The University of Queensland
School of Information Technology and Electrical Engineering
Semester Two, 2013
CSSE2310 / CSSE7231 - Assignment 3
Due: 11:10pm 27 September, 2012
Marks: 50
Weighting: 25% of your overall assignment mark (CSSE2310)
Revision 1.Ω

Introduction

In this assignment you will write a number of small programs("agents") which move pieces around a shared map. A central program (called `handler`) will interact with each of the agents via pipes, collect their moves and distribute the information to the other agents.

Your assignment submission must comply with the C style guide available on the course website. This is an individual assignment. You should feel free to discuss aspects of C programming and the assignment specification with fellow students. You should not actively help (or seek help from) other students with the actual coding of your assignment solution. It is cheating to look at another student's code and it is cheating to allow your code to be seen or shared in printed or electronic form. You should note that all submitted code may be subject to automated checks for plagiarism and collusion. If we detect plagiarism or collusion, formal misconduct proceedings will be initiated against you. A likely penalty for a first offence would be a mark of 0 for the assignment. Don't risk it! If you're having trouble, seek help from a member of the teaching staff. Don't be tempted to copy another student's code. You should read and understand the statements on student misconduct in the course profile and on the school web-site: `http://www.itee.uq.edu.au/itee-student-misconduct-including-plagiarism`

As with Assignment 1, we will use the subversion (svn) system to deal with assignment submissions. Do not commit any code to your repository unless it is your own work or it was given to you by teaching staff. **If you have questions about this, please ask.**

Invocation

The parameters required to start an agent will depend on the type of agent (see table later). Note that agents will normally be run by the handler rather than directly by the user.

The `handler` program will take the name of a map file which describes the shared environment, the maximum number of rounds to run and the name of a file describing which agents to run.
For example: `./handler my.map 20 agents.lst`

Agent file

The agent file will contain one or more lines describing agent processes to run. There will be one line per agent. Lines beginning with # are comments and should be ignored Each agent line will have the following parts (space separated):

1. row and column where the agent will start (with 1 1 being the top left corner).
2. a non-space character to indicate the position of the agent on the map.
3. The remainder of the line will be used as the name of the program to run and the command line parameters to use.

For example:

```
#my first agent file
1 1 X ./simple E
```

Would execute `./simple` with `argv[1]=="E"`. The agent would start in the top left corner with its position indicated by **X**.

Map file

The format of the map file will be as follows: The first line will give the number of rows(r) in the map, a space and then the number of columns(c). The remainder of the file will describe the map. Walls will be drawn as # and open space will be drawn as `.` . For example:

```
6 6
##.###
#....#
#.##.#
#....#
##....
######
```

Note that walls do not necessarily surround the entire border. Anything outside the borders of the map should be considered blocked though.

Interaction

In order to start a run, handler must be started. After reading and validating its input files, it will start up the agents specified in its file. It will then send the following information to each agent (in this order with a new line for each item):

- The total number of agents in the system.
- A line with the chars for each agent (in order).
- The number of this agent [starting from 1].
- The dimensions of the map (rows columns).
- The contents of the map with # representing obstacles and a space character representing open space.

Suppose we are using the following agent file:

```
# A sample agents file
2 2 A ./ag1
3 2 B ./ag2
#chars don't need to be letters.
5 3 1 ./agX
```

For example, the information sent to the first agent would be:

```
3
AB1
1
6 6
```

```
## ###
#    #
# ## #
#    #
##
######
```

Note that the agents use a different map representation to the map file. The handler will then ask each agent for a move in turn (in the order given in the agent file). It does this by sending the position of all agents (row column (one line per agent)) to the agent in question. This position information must be sent in order. For example:

```
2 2
3 2
5 3
```

The agent will reply with a line (terminated by '\n') containing either:

- a lone 'D' character indicating that the agent has succeeded in its goal and the run should end. [That is, the handler will close down].
- OR one of the following chars N S E W H — North, South, East, West or Hold(don't move) respectively.

When the handler reads the agent's reply, it displays the map (with agents represented by their characters) to *its* `stdout` (ie where the user can see it). Then it moves on to the next agent or in the case of a 'D' ends the run. When all agents in have made a move, the round counter is updated. If the maximum number of rounds has been reached, the run ends.

All communication with an agent is to be via its `stdin` and all agent output is to be read from its `stdout`.

Required Programs

In this assignment you need to have a `Makefile` to build a number of programs from your source files. You may divide up your source however you wish (with the proviso that the structure should be clear and sensible). Note that multiple programs can be built from the same `.o` files.

