Ian Malerich - Project 01

Generated by Doxygen 1.8.13

Contents

1 Data Structure Index

1.1 Data Structures

Here are the data structures with brief descriptions:

2 File Index

2.1 File List

Here is a list of all documented files with brief descriptions:

3 Data Structure Documentation

3.1 _History Struct Reference

Used a doubly linked list to describe the users history.

`#include <history.h>`

Data Fields

- size_t cid
- char * command
- struct _History * next
- struct _History * prev

3.1.1 Detailed Description

Used a doubly linked list to describe the users history.

3.1.2 Field Documentation

3.1.2.1 cid

```
size_t _History::cid
```

Command ID of this item in the history.

3.1.2.2 command

```
char* _History::command
```

Pointer to the command returned by getline.

3.1.2.3 next

```
struct _History* _History::next
```

Pointer to the next item in the linked list.

3.1.2.4 prev

```
struct _History* _History::prev
```

Pointer to the previous item in the linked list.

The documentation for this struct was generated from the following file:

- include/history.h

4 File Documentation

4.1 include/args.h File Reference

Argument parsing from a string input line.

```
#include <stdbool.h>
```

Functions

- char * next_command (char *line)
  Find the next command present in the multi-command line.
- char * split_line (char *line)
  Split a line by argument by replacing spaces with null terminators.
- char ** get_arg_array (char *args)
  Get the list of all arguments.
- bool run_in_background (char *line)
  Should the input line be run in the background?
- bool pipe_to_next (char *line)
  Should the input line pipe to the next command?
- int num_args (char *args)
  How many argumens are present in the input string?
- char * get_arg (char *args, int i)
  Given arguments as parsed by split_line, and an index where 0<=i<num_args(args), returns a pointer to the first character of the string denoting argumnent i. If no such argument can be found, NULL will be returned.
- void print_args (char *line)
  Debug utility: Parses all arguments in the input line. then prints each argument on its own line.

4.1.1 Detailed Description

Argument parsing from a string input line.

Author

Ian Malerich

4.1.2 Function Documentation

4.1.2.1 get_arg_array()

```
char** get_arg_array (
            char * args )
```

Get the list of all arguments.

Converts the arguments of form char * as returned by split_line into an array of pointers to corresponding locations in that string. This form can then be passed into execvp.

4.1.2.2 next_command()

```
char* next_command (
            char * line )
```

Find the next command present in the multi-command line.

Given a line containing a command, returns a pointer to the next command stored in the line. If there are no more commands, returns NULL. Commands are separated by ';'. Note that this does not allocate a new string, only returns a pointer into 'line'.

4.1.2.3 num_args()

```
int num_args (
            char * args )
```

How many argumens are present in the input string?

Given arguments which were passed by split_line, returns how many arguments were found in the line.

Parameters

Arguments	as parsed by split_line.

Returns

Number of arguments found by split_line.

4.1.2.4 pipe_to_next()

```
bool pipe_to_next (
            char * line )
```

Should the input line pipe to the next command?

Returns

True if the command is delimited by a |, False otherwise.

4.1.2.5 run_in_background()

```
bool run_in_background (
            char * line )
```

Should the input line be run in the background?

Returns

True if a '&' is found before a ';' or the end of the string.

4.1.2.6 split_line()

```
char* split_line (
            char * line )
```

Split a line by argument by replacing spaces with null terminators.

Allocates a new string of length(line)+1-LW. Where LW is the number of leading whitespace characters, these will be ommitted from the generated string. Further, length(line) is the length of line up to the first occurence of the ';' character, which terminates the command. Replaces white space characters in line with NULL terminators for each argument. An extra NULL terminator is then added so the return value may be used as an array of strings.

Parameters

line	A NULL terminated command to be parsed.

Returns

An array of strings corresponding to the arguments of 'line', this will need to be freed from memory when it is no longer needed.

4.2 include/exec.h File Reference

Execute processed commands, setup pipes.

```
#include <ctype.h>
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/wait.h>
#include "../include/history.h"
```

Functions

- void exec_line (char *line, History *h, int _fd[2], pid_t prev)

  Executes all commands in the given line.
- bool should_exit (char *line)

  Checks whether or not the first word in the given line is 'exit'.
- void exec_command (char *command, History *h)

  Executes the line and overlays the current process.

4.2.1 Detailed Description

Execute processed commands, setup pipes.

Author

Ian Malerich

4.2.2 Function Documentation

4.2.2.1 exec_command()

```
void exec_command (
            char * command,
            History * h )
```

Executes the line and overlays the current process.

If this function returns, an error has occured.

4.2.2.2 exec_line()

```
void exec_line (
            char * line,
            History * h,
            int _fd[2],
            pid_t prev )
```

Executes all commands in the given line.

Commands are executed in the order in which they appear. Each command may include a '&' character to be run in the background that is, the creating process will not wait for the new process to terminate.

4.2.2.3 should_exit()

```
bool should_exit (
            char * line )
```

Checks whether or not the first word in the given line is 'exit'.

