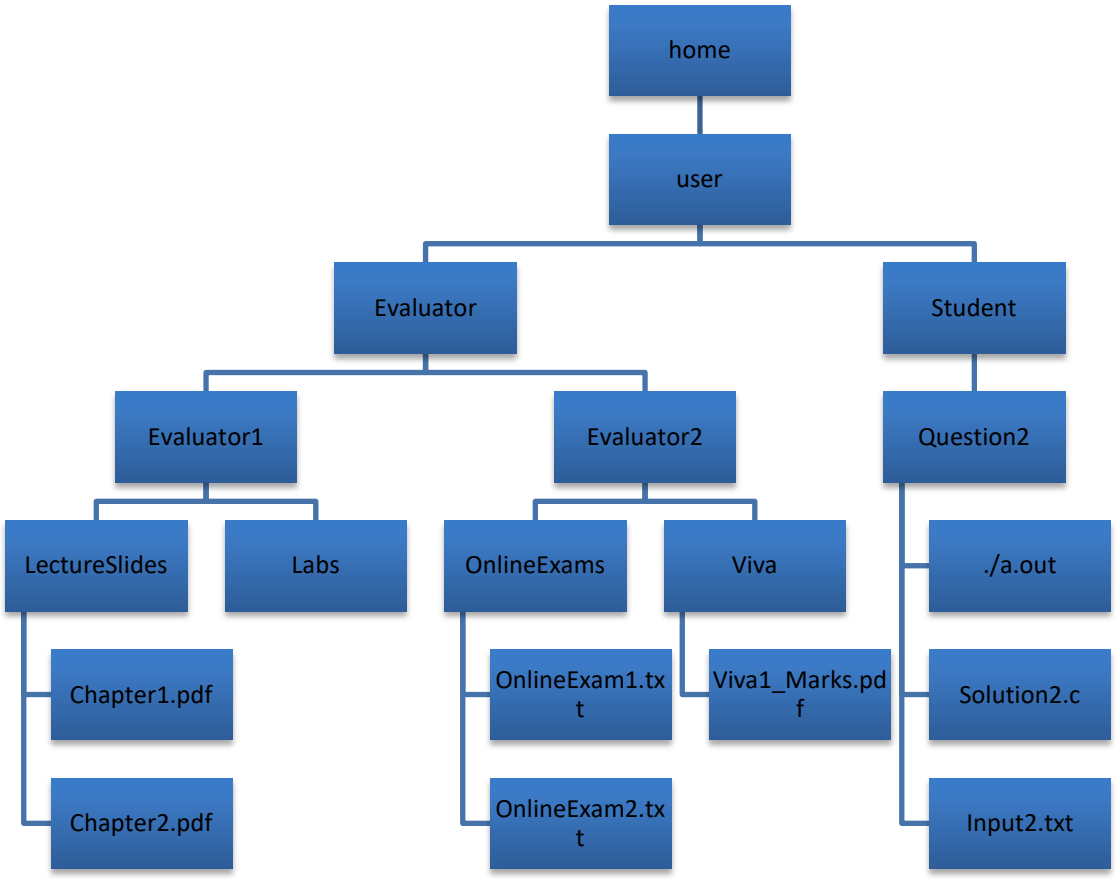


Figure 2 : Sample directory hierarchy with groups evaluator and student

Your code must also delete all the temporary files that have been created in the above procedure. Please be realistic while giving yourself marks for Test 1 so that "no user" will get suspicious.

If you have taken help from anyone, please be fair enough. Please remove all unnecessary printfs, so that "no user" will get suspicious and you will end up safely with your marks.

4. AMUSEMENT PARK

Problem Statement:

You are the owner of an amusement park. There is a Jumbo patrol van that takes rounds across the park. So that adding extra security doesn't cost you too much, you tried to put this van to other uses also. When the car goes around the park it also carries stuff from one place to another, but since there isn't always cargo to be transported they could also take a few people for a tour.

So for this, you have 3 people to manage this (Ticket Guy, Queue Maintainer, Driver). Since there is a fixed point where they pick & drop people, the ticket guy sends the people to the Queue Maintainer (Standing at pickup/drop point). But he needs to first verify if the queue has enough space to take more people. He sends a message (request) containing the number of people waiting at the ticket counter (new + who were already waiting at the ticket counter) whenever someone books tickets and receives a message (response) from the Queue-Maintainer how many he can send so the queue doesn't overflow. Other people wait at the ticket counter.

