minishell

Created 2021/08/24

tag 42Seoul 42Seoul minishell Write a Shell in C Redirection Pipe Built-In Commands Various Operators

Subjects

- External Functions

  we `<say.h>`

  - opendir
  - closedir
  - readdir
  - example

  on `<term.h>`

  - NCURSES
  - TERM environment variable
  - Terminal control method
  - tcgetattr
  - tcsetattr
  - tgetent

- tgetnum
- tgetstr
- tgoto
- tputs
- example

on `<sys/ioctl.h>` & `<sys/wait.h>`

- ioctl
- wait3
- wait4

on `<unistd.h>` & `<stdlib.h>` & `<signal.h>`

- getcwd
- chdir
- isatty
- ttyname
- ttyslot
- gettenv
- signal

on `<readline/readline.h>` & `<readline/history.h>`

- readline
- rl_on_new_line
- rl_replace_line
- rl_redisplay
- add_history
- example

Approach

- Cooperation
- Implementation

- Reference

`The functions allowed to implement a minishell` are very extensive compared to previous tasks. Let's check the functions that have been previously introduced by clicking the `Toggle below` .

Functions introduced in `get_next_line`

Functions introduced in `ft_printf`

Functions introduced in `miniRT`

Functions introduced in `pipex`

Functions introduced in `Philosophers`

Other functions

In addition to the functions presented above, the following functions are allowed, and we will learn about them.

Functions allowed in `<dirent.h>`

- opendir

- readdir

Functions allowed in `<term.h>`

- tgetstr
- tgoto
- tgetent
- tgetflag
- tgetnum
- tputs

Functions allowed in `<sys/termios.h>`

- tcsetattr
- tcgetattr

Functions allowed in `<sys/ioctl.h>` and `<sys/wait.h>`

- ioctl
- wait3
- wait4

- chdir
- getcwd
- isatty
- ttyname
- ttyslot
- getenv
- signal

Functions allowed in `<readline/readline.h>` and `<readline/history.h>`

- readline
- rl_on_new_line
- rl_replace_line
- rl_redisplay
- add_history

I wanted to write about the above functions by disassembling them by `library ,` such as `< math.h> , <pthread.h>,` and `<semaphore.h> , but when I looked through all the libraries presented` , there were a lot of them. Therefore, I decided to look for the remaining functions in `the library when needed in the future, and for now` , I compromised by being familiar with only the functions related to `the minishell .`

2. we <say.h>

1) opendir

1. Dependency

```
# include  <dirent.h>
```

```
DIR *opendir(const char *name);
```

3. Function description

Returns `a pointer` of type `DIR *` that refers to `a directory stream` with the name corresponding to `name .` If there is no directory corresponding to `name or there is a problem with function execution, the DIR *` type `pointer` return value is `NULL .`

```
/* structure describing an open directory. */
typedef struct {
    int __dd_fd;    /* file descriptor associated with directory */
    long    __dd_loc;   /* offset in current buffer */
    long    __dd_size;  /* amount of data returned */
    char    *__dd_buf;  /* data buffer */
    int __dd_len;   /* size of data buffer */
    long    __dd_seek;  /* magic cookie returned */
    __unused long   __padding; /* (__dd_rewind space left for bincompat) */
    int __dd_flags; /* flags for readdir */
    __darwin_pthread_mutex_t __dd_lock; /* for thread locking */
    struct _telldir *__dd_td; /* telldir position recording */
} DIR;
```

The type called `DIR` is a structure defined in `<dirent.h> .` The variables maintained inside the structure are as shown in the picture above, and it can be seen that there are variables to describe directories in an overall open state. In other words, `the DIR` structure is mainly used as a means for directory manipulation.

`DIR *, a pointer` type of a structure called `DIR , is` commonly called `a directory stream , and` is used in a similar format to FILE `*, a file stream` for regular file manipulation . `Stream` in `directory stream` and `file stream` refers to the data flow of an abstracted intermediary so that specific tasks can be easily performed. To understand this, we need to understand the origin of streams.