Agents

Implement the following agents:

Name	Action	Succeeds when
./simple	Takes an extra paramter {NSEW} which indicates the direction it should move. For its turn, if the cell in that direction not a wall, it should move, if not it should stay where it is.	After moving 10 steps in the specified direction.
./slow	Copies the moves of an agent with char='+' but waits for 10 steps before beginning. That is, it plays back with a delay	never
./slow2	Watches an agent with char='+' and records its moves. slow2 should not move while this is happening. When $agent_+$ "stops", your agent should replay all its moves since the beginning of the run	When the replay is complete.
./waller	Takes an extra parameter which is the inital movement direction. See below for algorithm.	never
./fromfile	Takes an extra parameter which is a file to read. Read a char $\in${N,S,E,W,H} if any other char is read, or if the char represents a move that would run the agent into a wall, then your move is 'H' (on your next turn go to the next move in the file). If the move is legal, then make it.	When there are no more moves to read.

Waller algorithm

The following instructions assume that the current movement direction is *East*. They must be adjusted for a different direction. Once a move is chosen, the algorthm should stop until another movement is required.

1. If you are surrounded by walls, then your move is *H*.
2. If there is a wall to the North-West and no wall to the North:
   (a) Turn left.
   (b) If there is an agent to the North, then your move is *H*.
   (c) else move forward.
3. If there is a wall in front of you then:
   - Turn right until there is no wall in front of you
4. If an agent is blocking your path, then your move is *H*.
5. else move forward.

The Handler

You will need to implement the handler as described in this spec. This part will be where most of the work on inter-process communication will happen. However, since it relies on the existance of at least one agent program to do much, your first step should probably be to get the `fromfile` agent at least partially working first.

Exits

Exit status for handler

Condition	Exit	Message	Dest.
Agent declared success	0	Agent %c succeeded.	**stdout**
Wrong number of command line arguments	1	Usage: handler mapfile maxsteps agentfile	stderr
maxsteps$\leq$ 0 or not a number or too big to fit in an `int`	2	Invalid maxsteps.	stderr
Error opening map file	3	Unable to open map file.	stderr
Error reading map file	4	Corrupt map.	stderr
Error opening agent file	5	Unable to open agent file.	stderr
Error reading agent file	6	Corrupt agents.	stderr
Can't start an agent	7	Error running agent.	stderr
Agent moved into obstacle	8	Agent %c walled.	stderr
Agent moved onto another agent	9	Agent %c collided.	stderr
Maximum steps reached	10	Too many steps.	stderr
Invalid response from agent	11	Agent %c sent invalid response.	stderr
Agent closed unexpectedly	12	Agent %c exited with status %d.	stderr
Agent died from signal	13	Agent %c exited due to signal %d.	stderr
SIGINT received	14	Exiting due to INT signal.	stderr
Serious system error [won't test this one]	20	ARRGGGGGG	stderr

In the above, %c should be replaced with the char for that agent. Replace %d with the exit status of the agent.

Conditions should be checked in the order given in the table. For example, you should check if you have the correct number of arguments before checking if maxsteps is a number or not. Agents going missing or sending invalid responses should be checked when a response is expected (ie after the handler has sent an update).

Exit status for agents

Condition	Exit	Message	Dest.
Achieved goal	0	D	stdout
Incorrect number of parameters	1	Incorrect number of params.	stdout
Invalid parameters	2	Invalid params.	stdout
Needs another agent to work and that agent is not present. For example needs an agent with symbol '+'. Or refers to a file which is empty or cannot be read.	3	Dependencies not met.	stdout
Handler shutsdown unexpectedly or sends bad info	4	Handler communication breakdown.	stdout

Compilation

Your code must compile (on a clean checkout) with the command:
`make`

Each individual file must compile with at least `-Wall -pedantic -std=gnu99`. You may of course use additional flags but you must not use them to try to disable or hide warnings. You must also not use pragmas to achieve the same goal.

If the make command does not produce one or more of the required programs, then those programs will not be marked. If none of the required programs are produced, then you will receive 0 marks for functionality. Any code without academic merit will be removed from your program before compilation is attempted [This will be done even if it prevents the code from compiling]. If your code produces warnings (as opposed to errors), then you will lose style marks (see later).

Your solution must not invoke other programs apart from those listed in the agent files. Your solution must not use non-standard headers/libraries.

Submission

Submission must be made electronically by committing using subversion. In order to mark your assignment, the markers will check out `/trunk/ass3/` from your repository on `source.eait.uq.edu.au`. Code checked in to any other part of your repository will not be marked.

The due date for this assignment is given on the front page of this specification. Note that no submissions can be made more than 120 hours past the deadline under any circumstances.

Test scripts will be provided to test the code on the trunk. Students are *strongly advised* to make use of this facility after committing.

Note: Any .h or .c files in your trunk will be marked for style *even if they are not linked by the makefile.* If you need help moving/removing files in svn, then ask.

You must submit a `Makefile` *or we will not be able to compile your assignment.* Remember that your assignment will be marked electronically and strict adherance to the specification is critical.