Whenever the driver reaches the pickup point, he sends a message containing the number of empty seats (excluding the ones occupied by the cargo) and the Queue maintainer let's that many people get into the van. If people in the queue are less than the seats, then some of the seats go empty in that round.

Queue size for Queue maintainer is 20. There is enough space to accommodate as many people at the ticket counter.

For the tour around the park simulate it using a random generator for 10-15 seconds.

For the ticket, guy simulate using a random generator for people coming in a group (1-6) at random intervals (5-20 seconds)

Simulate the empty seats in the van using a random generator (1-7)

Consider each of these 3 people as a different process.

Run the simulation for 100 seconds.

DO NOT USE SHARED-MEMORY / PIPE / FILES FOR MESSAGE PASSING IN THIS QUESTION. TRY USING ONLY SIGNALS. (Read about user-defined/real-time signals)

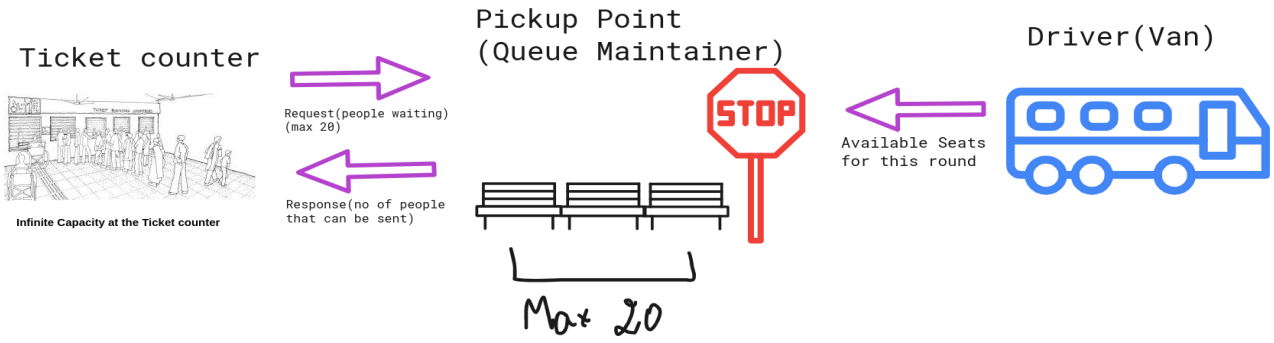


Figure 3 : Workflow in the Amusement Park

SAMPLE OUTPUT 1

5 people bought a ticket
5 people sent to ride queue
3 people bought a ticket
3 people sent to ride queue
4 people bought a ticket
4 people sent to ride queue
Ride Capacity: 7
Queue Size before ride: 12
Queue Size after ride left: 5
2 people bought a ticket
2 people sent to ride queue
Ride Capacity: 2
Queue Size before ride: 7
Queue Size after ride left: 5
5 people bought a ticket
5 people sent to ride queue
Ride Capacity: 1
Queue Size before ride: 10
Queue Size after ride left: 9
1 people bought a ticket
1 people sent to ride queue
Ride Capacity: 1
Queue Size before ride: 10
Queue Size after ride left: 9
1 people bought a ticket
1 people sent to ride queue
2 people bought a ticket
2 people sent to ride queue
Ride Capacity: 4
Queue Size before ride: 12
Queue Size after ride left: 8
Ride Capacity: 5
Queue Size before ride: 8
Queue Size after ride left: 3
5 people bought a ticket
5 people sent to ride queue
(....)

SAMPLE OUTPUT 2

3 people bought a ticket
3 people sent to ride queue
2 people bought a ticket
2 people sent to ride queue
6 people bought a ticket
6 people sent to ride queue
Ride Capacity: 6
Queue Size before ride: 11
Queue Size after ride left: 5
1 people bought a ticket
1 people sent to ride queue
4 people bought a ticket
4 people sent to ride queue
5 people bought a ticket
5 people sent to ride queue
Ride Capacity: 5
Queue Size before ride: 15
Queue Size after ride left: 10
2 people bought a ticket
2 people sent to ride queue
4 people bought a ticket
4 people sent to ride queue
6 people bought a ticket
4 people sent to ride queue
Ride Capacity: 6
Queue Size before ride: 20
Queue Size after ride left: 14
3 people bought a ticket
5 people sent to ride queue
1 people bought a ticket
1 people sent to ride queue
Ride Capacity: 3
Queue Size before ride: 20
Queue Size after ride left: 17
1 people bought a ticket
1 people sent to ride queue
6 people bought a ticket
2 people sent to ride queue

5. WORTHY DISHES

Problem Statement:

There are N chefs in a kitchen, numbered 1-N. The i^{th} chef takes time T_i to make a batch of V_i dishes. There is also a tree-like hierarchy in the kitchen. Each chef may have some non-negative number of sub-chefs working under him; Chef 1 is the supreme head chef (the root of the tree).