`In Unix,` in order to express the device, the device itself is abstracted and implemented as a file. Expressing the device itself as a file means that the

reducing many existing inefficient manual tasks. At this time, the series of data flows exchanged between the files was defined as `a stream .`

In the past, computers were configured for input and output manually using punch cards, disks, and custom tapes, but this was overcome by abstracting devices into files. Connected devices are mounted in the `/dev` directory and maintained in file format.

As can be seen from the above origin, `a directory stream` is a flow of data exchanged to make directory-related tasks easier to perform, and the structure that abstracts this can be understood as `DIR .`

2) closedir

1. Dependency

```
# include <dirent.h>
```

2. Function prototype

```
int closeddir ( DIR * dirp ) ;
```

3. Function description

It is responsible for closing `the directory stream` opened through the `opendir function. If the call to closedir` is successful, it returns `0` , otherwise it returns `-1 .`

3) readdir

```
# include  <dirent.h>
```

2. Function prototype

```
struct dirent *readdir(DIR *dirp);
```

3. Function description

It receives the opened `directory stream as an argument and` returns a `pointer type` of `a dirent structure` that refers to the entries in the directory . If a problem occurs in function execution, `NULL` is returned, and even if the end of `the directory stream` is reached and there are no more entries, `NULL` is returned.

`NULL` is returned both when the end of `the stream` is reached and when a problem occurs , so it is necessary to distinguish between the two cases. This can be distinguished through `errno .` `When the end of the stream is reached` , there is no change in the previously set `errno , and when a problem occurs,` `the errno` changes to fit the problem situation. Therefore, in order to handle errors, verification of `the errno value is required` after checking for `NULL .`

```
#define __DARWIN_STRUCT_DIRENTRY { \
    __uint64_t  d_ino;      /* file number of entry */ \
    __uint64_t  d_seekoff;  /* seek offset (optional, used by servers) */ \
    __uint16_t  d_reclen;   /* length of this record */ \
    __uint16_t  d_namlen;   /* length of string in d_name */ \
    __uint8_t   d_type;     /* file type, see below */ \
    char        d_name[__DARWIN_MAXPATHLEN]; /* entry name (up to MAXPATHLEN bytes) */ \
}

#if __DARWIN_64_BIT_INO_T
struct dirent __DARWIN_STRUCT_DIRENTRY;
#endif /* __DARWIN_64_BIT_INO_T */
```

In order to properly utilize the returned entry, it is necessary to understand

```
/*
 * File types
 */
#define DT_UNKNOWN       0
#define DT_FIFO          1
#define DT_CHR           2
#define DT_DIR           4
#define DT_BLK           6
#define DT_REG           8
#define DT_LNK          10
#define DT_SOCK         12
#define DT_WHT          14
```

In Mac OS _ _ _ In this case, you must use the stat structure and stat function in <stat.h> to receive information about the file, and then use a macro function to check the type of the file. All you need to know is that not only are these functions not allowed in minishell , but they can be replaced by simply using the dirent structure, so you can use things like <stat.h> .

For the pointer type of the dirent structure obtained through the readdir function , the space in memory referenced by the pointer is statically allocated, so there is no need to call the free function separately.

4) Example

In the example using <dirent.h>, we will write code that reads a specific directory based on the program execution location and checks the entries inside the directory.

