The University of Queensland
School of Information Technology and Electrical Engineering
Semester Two, 2018
CSSE2310 / CSSE7231 - Assignment 3
Due: 11:10pm 28th September, 2018
Marks: 50
Weighting: 25% of your overall assignment mark (CSSE2310)
Revision 1.2

Introduction

In this assignment, you will write two types of `C99` programs to run and play a game (described later). The first type of program (players) will listen on their `stdin` for information about the game and give their actions to `stdout`. These types of program will send information about the state of the game to `stderr`. The second program (`austerity` — also known as the *hub*) will start a number of player processes and communicate with them via pipes. The hub will be responsible for running the game (sending information to players; processing their moves and determining the winner(s)). The hub will also ensure that information sent to `stderr` by the players, is discarded.

You will produce a number of player programs (with different names) which implement different playing strategies. However all players will probably share a large amount of common code (with the other players *you* write).

Your programs must not create any files on disk not mentioned in this specification or in command line arguments. Your assignment submission must comply with the C style guide (version 2.0.4) available on the course blackboard area. 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.**

While you are permitted to use sample code supplied by teaching staff this year in this course, code supplied for other courses or other offerings of this course is off limits — it may be deemed to be without academic merit and removed from your program before testing.

The game

The game consists of two or more players, a deck of cards and five piles of tokens. The token types are: Purple (P), Brown (B), Yellow (Y), Red (R) and Wild. There are a limited number of tokens of each colour, while there are an unlimited number of tokens in the Wild pile.

To begin the game, 8 cards are drawn from the top of the deck and placed face up. If ever a face up card is bought by a player, a new card will be drawn from the deck (additional cards will not be drawn if the deck is empty). All players begin with nothing - ie. they have 0 tokens and 0 cards.

Cards

Each card has the following features:

- Cost to purchase (number of each colour is always a non-negative integer)
  - Eg: 1P + 2B + 0Y + 0R
- Number of points the card is worth (always a non-negative integer)
  - Eg: 2
- Colour, used for discounts on all future card purchases
  - Eg: R

Players will be granted a single token discount of the card colour for each card they own. For example, a player with three cards, one R and two B, will now have a discount of 1R and 2B. The discount will apply to all future card purchases.

- Scenario 1:
  Player A has two Brown and three Red cards. This means they have a discount of 2B and 3R. When purchasing a card worth 3R, no additional tokens are required as the card discount of 3R has supplied the cost of Red for this card.
- Scenario 2:
  Player C has one Purple, three Brown and two Red cards. This means they have a discount of 1P, 3B and 2R. When purchasing a card worth 2P+1B+3Y, they will need 1 Purple and 3 Yellow tokens to make up the remaining cost after applying their discount.

Player turns

On their turn, each player takes **one** of the following actions:

1. Purchase a card, by paying the token cost (applying any card discounts). Non-wild tokens to be used in preference to wilds. Tokens used in a purchase are returned to their corresponding pile (including wild tokens).
2. Take 3 tokens, each of a different colour. If two or more of the coloured piles are empty, this action will not be possible.
3. Take a single wild token. (Wild tokens can be used to make up for missing tokens of any colour when purchasing cards).

End of game

The game continues until a player reaches the specified number of points required for a win. Once a player reaches this number of points, the round continues until all players have had their turn (this means all players will have had the same number of turns at the end of the game). It is possible for multiple players to reach the target number of points by the end of the game. The winner(s) of the game are the players with the highest number of points. Note that it is possible for the first player to reach the target points to not be a winner, as later players could end up on more points during their final turn.

If the end of the deck is reached before players have reached the points required for a win, the game will continue while there are still cards available for purchase. No new cards will be turned over, as there are none to draw from. Remaining face up cards will continue to be available for purchase. Once all face up cards have been purchased, the game will end. The winner(s) in this scenario are the players with the highest number of points.

Game parameters

A game will require the following information:

1. A file storing the deck of cards to be used.
2. The number of tokens of each colour (this is a single value).
3. The number of points which triggers game end.