Whenever Chef 1 starts cooking, he calls **all the sub-chefs directly under him (in any sequence)** and asks them to make one batch of dishes each. After ordering all his sub-chefs, Chef 1 starts working on his own batch. These chefs in turn, call the sub-chefs directly under them to make one batch each in a similar fashion and then start working on their batch, and so on.

As soon as **any chef** (say, A) finishes making his batch of dishes, he starts collecting dishes from his sub-chefs (in the same order he called them) and adds the completed dishes in his own batch. Now if a **sub-chef of A** (say, B) hasn't finished making the dishes when A comes to them for collection, B will be asked to **stop cooking** and thus won't be able to provide any dishes to A. (i.e. a head chef won't wait for any sub-chef to complete a batch of dishes). Any chef given an order to stop cooking, **stops and then asks all his sub-chefs to stop** (i.e. if a chef is stopped, his whole subtree will be stopped recursively and no contribution will come from that subtree). Finally, **A will terminate and wait for his head** to come and collect all the dishes A has collected (i.e A's dishes + all the dishes A could procure from his sub-chefs).

As soon as Chef 1 terminates, the whole routine stops. A dish is **worthy** only if it is present in the collection of Chef 1 by the end of the routine.

Your task is to simulate this routine using processes (consider each chef as a process and the sub-chefs as child processes; assume stopping a chef is equivalent to killing that chef's process). Output for each i, the number of dishes $chef_i$ produced and the number of dishes $chef_i$ collected from his sub-chefs. Also, print the total number of worthy dishes gathered from all the chefs.

Input Format:

- First line contains a single positive integer, N
- N-1 lines of the form (chef id, sub-chef id)
- Next line contains N numbers where i^{th} number = size of batch of chef i (V_i)
- Next line contains N numbers where i^{th} number = time taken by chef i to complete his batch (T_i)

[Assume the whole input is present in 'input.txt']

Note 1: The i^{th} chef will either contribute a single batch of V_i dishes or won't contribute at all in the final collection of worthy dishes (i.e. a chef can't create a partial batch of dishes)
Note 2: Ensure that there are no orphan processes created during execution.
Note 3: Since each chef recursively stops it's sub-chefs, it may happen that a non-terminated sub-chef may complete his dishes before it gets the signal to stop - in that case, the sub-chef's batch of dishes will be considered to be complete, but it won't be contributed to the head chef.

Sample Input #1:

6

1 2

1 3

2 4

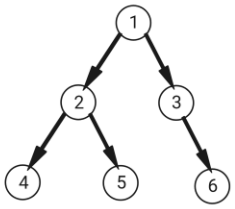
2 5

3 6

1 2 3 4 5 6

3 1 2 1 1 1

Sample Input Visualisation #1:



Sample Output #1:

Chef #1-> #dishes made: 1, #dishes collected: 11
Chef #2-> #dishes made: 2, #dishes collected: 0
Chef #3-> #dishes made: 3, #dishes collected: 6
Chef #4-> #dishes made: 0, #dishes collected: 0
Chef #5-> #dishes made: 0, #dishes collected: 0
Chef #6-> #dishes made: 6, #dishes collected: 0
Total Worthy Dishes: 12

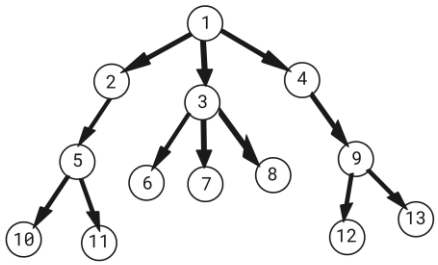
[**P.S.:** since we don't know how the processes will be scheduled in the CPU, there is no fixed answer - the following is also one of the acceptable answers for the given sample input]

Chef #1-> #dishes made: 1, #dishes collected: 16
Chef #2-> #dishes made: 2, #dishes collected: 5
Chef #3-> #dishes made: 3, #dishes collected: 6
Chef #4-> #dishes made: 0, #dishes collected: 0
Chef #5-> #dishes made: 5, #dishes collected: 0
Chef #6-> #dishes made: 6, #dishes collected: 0
Total Worthy Dishes: 17