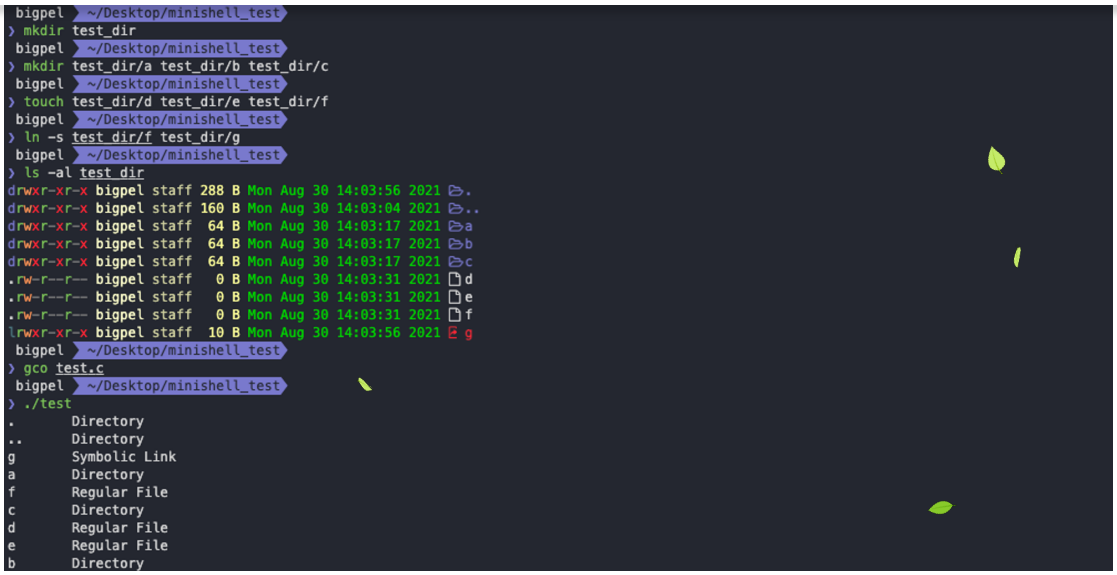
```
#include <dirent.h>
#include <errno.h>
#include <stdbool.h>
#include <stdio.h>

void  classify(struct dirent *ent)
{
  printf("%s\t", ent->d_name);
  if (ent->d_type == DT_BLK)
    printf("Block Device\n");
```

```
  else if (ent->d_type == DT_DIR)
    printf("Directory\n");
  else if (ent->d_type == DT_LNK)
    printf("Symbolic Link\n");
  else if (ent->d_type == DT_REG)
    printf("Regular File\n");
  else if (ent->d_type == DT_SOCK)
    printf("Unix Domain Socket\n");
  else
    printf("Unknown Type File\n");
}

int   main(void)
{
  int       temp;
  DIR       *dirp;
  struct dirent *file;

  dirp = opendir("test_dir");
  if (!dirp)
  {
    printf("error\n");
    return (1);
  }
  while (true)
  {
    temp = errno;
    file = readdir(dirp);
    if (!file && temp != errno)
    {
      printf("error\n");
      break ;
    }
    if (!file)
      break ;
    classify(file);
  }
  closedir(dirp);
  return (0);
}
```



3. on <term.h>

Before checking the functions that exist within `<term.h>` , you need to first understand the concepts required in `<term.h>` . `The library` called `<term.h>` that can be used in `Mac OS X` can be understood as `a library` for controlling functions on the terminal . Not only can tasks such as controlling the cursor on the terminal, deleting part of the screen on the terminal, or changing the currently visible color be performed through `<term.h> , but programs executed by the user on the terminal can also be output.` To interact with the terminal using characters that cannot be used, `<term.h> is also` used.

1) NCURSES

```
/* termcap database emulation (XPG4 uses const only for 2nd param of tgetent) */
#if !defined(NCURSES_TERMCAP_H_incl)
extern NCURSES_EXPORT(char *) tgetstr (NCURSES_CONST char *, char **);
extern NCURSES_EXPORT(char *) tgoto (const char *, int, int);
extern NCURSES_EXPORT(int) tgetent (char *, const char *);
extern NCURSES_EXPORT(int) tgetflag (NCURSES_CONST char *);
extern NCURSES_EXPORT(int) tgetnum (NCURSES_CONST char *);
extern NCURSES_EXPORT(int) tputs (const char *, int, int (*)(int));
#endif /* NCURSES_TERMCAP_H_incl */
```

If you look for the functions allowed in the assignment within `<term.h>` , you can see that they are grouped with `the NCURSES macro definition as`

a toolkit for creating GUI applications that can run on a Terminal Emulator and provides various APIs . It can open multiple windows, control the keyboard or cursor, add color, etc., and is mainly used on Unix . According to the picture above, you can see that <term.h> operates based on the terminal-related functions of NCURSES .