Deck file

The hub will use the deck file to load the available deck of cards for a game. Each card will be on a new line within the file. There must be at least one card in the deck file for it to be valid.
The structure for a card is: $D{:}V{:}P_P, P_B, P_Y, P_R$

- D $\in \{P, B, Y, R\}$ — The colour of discount this card gives.
- V — Number of points this card is worth.
- $P_?$ — Price of this card in the colour of tokens (can be zero)

For example:

```
B:1:1,0,0,0
Y:0:0,3,0,2
```

Invocation

Invocation — players

The parameters to start a player are (in order):

1. The number of players (in total).
2. The number of this particular player (starting at 0).

The maximum number of players permitted in a game will be 26.
The player will read game information from `stdin` and send its responses to `stdout`.

Invocation — hub

The parameters to start the hub are (in order):

1. The number of tokens in each of the (non-wild) piles. This is a non-negative integer.
2. The number of points required to trigger the end of the game. This is a non-negative integer.
3. The deck file to use.
4. Player programs to run. There must be at least two players.

For example:
`./austerity 7 15 cards ./shenzi ./shenzi ./shenzi`
would start a game having 7 tokens in each non-wild pile, requiring 15 points to trigger the end of game, using the `cards` deck file, and with 3 players (each running `./shenzi`).

Arguments should be parsed in the order specified above. The deck file should be loaded before attempting to start player proceses.

The hub will be responsible for starting up the player processes and communicting with them via pipes to `stdin` and `stdout`.

Message Representations

When the players and hub communicate, they must do so using messages encoded in the following way.

Messages from the hub to players

Encoded form	Parameters	Purpose
tokens$T$	T — Number of tokens in each non-wild pile.	Informs the player how many tokens are in the non-wild piles. Only sent at the start of the game, before first round starts.
newcard$D{:}V{:}T_P, T_B, T_Y, T_R$	D $\in \{P, B, Y, R\}$ — The colour of discount this card gives V — Number of points this card is worth $T_?$ — Number of each colour of token this card costs (can be zero)	Inform players that a card has been drawn from the deck and added to the board.
dowhat		Ask the player for their choice of action.
purchased$P{:}C{:}T_P{,}T_B{,}T_Y{,}T_R{,}T_W$	P — Player who submits the action C — Card number (0 to 7) $T_?$ — Number of each colour of token involved in the purchase	Inform players that a card was purchased.
took$P{:}T_P, T_B, T_Y, T_R$	P — Player who submits the action $T_?$ — Number of each colour of token taken	Inform players that a player took tokens.
wild$P$	P — Player who submits the action	Inform players that someone took a wild token.
eog		Inform player of end of game.

Whenever player's identity is communicated in messages, player 0 is indicated by 'A', 1 by 'B' and so on.

Cards are treated as being in a sorted list, oldest to newest (oldest is the first card to be available face up). The card number references the position of the card in this list, with the oldest being 0 and the newest being 7.

Players should check that messages are correct in structure, but they do not need to ensure that messages are correct as per game state. This means that if a message from the hub is of the correct structure and can be parsed, the player should accept this as a valid message and perform the appropriate action.

Messages from players to the hub

Encoded form	Parameters	Purpose
purchase$C{:}T_P, T_B, T_Y, T_R, T_W$	C — Card number (0 to 7) $T_?$ — Number of each colour of token used	Purchase a card
takeT_P, T_B, T_Y, T_R	$T_?$ — Number of each colour of token taken	Take tokens
wild		Take a wild token

The hub should perform thorough checks of all messages received from players. For each message received, it should check that the structure is correct and can be parsed. Once this has been done, the hub should check that the action is valid in the context of the current game state.

Output

The hub and players will print information explaining what is happening within the game while it is in progress.

Output — hub

The hub will print the following messages to standard out.

Event	Output to stdout	Parameters
Card added to market	New card = Bonus D, worth V, costs T_P, T_B, T_Y, T_R	D $\in \{P, B, Y, R\}$ — The colour of discount this card gives V — Number of points this card is worth $T_?$ — Number of each colour of token this card costs (can be zero)
Player bought a card	Player P purchased C using T_P, T_B, T_Y, T_R, T_W	P — Player letter C — Card number (0 to 7) $T_?$ — Number of each colour of token used
Player took tokens	Player P drew T_P, T_B, T_Y, T_R	P — Player letter $T_?$ — Number of each colour of token used
Player took a wild	Player P took a wild	P — Player letter
Game ends early	Game ended due to disconnect	
Game over	Winner(s) L	L — comma *separated* list of winners (sorted by player letter, smallest to largest) Eg. *A,B,C*

Output — players

After receiving each message, the players will send the following to standard error.