Marks

Marks will be awarded for both functionality and style.

Functionality (42 marks)

Provided that your code compiles (see above), you will earn functionality marks based on the number of features your program correctly implements, as outlined below. Partial marks may be awarded for partially meeting the functionality requirements. Not all features are of equal difficulty. If your program does not allow a feature to be tested then you will receive 0 marks for that feature, even if you claim to have implemented it. For example, if your program can never open a file, we can not determine if your program would have loaded input from it. The markers will make no alterations to your code (other than to remove code without academic merit). Your programs should not crash or lock up/loop indefinitely. Your programs should not run for unreasonably long times.

- fromfile agent* (4 marks)
- simple agent* (4 marks)
- handler errors 1 . . . 6 (5 marks)

- fromfile + handler (5 marks)
- simple + handler (4 marks)
- slow + handler (4 marks)
- slow2 + handler (4 marks)
- waller + handler (6 marks)
- combined (6 marks)
  This item covers tests with a variety of agents as well as edge cases.

The items marked with * indicate the agent programs being tested independantly of the handler. That is, the handler does not need to exist to get these marks. However, *these agent types will be used to help test later types so they are extremely important.* The `+ handler` items mean that the system will be tested by checking the output of the handler program rather than individual agents. Later agents may use earlier agents as part of their tests.

Style (8 marks)

If g is the number of style guide violations and w is the number of compilation warnings, your style mark will be the minimum of your functionality mark and:

$$8 \times 0.9^{g+w}$$

The number of compilation warnings will be the total number of distinct warning lines reported during the compilation process described above. The number of style guide violations refers to the number of violations of version 1.6 of the C Programming Style Guide. A maximum of 5 violations will be penalised for each broad guideline area. The broad guideline areas are Naming, Comments, Braces, Whitespace, Indentation, Line Length and Overall. For naming violations, the penalty will be one violation per offending name (not per use of the name) up to the maximum of five. You should pay particular attention to commenting so that others can understand your code. The marker's decision with respect to commenting violations is final — it is the marker who has to understand your code. To satisfy layout related guidelines, you may wish to consider the `indent(1)` and `expand(1)` tools. Your style mark can never be more than your functionality mark — this prevents the submission of well styled programs which don't meet at least a minimum level of required functionality.

Late Penalties

Late penalties will apply as outlined in the course profile.

Specification Updates

It is possible that this specification contains errors or inconsistencies or missing information. It is possible that clarifications will be issued via the course website. Any such clarifications posted 5 days (120 hours) or more before the due date will form part of the assignment specification. If you find any inconsistencies or omissions, please notify the teaching staff.

Test Data

Test data and scripts for this assignment will be made available. The idea is to help clarify some areas of the specification and to provide a basic sanity check of code which you have committed. *They are not guaranteed to check all possible problems nor are they guaranteed to resemble the tests which will be used to mark your assignments.* Testing that your assignment complies with this specification is still *your* responsibility.

Notes and Addenda:

1. This assignment implicitly tests your ability to recognise repeated operations/steps and move them into functions to be reused. If you rewrite everything each time it is used, then writing and debugging your code will take much longer.
2. Start early.
3. Write simple programs to try out fork(), exec() and pipe().
4. Be sure to test on moss.
5. You should not assume that system calls always succeed.
6. You are not permitted to use any of the following functions in this assignment.
   - `system()`
   - `popen()`
   - `prctl()`
7. You may assume that only 8bit characters are in use[no unicode].
8. When handler exits, it should not leave any child processes running.
9. You must follow *version 1.7* of the C programming style guide found at on the blackboard site All tab characters will be replaced using the `expand` tool before assignment are marked.
10. Agents should not send more than one move at a time and should wait to be asked for the next move before sending.
11. The maximum number of steps parameter is invalid if it cannot be stored in an `int` variable.
12. The agent's stderr is not used so make use of it for debugging. Make sure to remove it all before final commit though.
13. Regarding parsing **numbers**, for this assignment, you can assume that anything `scanf("%d")` would accept is valid.
14. Any numeric quantities, (eg dimensions, number of agents) should fit in an `int` variable. If they do not, that is an error. (See `INT_MAX`).
15. Slow2 stops recording moves by the + agent when playback begins. So it is actually possible for slow2 to complete its goal.
16. Changes in 1.1
    (a) Do not use any `#pragma` in this assignment

(b) Added exit status for serious system error. eg can't create pipes. I will not test for this error. It is just here to give you something to return if it happens. `exec()` failure is not a serious error.

17. Changes in 1.Ω

(a) The slow2 agent should never output H once it has started playback. That is, the H from the + agent which triggered playback should not form part of slow2's move sequence.

(b) The map should only be displayed when:
You have got a valid move back from an agent.
AND
That move is not 'D'
AND
The agent has not crashed into something.