NCURSES is derived from the existing library called CURSES for Cursor Optimization , and is one of the control libraries for Unix- based operating systems . Development was discontinued in the 1990s, but development began in GNU and was named NCURSES . Therefore, if you are using the latest version of a Unix- based system , it is automatically installed when installing GCC .

```
#if !defined(NCURSES_TERM_H_incl)
extern NCURSES_EXPORT(char *) tgetstr (NCURSES_CONST char *, char **);
extern NCURSES_EXPORT(char *) tgoto (const char *, int, int);
extern NCURSES_EXPORT(int) tgetent (char *, const char *);
extern NCURSES_EXPORT(int) tgetflag (NCURSES_CONST char *);
extern NCURSES_EXPORT(int) tgetnum (NCURSES_CONST char *);
extern NCURSES_EXPORT(int) tputs (const char *, int, int (*)(int));
#endif
```

In addition to <term.h> , there is a library that operates based on the terminal-related functions of NCURSES , <termcap.h> , and the roles of the two libraries for terminal control are the same. However, if you open each library and check, you can see that <term.h> includes functions other than NCURSES . The picture above is the part bound by the NCURSES macro definition in <termcap.h> . If you compare it with the picture in <term.h> , you can see that all the functions imported from NCURSES are the same , with only the macro definition that prevents duplicate inclusion of NCURSES being different. there is.

```
✘ bigpel ~/Desktop/minishell_test
❯ gco test.c
Undefined symbols for architecture x86_64:
  "_tgetent", referenced from:
      _main in test-4ac63d.o
ld: symbol(s) not found for architecture x86_64
clang: error: linker command failed with exit code 1 (use -v to see invocation)
```

In order to use functions imported from NCURSES in <term.h> or <termcap.h>, the process of importing the actual code of the function is also necessary. For example, if you compile a code that calls one of the functions imported from NCURSES , you can see that the compilation failed because the corresponding function cannot be found, as shown in the picture above.

The reason it was not compiled is because the actual codes of the functions exported through `NCURSES_EXPORT were not imported, so` you can check that it compiles normally by specifying `NCURSES` with the compilation option `-l as shown in the picture above.`

2) TERM environment variable

As `<term.h> is a library for terminal control` , it is affected by `the TERM environment variable.` Therefore, it is necessary to understand what exactly `the TERM` environment variable is, why `the TERM` environment variable is needed, and the values that can be used as `the TERM environment variable`

What is the TERM environment variable?

`The TERM` environment variable is an environment variable that specifies the type of login terminal. It is important to note that the type of terminal mentioned here does not refer to which terminal emulator is used. `The role of the TERM` environment variable is quite clear: it tells user applications running on the terminal how to interact with the terminal. In other words, `the TERM` environment variable can be considered to be used by the user application, and the interaction between the user application and the terminal at this time includes even input/output operations.

TERM environment variable (UNIX)

https://www.ibm.com/docs/en/informix-servers/12.10?topic=products-term-environment...

Need for TERM environment variable?

Interaction between the user application and the terminal is accomplished through `the Escape Sequence` , which can be used to control the keyboard, cursor, and terminal screen. However, because there was no clearly defined `escape sequence` standard, physical terminals in the past had different `escape sequences that could be used depending on the type of terminal.`

sequences that are combinations of meaningless characters . Therefore, when a query for a desired task is sent to the operating system, the operating system is designed to process the query and deliver an appropriate escape sequence . As a result, from the operating system's point of view, to be able to do this, it must understand all escape sequences used by a specific type of terminal, so the operating system was designed to have an internal DB to properly convert it. This is called Terminal Capability (TermCap) .