Message	Output to stderr	Parameters
eog	Game over. Winners are L	L — comma *separated* list of winners (sorted by player letter, smallest to largest) Eg. *A,B,C*
dowhat	Received dowhat	

For each other *valid* message received (except those in the table above), show the current game information:

1. Display each card in the market (in order oldest to newest, each on a new line):
   Card C:*Bonus*/*Score*/T_P, T_B, T_Y, T_R
2. For each player (in position order, each on a new line):
   Player *letter*:*TotalPoints*:Discounts=D_P, D_B, D_Y, D_R:Tokens=T_P, T_B, T_Y, T_R, T_W

For example:

```
Card 0:B/3/0,0,2,1
Card 1:Y/1/1,0,0,0
Card 2:Y/2/0,2,0,1
Card 3:B/0/0,0,0,1
Card 4:R/4/3,2,0,0
Card 5:P/2/0,2,1,1
Card 6:R/1/1,0,0,1
Card 7:B/1/0,1,0,1
Player A:5:Discounts=1,2,0,1:Tokens=2,2,0,1,1
Player B:2:Discounts=0,4,3,0:Tokens=0,0,3,4,0
```

Hub

Game play

To commence the game, the hub will send all players the cards that have been turned face up using the `newcard` message. After this, it will coordinate a number of rounds of the game, until the end condition is met. The basic order of play in each round is:

1. Ask a player to choose an action
2. Once a valid action has been chosen, inform **all** players of the chosen action
3. Move to next player and start process again.

When asking a player for an action, if the player sends an invalid response, the hub shall prompt the player again for an action. If the player again responds with an invalid message (ie. the hub has received two invalid responses in a row), the hub will then end the game with a protocol error.

Shutting down the hub

Upon completion of the game by any means, or in response to SIGINT, the hub must shutdown and exit. Whether the hub shuts down in response to a SIGINT or of its own volition, it should follow the same procedure. First, all child processes should be instructed that the end of game has been reached. The hub should then ensure that all child processes have terminated. If they have not all terminated within 2 seconds of the hub sending the end of game message, then the hub should terminate them with SIGKILL.

During shutdown, additional information may be printed out by the hub. These messages will only be printed if certain conditions are met at the time of shutdown. If the conditions are not met, no information will be printed out during shutdown. The conditions that need to be met:

1. all children started successfully.
2. the hub is shutting down normally (not as a result of SIGINT).

When these conditions are met, for each player (in order), the hub will do one of the following:

1. If the process terminated normally with exit status 0, then don't print anything.
2. If the process terminated normally with a non-zero exit status, then print (to `stderr`):

   ```
   Player ? ended with status ?
   ```

3. If the process terminated due to a signal, then print (to `stderr`):

   ```
   Player ? shutdown after receiving signal ?
   ```

The `?` placeholders should be filled in with the appropriate values for player letter and status/signal number.

Players

The following describes players which you will create. This does not mean that every player your hub is run with will be one of these three.

Type A - shenzi

The executable for this player will be called: `shenzi`
Makes decisions as follows:

1. If you have enough tokens to buy a card, buy one. Choose the card worth the most points. If there are multiple cards which you can afford, prioritise cards as follows. Stop when one of these gives a single choice:

   (a) The smallest number of tokens in the cost.
   (b) The card which was added most recently.