If so, returns true, otherwise returns false. Note that the line may have words after or whitespace before. and this function can still return true. However it is a strict requirment that it must be a word, therefore 'exitcl' for example would return false.

4.3 include/history.h File Reference

Store user command history.

```
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <unistd.h>
```

Data Structures

- struct _History

  Used a doubly linked list to describe the users history.

Typedefs

- typedef struct _History **History**

Functions

- History * history_add_or_create (History *h, char *command, int MAX_HIST)
- History * history_alloc (char *command)
- void history_remove (History *h)
- History * history_push (History *head, char *command)
- History * history_pop (History *head)
- unsigned history_length (History *h)
- void history_print (History *h)
- char * history_get_command (int cid, History *h)

4.3.1 Detailed Description

Store user command history.

Author

Ian Malerich

4.3.2 Function Documentation

4.3.2.1 history_add_or_create()

```
History* history_add_or_create (
            History * h,
            char * command,
            int MAX_HIST )
```

If h = NULL, creates a new history list and returns it, containing only command. Otherwise, pushes command to the front of the history stack. If this causes history to be larger than MAX_HIST, then the tail of the stack is popped.

4.3.2.2 history_alloc()

```
History* history_alloc (
            char * command )
```

Allocates a new history list, containing only the given command.

4.3.2.3 history_get_command()

```
char* history_get_command (
            int cid,
            History * h )
```

Get the command from the history for the given cid. If no such command is found, NULL is returned.

4.3.2.4 history_length()

```
unsigned history_length (
            History * h )
```

Returns the length of the linked list of which h is a member. Note, this is always the entire size of the doubly linked list, not just the remaining elements as any element can be treated as the head.

4.3.2.5 history_pop()

```
History* history_pop (
            History * head )
```

Pops the tail (head->prev) of the linked list via the history_remove method.

4.3.2.6 history_print()

```
void history_print (
            History * h )
```

Output the history list to the console.

Parameters

h	Pointer to history list (most recent is head).

4.3.2.7 history_push()

```
History* history_push (
            History * head,
            char * command )
```

Pushes a new history node in front of 'h' storing the given command as data. A pointer to that node is then returned.

4.3.2.8 history_remove()

```
void history_remove (
            History * h )
```

Removes the given node from the history. This will free h->command as well as h from memory.

4.4 include/line.h File Reference

Process a line before execution begins.

```
#include <ctype.h>
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <stdbool.h>
#include <unistd.h>
#include "history.h"
```

Functions

- char * insert_spaces (char *line)
  Adds spaces after ';' in a string.
- char * proc_line (char *line, History *h)
  Replace history tags with actual commands in a line.
- bool is_digit (char c)
  TRUE if the input character is a digit (0-9), FALSE otherwise.
- int get_digit (char c)
  Converts the input character into an integer digit.
- int add_to_string (char **str, char *add, int idx, size_t *buffer_size)
  Utility call for proc_line.
- void * parse_error (const char *err, char *line)
  Outputs the given error to the console.
- char * copy_str (char *string)
  Makes a new copy of the input strig in memory.

4.4.1 Detailed Description

Process a line before execution begins.

Author

Ian Malerich

4.4.2 Function Documentation

4.4.2.1 add_to_string()

```
int add_to_string (
            char ** str,
            char * add,
            int idx,
            size_t * buffer_size )
```

Utility call for proc_line.

Copies the input 'add' string to the reference 'str' starting at the location 'add'. This will reallocate str as necessary, the new buffer size of str will be modified as input. Once done, returns the updated idx, where the caller can continue writing to 'str'.

4.4.2.2 copy_str()

```
char* copy_str (
            char * string )
```

Makes a new copy of the input strig in memory.

Since the history object is freeing commands as they leave memory, whenever I reference an old commannd, I need to make a copy of the string before adding it to my history object.

4.4.2.3 get_digit()

```
int get_digit (
            char c )
```

Converts the input character into an integer digit.

Assumes is_digit(c) == TRUE.

4.4.2.4 insert_spaces()

```
char* insert_spaces (
            char * line )
```

Adds spaces after ';' in a string.

Commands are stored compact in the history. For example "ls;pwd" but my algorithm needs spaces to deterimen the ends of strings, this will insert spaces after the end of each command "ls ;pwd ". This creates a new string, and does not deallocate the calling string.

4.4.2.5 parse_error()

```
void* parse_error (
            const char * err,
            char * line )
```

Outputs the given error to the console.

If line != NULL, line will be freed from memory. Always returns NULL;

4.4.2.6 proc_line()

```
char* proc_line (
            char * line,
            History * h )
```

Replace history tags with actual commands in a line.

Uses 'get_command' to process the given line and create a new equivalent line which includes no history commands. For example, on an input of "!!;ls" where !! refers to a pwd command, this function will return a new string "pwd;ls". Further, this function will free 'line'. Thus, it is safe and encouraged to use "line = proc_line(line, h);".

Index