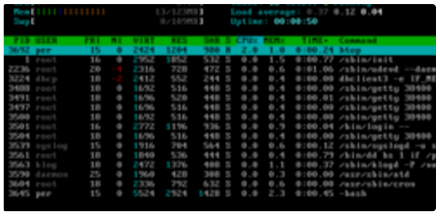
In many recent systems, a DB called TermInfo is used instead of TermCap . Additional information related to terminals, such as TermCap and TermInfo , can be obtained from the link below.

[Linux] shmat - Shared memory management ...
This chapter deals with screen input and output that is character-based, not pixel-based. When we talk about
https://anythink.tistory.com/entry/Linux-shmat-sh...

In summary, in order for the user application to send a query for the desired task and the operating system to send an appropriate Escape Sequence from TermCap , it is necessary to first set the type of terminal the user application wants to use, and in this case, the TERM environment variable is used.

Recently, a standard for Escape Sequence has been established, and most terminals follow it. Since the functions provided may differ depending on the terminal type, the DB itself is still required, but there is no significant functional difference depending on the value of the TERM environment variable. Escape Sequence defined as a standard can be found at the link below.

ANSI escape code - Wikipedia, the free encycl...
ANSI escape sequences are a standard for in-band signaling to control cursor location, color, font styling,
https://en.wikipedia.org/wiki/ANSI_escape_code



What values are available for the TERM environment variable?

The values available for the TERM environment variable generally have the same name as the terminal emulator, so the value of the TERM environment variable should not be understood to mean the terminal

example, using `gnome-terminal does not necessarily mean that the TERM` environment variable can only be used with `gnome-terminal .` Even if you use `gnome-terminal, you can use xterm` or `xterm-256color` , and only the interaction method changes depending on the terminal type. So, is there a correct answer to the value used as `the TERM environment variable?`

A list of terminal emulators available for each system can be found at the link below.

List of terminal emulators - Wikipedia, the free ...

https://ko.wikipedia.org/wiki/Terminal_emulator_list

There is no clear answer as to what should be set as the value of the `TERM environment variable,` but it is recommended to use `xterm` or `xterm-256color .` This is because most terminal emulators that support `GUI` use or are compatible with the terminal type called `xterm .` This can be understood a little more easily through remote communication situations such as `ssh .`

For example, the environment in which a program is running$A$And the environment that shows this is$B$Let's say in other words,BVia `ssh` in the environment$A$The program is running in the environment. The environment in which the user application interacts with the terminal is$A$Because of,BIf you want to use appropriate `TermCap` in a situation where you are controlling the environment,AYou must also have an entry for the environment. Therefore, in remote communication, it is necessary to match the types of both terminals to be the same. Since `xterm` has been widely used and has high compatibility, it is common to stick to `xterm and use it.`

Of course, if the values of the terminal emulator and `the TERM` environment variable are the same, the terminal emulator can interact with its own terminal type, so the terminal can benefit in terms of query processing. However, since the user application only needs to send a query, the difference in impact on the user application is minimal. Therefore, it is recommended to use a more widely used terminal type such as `xterm` or `xterm-256color .`

There is not a huge difference between `xterm` or `xterm-256color .` As

slight difference in the number of color expressions, `xterm` also supports `Text Color .`

3) Terminal control method

`To control the terminal, you must use the TermCap` or `TermInfo query` mentioned above , and it is necessary to register an entry in advance so that you can use the query. Since such entry registration and query operations basically use the functions of the terminal itself, it is necessary to first set the attribute values of the object you want to interact with in the terminal.

```
struct termios {
    tcflag_t     c_iflag;     /* input flags */
    tcflag_t     c_oflag;     /* output flags */
    tcflag_t     c_cflag;     /* control flags */
    tcflag_t     c_lflag;     /* local flags */
    cc_t         c_cc[NCCS];  /* control chars */
    speed_t      c_ispeed;    /* input speed */
    speed_t      c_ospeed;    /* output speed */
};
```

Setting property values of interaction objects can be handled using a structure called `termios .` Types such as `tcflag_t` , `cc_t` , and `speed_t` are just types defined as `unsigned long` , `unsigned char` , `etc., and the role of member variables in termios` can be understood from the annotations shown in the picture above.