2. If we could take three tokens, prioritise token colours in the following order:
   (a) Purple
   (b) Brown
   (c) Yellow
   (d) Red

3. Take a wild

Type B - banzai

The executable for this player will be called: `banzai`
Makes decisions as follows:

1. If you have less than three tokens, take three in the following order:
   (a) Yellow
   (b) Brown
   (c) Purple
   (d) Red

2. If one or more cards are available and worth non-zero points, purchase one. Prioritise based on (factors are additive):
   (a) The most expensive card (total number of tokens)
   (b) Card which would use the most wild tokens
   (c) The oldest card

3. Take a wild

For example: On its turn, the `banzai` player goes through the above process. The first option is to take tokens. It currently has 5 tokens, so it does not take any more. The next option is to take a card. There are currently 4 cards, costing a total of 1, 2, 4 and 4 tokens respectively. Based on this, `banzai` will take this action, so it now needs to select a card. The most expensive card costs 4 tokens, but there are two of these cards. It then needs to apply further filtering to get down to a single card to purchase. From these two cards costing 4 tokens, based on the tokens and discounts `banzai` currently has, both cards would need 3 wild tokens in order to puchase them. This has still not filtered it to one card, so the oldest card of the two is selected. (Remember that the oldest card is the one that has been face up the longest). A card has now been selected, and now it will be purchased.

Type C - ed

The executable for this player will be called: `ed`
Makes decisions as follows:

Look at all opponents, identify the face up card worth the highest number of points which any of them can afford now. If there is more than one such card, prefer cards affordable by players after you in the round (ie. if you are player 2 of 4, then prefer cards for Player 3, then Player 0, then Player 1.) If there are multple such cards, choose the oldest card. Once the card has been identified, determine an action based on the following steps. (Note: Skip steps with a * if there was no card identified)

1. You can afford the card, purchase it*.

2. If you can take three tokens, take them in the following order:
   (a) Yellow* (if there is Yellow in the card's price and you do not have enough Yellow to afford it)

   (b) Red* (if there is Red in the card's price and you do not have enough Red to afford it)
   (c) Brown* (if there is Brown in the card's price and you do not have enough Brown to afford it)
   (d) Purple* (if there is Purple in the card's price and you do not have enough Purple to afford it)
   (e) Yellow
   (f) Red
   (g) Brown
   (h) Purple
3. Take a wild

For example: On its turn, the `ed` player identifies that the card with the highest points that the other players can afford is worth 3P,2B,0Y,1R. The player currently has tokens worth 1P, 2B, 1Y, 0R, and does not have any discounts, so cannot afford to purchase the card. It then decides to take three tokens. The current number of tokens in each pile is: 4P, 5B, 0Y, 2R. The token selection process it goes through:

1. Yellow (if there is Yellow in the card's price and you do not have enough Yellow to afford it)
   - Do not need Yellow for the card, so do not take.
2. Red (if there is Red in the card's price and you do not have enough Red to afford it)
   - Card is worth 1R, and player has 0R, so take 1 Red.
3. Brown (if there is Brown in the card's price and you do not have enough Brown to afford it)
   - Card is worth 2B, and player has 2B, so do not take.
4. Purple (if there is Purple in the card's price and you do not have enough Purple to afford it)
   - Card is worth 3P, and player has 1P, so take 1 Purple.
5. Yellow
   - There are no Yellow tokens available, so do not take.
6. Red
   - Player has taken 2 of 3 tokens, but has already taken 1 Red, so do not take.
7. Brown
   - Player has taken 2 of 3 tokens, so take 1 Brown.
8. Purple
   - Player has taken 3 tokens, so do not take.

The player has now taken 1R, 1P, 1B to complete its turn.

Exit statuses

Exit status for players

All messages to be printed to `stderr`.

Condition	Exit	Message
Normal exit due to game over	0	
Wrong number of arguments	1	`Usage: {player} pcount myid` NOTE: Replace `{player}` with the name of the player, eg. `shenzi`
Invalid number of players	2	`Invalid player count`
Invalid player ID	3	`Invalid player ID`
Pipe closed before end of game message received or invalid message received	6	`Communication Error`

Exit status for the hub

All messages to be printed to `stderr`.

Condition	Exit	Message
Normal exit due to game over	0	
Wrong number of arguments	1	`Usage: austerity tokens points deck player player [player ...]`
Invalid command line argument	2	`Bad argument`
Deck file cannot be read	3	`Cannot access deck file`
Deck file is incorrect	4	`Invalid deck file contents`
Failure starting any of the player processes	5	`Bad start`
Player closed before end of game message was sent	6	`Client disconnected`
Protocol error by player	7	`Protocol error by client`
The hub received SIGINT	10	`SIGINT caught`

Note that both sides detect the loss of the other by a read failure. Write calls could also fail, but your program should ignore these failures and wait for the associated read failure. Such write failures must not cause your program to crash.