Sample Input #2:

13
1 2
1 3
1 4
2 5
3 6
3 7
3 8
4 9
5 10
5 11
9 12
9 13
5 6 4 2 10 1 4 7 8 1 3 9 9
5 4 3 2 2 1 1 1 1 1 1 1 1

Sample Input Visualisation #2:



Sample Output #2:

Chef #1-> #dishes made: 5, #dishes collected: 55
Chef #2-> #dishes made: 6, #dishes collected: 14
Chef #3-> #dishes made: 4, #dishes collected: 12
Chef #4-> #dishes made: 2, #dishes collected: 17
Chef #5-> #dishes made: 10, #dishes collected: 4
Chef #6-> #dishes made: 1, #dishes collected: 0
Chef #7-> #dishes made: 4, #dishes collected: 0
Chef #8-> #dishes made: 7, #dishes collected: 0
Chef #9-> #dishes made: 8, #dishes collected: 9
Chef #10-> #dishes made: 1, #dishes collected: 0
Chef #11-> #dishes made: 3, #dishes collected: 0
Chef #12-> #dishes made: 9, #dishes collected: 0
Chef #13-> #dishes made: 0, #dishes collected: 0
Total Worthy Dishes: 60

6. PREMIER LEAGUE

Problem Statement:

It is the premier league season and you have to simulate the league to find out the champions. The league comprises N football teams (N<=10), numbered 1 to N. There are total N stadiums (indexed 1 to N) and each stadium has a stadium manager. The i^{th} stadium is the home ground of the i^{th} team.

Each team plays 2 games against every other team - One at its home ground and one away (home ground of opponent team).

If a team wins a match, it gets 3 points. If it's a draw, both teams get 1 point. If a team loses, they won't get any points.

At the end of the season, the team with the maximum points wins the premier league title. If some teams finish with the same number of points, their position in the League table is determined by the number of goals scored. If the teams have the same number of points and number of goals scored, then the team with lower team index gets a higher position.

You are provided the league fixture in an input file. Each line of input consists of the ids of the teams playing the match. (Refer input format) Your task is to schedule each match and notify the manager of the stadium on which the match is to be played. Each Match goes on for 3s. The goals scored by both teams can be generated randomly. Assume no team can score more than 5 goals in a match. At the end of the match, the stadium manager updates the score of the teams.

More than one match can be played simultaneously, but note that if team A is playing against team B then it cannot play against team C at the same time. If a match is ongoing at the stadium i, then a match cannot be scheduled at the same stadium at the same time.

Your task is to schedule and simulate the matches according to the fixture given. Consider each stadium manager as a process. All the stadium manager processes must be created before the matches start and each process must remain alive until all the matches to be played at the corresponding stadium are completed.

Display the final result of the league in the decreasing order of the points scored by each team (Refer the sample output)

Input Format:

First line contains N - Number of teams

Followed by N*(N-1) lines of the form i j, which denotes match Team i Vs Team j, played at the home ground of i^{th} team.

Sample Input #1:

```
3
1 2
2 3
3 1
2 1
3 2
1 3
```

Sample Input #2:

```
4
1 2
2 1
4 1
1 3
2 4
2 3
3 2
4 3
4 2
3 1
1 4
3 4
```

Sample Output #1:

```
Starting match: Team 1 vs Team 2
Match ended: Team 1 vs Team 2    Result:4-1
Starting match: Team 2 vs Team 3
Match ended: Team 2 vs Team 3    Result:4-0
Starting match: Team 3 vs Team 1
Match ended: Team 3 vs Team 1    Result:2-0
Starting match: Team 2 vs Team 1
Match ended: Team 2 vs Team 1    Result:4-2
Starting match: Team 3 vs Team 2
Match ended: Team 3 vs Team 2    Result:1-2
Starting match: Team 1 vs Team 3
Match ended: Team 1 vs Team 3    Result:0-4

Team  W  D  L  GS  Points
----------------------------------------------------
2     3  0  1  11  9
3     2  0  2  7   6
1     1  0  3  6   3
```

NOTE:

W- No of matches won

D- No of matches drawn

L- No of matches lost

GS- Total goals scored in the league

Sample Output #2:

```
Starting match: Team 1 vs Team 2
Match ended: Team 1 vs Team 2    Result:0-2
Starting match: Team 2 vs Team 1
Match ended: Team 2 vs Team 1    Result:2-2
Starting match: Team 4 vs Team 1
Match ended: Team 4 vs Team 1    Result:3-3
Starting match: Team 1 vs Team 3
Starting match: Team 2 vs Team 4
Match ended: Team 1 vs Team 3    Result:3-4
Match ended: Team 2 vs Team 4    Result:3-4
Starting match: Team 2 vs Team 3
Match ended: Team 2 vs Team 3    Result:0-0
Starting match: Team 3 vs Team 2
Match ended: Team 3 vs Team 2    Result:2-1
Starting match: Team 4 vs Team 3
Match ended: Team 4 vs Team 3    Result:4-2
Starting match: Team 4 vs Team 2
Starting match: Team 3 vs Team 1
Match ended: Team 4 vs Team 2    Result:0-0
Match ended: Team 3 vs Team 1    Result:0-0
Starting match: Team 1 vs Team 4
Match ended: Team 1 vs Team 4    Result:3-2
Starting match: Team 3 vs Team 4
Match ended: Team 3 vs Team 4    Result:3-3

Team  W  D  L  GS  Points
----------------------------------------------------
4     2  3  1  16  9
3     2  3  1  11  9
1     1  3  2  11  6
2     1  3  2  8   6
```

7. SPIDERMAN TO THE RESCUE!!

Problem Statement:

There is an NxN dimensional maze with each entry either 0 or 1.

Spiderman is at (0, 0). Mary Jane (MJ) is held up by the Green Goblin at ((N-1), (N-1)). Spiderman wants to rescue MJ as fast as possible. i.e.
Source: Top left corner of the matrix whose entry is 0. (Spiderman)
Destination: Bottom right corner of the matrix whose entry is 2. (MJ)

To get a quick way out, Spiderman asks his spies to help him find MJ. Each spy represents a process and Spiderman is the main process. Initially Spiderman who is at (0, 0), calls two spies which will check (0, 1) and (1, 0) entries. Depending on those entries, each will decide whether to call one or two other spies. If the spy is on the last row or column of the maze, it will call only one spy otherwise it will call two spies. This continues till a spy reaches MJ or no spy can find a way further.

Which means: Each process at (i, j) will have either one or two child processes which can find the possible path in the maze. These child processes read the entries that are to the right- (i, j+1) of and below- (i+1, j) the current entry- (i, j).

The right child in the tree (shown in Figure 5) corresponds to the child reading the entry to the right of the current process whereas the left child in the tree (shown in Figure 5) corresponds to the child reading the entry below the current entry.

In the maze, the path is created by 0's whereas 1's are the obstructions and 2 is the final destination where MJ is. So, if you have a 0 as the entry for the current process, you will have a path further from that node by creating either one or two child processes. If the entry for the current process is 1, you cannot have a path from that process and it dies.

The input will be such that there is only one possible path from the source to the destination.

You have to print the path in reverse order where each process that participated in the actual path must print the corresponding entry as an (i, j) pair.

The following is the input matrix.

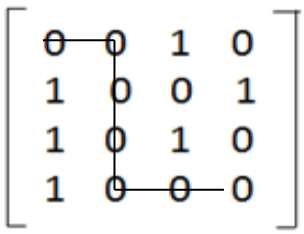


Figure 4 : Input Matrix

The corresponding process tree for the above given input matrix will be:

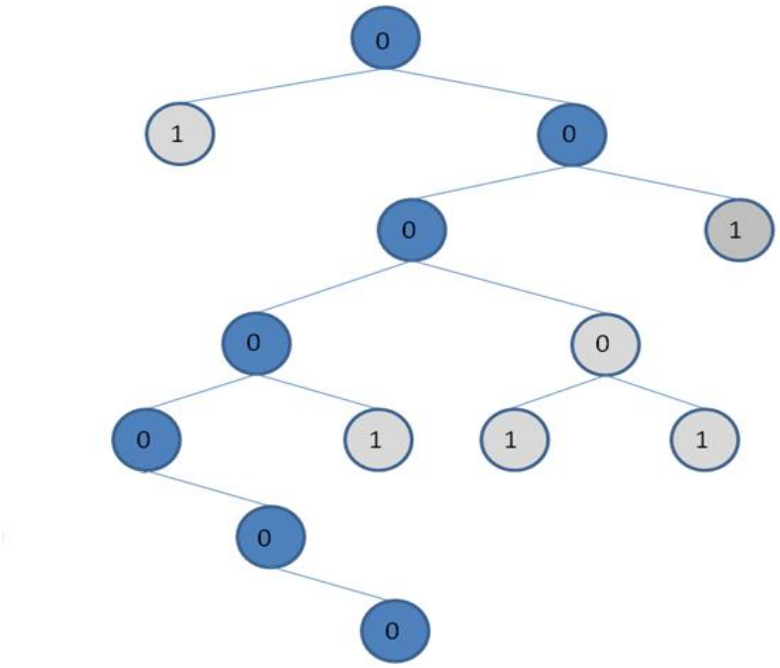


Figure 5 : The process tree

Sample Input #1:

0	0	1	0
1	0	0	1
1	0	1	0
1	0	0	0

Sample Output #1:

(3,3) -> (3,2) -> (3,1) -> (2,1) -> (1,1) -> (0,1) -> (0,0)

MJ successfully rescued.

Sample Input #2:

0	0	1	0
1	0	0	1
1	0	1	0
1	1	0	0

Sample Output #2:

Spiderman cannot rescue MJ.

8. SHELL COMMAND LINE INTERPRETER

Problem Statement:

This program is to test your understanding in the following:

fork, signal, wait/waitpid, exit, exec, dup/dup2

In this question your task is to write a replacement for the shell command line interpreter. Typical command line shells under UNIX, which you may have used, are bash, sh, csh, tcsh. The goal of this assignment is to increase your understanding of processes and the role of system software.

Your shell provides a sample interface, prompting the user for commands, and then interpreting the supplied commands. It should have the following features, which constitute an informal specification for your program.

Input

The shell reads an input line at a time from the standard input until the end of line is reached. Each input line is treated as a sequence of commands separated by a pipe character (|). For example: ls –l | grep alpha | more

(There can be one or more white spaces separating the words – command name, options and pipe)

In this example there are three commands – the command ls -l, the command grep alpha, and the command more. Each command consists of words separated by one or more spaces. No special extra command editing facilities are necessary beyond what is already done by regular terminal input (example see man tty for the default editing features namely backspace, erase line, erase word and end of file).

Internal commands

Your program does not need to interpret any internal shell commands except the following ones

entry

The command line interpreter will start functioning with this command. Any external or internal command issued before the entry command should display “Command line interpreter not started” message.

exit

The command line interpreter exits. Any external or internal command issued after the exit command should display “Command line interpreter exited” message.

log

The command line interpreter starts the command and output logging into "command.log" and "output.log" files respectively with the log command.

unlog

The command line interpreter stops the command and output logging with the unlog command.

viewcmdlog

The viewcmdlog command displays the content of the command.log file.

viewoutlog

The viewoutlog command displays the content of output.log file.

changedir <argument>

The changedir command takes the new present working directory path as argument. If the path is valid then the current working directory will be changed to the new directory specified in the argument. Example: changedir /home/usr/bin

Directory changed. The present working directory is /home/usr/bin

External commands

All commands that are not internal commands are external commands. An external command is a sequence of words separated by white space(s) of which the first is the executable name of the command and the remainder is the arguments to the command. If no executable file was found at any of the environment paths, the shell must check the same in the current working directory. If the executable is still not found then the shell command line interpreter should print the error message: "The command <command name> not found". If the executable files are found, the shell should create a new process to run the programs. The child process must use the exec() family of system calls to run each program together with the supplied arguments and shell environment variables.

Processing of commands

The tricky thing about this assignment is that you have to arrange for the piping between processes for each command. For example in the example given before, the stdout of ls must be piped to the stdin of grep alpha. The stdout of grep alpha must be fed to the stdin of more.

Your program must work for arbitrary numbers of processes on the command line, with arbitrary number of parameters. You would normally use dup2 to achieve this.

Your shell should record all input commands by appending to a log file called "command.log". The format of "command.log" may be as shown below

<Command name> <Date> <Time of execution> <Action taken (success/failure)>

Your program should record the output of all commands, including the intermediate commands, by appending to a log file called "output.log". The format of the "output.log" file is such that each line of command output is associated with the command that generated it. For example the command ls | grep alpha | more should add the following entries in output.log file. (I would strongly recommend you to follow this format).

[ls]
<output of ls command>

[ls] | [grep alpha]
<output of the ls | grep alpha commands>

[ls] | [grep alpha] | [more]
<output of the ls | grep alpha | more commands>

Make sure no orphan processes are getting created while executing the commands.

Implement the shell with the help of dup2 and intermediate files.