Each member variable is used as a flag to set terminal property values. If you vary the values assigned to each flag, you can see that the results for input and output operations are different. The value assigned to each flag can be easily found by searching the comments shown in the picture, such as `input flag` and `output flag ,` inside `<sys/termios.h> , and` it is easy to see that these values are used as `bit operations.` Therefore, if the expected results are not obtained when implementing a shell that runs in the terminal, `the termios` structure can be manipulated appropriately. If you cannot find a flag through the library code or there is a part you do not understand, we recommend searching for the flag in the link below.

If you included `<term.h> ,` you do not need to include `<sys/termios.h> separately.` If you look closely at `<term.h>,` you can see that it includes

tcgetattr(3) - Linux man page

termios, tcgetattr, tcsetattr, tcsendbreak, tcdrain, tcflush, tcflow, cfmakeraw, cfgetospeed, cfgetispeed, cfsetispeed, cfsetospeed, cfsetspeed - get and set terminal attributes, line

https://linux.die.net/man/3/tcgetattr

In order to determine the terminal property value by manipulating the termios structure, you must import the existing terminal property value as a termios * type, modify the value inside the imported termios structure, and then apply it. These tasks can be performed with functions called tcgetattr and tcsetattr, respectively.

After determining the terminal property values, as mentioned at the beginning, you need to get an entry for which terminal type you are using so that TermCap or TermInfo can properly process the query. This kind of work can be performed through a routine function called tgetent .

If you specify the terminal type while loading the entry, you can use the query of TermCap or TermInfo . Functions performed through queries are called routines . The routine function used varies depending on whether TermCap or TermInfo is used , but TermCap functions are allowed in the minishell . The corresponding functions include routines such as tgetstr , tputs , tgetflag , tgetnum , and tgoto .

As previously mentioned, <term.h> contains all the functions of <termcap.h>, so there is no need to additionally include <termcap.h> . tgetent is also a routine that uses the entry name as a query .

The query used for each routine is different, and the query is used as an argument called id of char * type in the routine function. Queries that can be used in each routine can be found at the link below.

twintk/load_termcap.c at 8b7adb088f1e4b467...

Terminal Windows Toolkit. Contribute to outpaddling/twintk development by creating an account

https://github.com/outpaddling/twintk/blob/8b7ad...

https://nxmnpg.lemoda.net/ko/5/termcap

4) tcgetattr

1. Dependency

```
#include <term.h>
```

2. Function prototype

```
int tcgetattr(int fd, struct termios *t);
```

3. Function description

`It receives the file descriptor` of the object you want to interact with as an argument . Assign the `termios` structure to record the target property value as `a pointer type.` Since attribute values are recorded using `pointers , the return value` represents the execution result of the `tcgetattr function.` If the function is successfully executed, `0` is returned; otherwise, `-1` is returned.

5) tcsetattr

1. Dependency

```
#include <term.h>
```

```
int tcsetattr(int fd, int action, const struct termios *t);
```

3. Function description

`It receives the file descriptor` of the object you want to interact with as an argument . To set the property value of the target, `the termios` structure is assigned as a `pointer type.` The timing at which the attribute value of the `termios structure is applied` is determined by `action .`

```
/*
 * Commands passed to tcsetattr() for setting the termios structure.
 */
#define TCSANOW         0               /* make change immediate */
#define TCSADRAIN       1               /* drain output, then change */
#define TCSAFLUSH       2               /* drain output, flush input */
#if !defined(_POSIX_C_SOURCE) || defined(_DARWIN_C_SOURCE)
#define TCSASOFT        0x10            /* flag - don't alter h.w. state */
#endif
```

There are four types of values that can be used as `action , as shown in the picture above.` `TCSANOW` means immediate change to the value of the `termios structure.` `TSCADRAIN` and `TCSAFLUSH` indicate a change after all write operations have been performed on the target `file descriptor .` `The difference is that` `TCSAFLUSH` discards the input operation being processed, while `TCSADRAIN` does not. `When TCSASOFT` is used, the values of `c_cflag` , `c_ispeed` , and `c_ospeed in` `the termios` structure are ignored.