Memory usage

It is expected that your program will allocate memory for use while operating. Your programs should operate without leaking memory. This means that any memory allocated by your program will be freed before the program exits. Memory usage and leakage will be checked for both the hub and players.

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. Your code must be compiled with the `gcc` compiler.

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 command line arguments for the hub. Your programs are not permitted to create additional files on the filesystem. Your solution must not use non-standard headers/libraries. Further, you are not permitted to make use of gnu language extensions or any of the following: `__attribute__`, `getdelim`, `getline`, `setjump`, `longjump`, `goto`, `select`. You are not permitted to use posix regex features. You are also not to use non-blocking IO nor `setbuff()` or equivalents to switch off buffering on `FILE*`.

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` [1]. 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. (Only the contents of the `trunk/ass3` directory at the deadline will be marked).

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/ass3` directory 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. Consult the style guide for other restrictions.

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.

[1] That is, `https://source.eait.uq.edu.au/svn/csse2310-$USER/trunk/ass3`

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.

- For *shenzi* player
  - Argument checking (1 mark)
  - Responds correctly to their first turn (2 marks)
  - Correctly handles early loss of hub and ending messages (2 marks)
  - (Player) Correctly handles complete game (6 marks)
  - (Player) Detects invalid messages (2 marks)
- For *austerity* hub
  - Argument checking (1 mark)
  - Correctly loads deck file (2 marks)
  - Detects failure to start players (2 marks)
  - Correctly handles players which close early (3 marks)
  - Correctly handles 2 player games with *shenzi* players (4 marks)
- Play complete games with 3 *banzai* players. (2 marks)
- Play complete games with 4 *ed* players. (4 marks)
- Play complete games with a mixture of player types. (7 marks)
- Memory usage
  - *shenzi* does not leak memory when playing a game to completion (1 mark)
  - *austerity* does not leak memory under normal operation and shutdown (1 mark)
  - *austerity* does not leak memory after SIGINT received and shutdown (2 marks)

Style (8 marks)

Style marks will be calculated as follows:
Let A be the number of style violations detected by simpatico plus the number of build warnings. Let H be the number of style violations detected by human markers. Let F be the functionality mark for your assignment.

- If $A > 10$, then your style mark will be zero and M will not be calculated.
- Otherwise, let $M_A = 4 \times 0.8^A$ and $M_H = M_A - 0.5 \times H$. Your style mark S will be $M_A + \max\{0, M_H\}$.

Your total mark for the assignment will be $F + \min\{F, S\}$.

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)` tool. 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. (`testa3.sh`, `reptesta3.sh`) 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()`
   - `setjmp()`
7. You may not use any `#pragma` or `goto` in this assignment.
8. You may not use regex in this assignment.
9. Neither program should assume that the other will be well behaved. That is, if the hub sends a valid message, you may believe it. However, if the hub sends an invalid message, your player should not crash.
10. All messages are sent as strings, terminated by a new line. There are no null bytes within a valid message.
11. You will need to do something with SIGPIPE.
12. Valid messages contain no leading, trailing or embedded spaces. Messages must be exactly the correct length.
13. Make a separate player to test the hub against badly behaved players rather than breaking an existing one.
14. You should only report on the exit statuses of the players if all players started successfully.
15. `struct`s and `enum`s are your friends. Use them.

16. Just because communication is required to be in a particular format, does not mean that you must store information internally in that form. Instead, convert from the external format to the internal format as soon as possible and convert to the external form as late as possible.

17. There will be a number of different communication channels in a complete system. Make sure that you can monitor any of the channels.

18. The recommended way to detect a child failing to start is by using the close-on-exec flag (`FD_CLOEXEC`) for a file descriptor. This will cause the file descriptor to be automatically closed if a call to the `exec()` family of functions is successful.

19. The hub may not always be given full paths to players. Instead, players could be located on the `$PATH` of the system. You should use an appropriate version of the `exec()` family of functions to support this.

20. Valid numbers are anything that is accepted by the `strtol()` function, excluding leading whitespace. Only whole numbers (integers) should be accepted.