If the function is successfully executed, `0` is returned; otherwise, `-1` is returned.

6) tgetent

1. Dependency

```
#include <term.h>
```

2. Function prototype

```
int tgetent(char *bp, const char *name);
```

3. Function description

This is a routine that sets the entry of the terminal type corresponding to the name and makes it possible to perform a TermCap query for the entry . Generally, the value assigned to name uses the terminal type assigned to the TERM environment variable. At this time, the Buffer Pointer called bp is an ignored argument, so it is common to assign NULL .

There are few cases where Buffer Pointer is used without being ignored. If a Buffer Pointer is used, it is also used in other routine functions. However, in most cases , the Buffer Pointer is ignored, and the Buffer commonly used by Routines is internally allocated and used.

If the routine 's operation is successful , 1 is returned. If the value of name is empty, such as a null string, when the routine is executed, 0 is returned. In case of failure in routine execution, such as not finding the entry of name in the DB , -1 is returned.

7) tgetflag

1. Dependency

```
#include <term.h>
```

2. Function prototype

```
int tgetflag(char *id);
```

3. Function description

The name to be used in the query is received as an argument called `id` . If the query can obtain the flag, it returns `true (1)` , otherwise it returns `false (0)` .

8) tgetnum

1. Dependency

```
#include <term.h>
```

2. Function prototype

```
int tgetnum(char *id);
```

3. Function description

The name to be used in the query is received as an argument called `id` . If the value corresponding to the query can be obtained, that value is returned. Otherwise, `-1` is returned.

1. Dependency

```
#include <term.h>
```

2. Function prototype

```
char *tgetstr(char *id, char **area);
```

3. Function description

The name to be used in the query is received as a parameter called `id` . `If the Escape Sequence` corresponding to the query can be obtained, the string is returned. Otherwise, `NULL` is returned. The argument called `area refers to the Buffer Pointer` used in `tgetent` , but since it is an ignored argument, `NULL` is usually given.

`Just as the Buffer Pointer` was internally allocated and used in `the routine` called `tgetent , the area` is also said to be ignored, but it is not not used at all, but is internally allocated and used. It can be understood as being ignored in the sense that it is not used on the surface.

10) tgoto

1. Dependency

```
#include <term.h>
```

2. Function prototype

```
char *tgoto(const char *cap, int col, int row);
```

3. Function description

`col` refers to the position of the vertical row of the terminal, and `row` refers to the position of the horizontal row of the terminal. `cap` means `Capability , and generally` uses `the Escape Sequence` for `cm` , which is `Cursor Motion .` Of course , you can use another `Escape Sequence` , but since it is safest to use `the Escape Sequence` for `Cursor Motion , it is better to avoid other Escape Sequences` for `the routine` called `tgoto` .

The `routine` called `tgoto` returns `the Escape Sequence` of `Cursor Motion` considering `col` and `row .` Therefore, if you use this string as an argument to `tputs` , you can see the cursor on the terminal move. If the operation of `the routine fails, NULL` is returned.

11) tputs

1. Dependency

```
#include <term.h>
```

2. Function prototype

```
int tputs(const char *str, int affcnt, int (*putc)(int));
```

`tputs` is `a routine that` produces terminal output results for `the Escape Sequence` , and the argument `str is the Escape Sequence` obtained through `tgetstr` or `tgoto` . `affcnt` refers to the number of lines to be affected by `tputs ,` and is generally set to `1 unless it will affect multiple` lines . You can see that the argument called `putc is a function pointer` that receives an argument of `int type` and returns an `int type. This is a function that receives an ASCII` character value as an argument and outputs `the ASCII character value` to the terminal through a writing operation to `standard output` .

`If the routine` called `tputs` is executed without a problem, it returns `0` , otherwise it returns `-1 .`

12) Example

In a typical shell, when you operate keys using the arrow keys, etc., strange symbols do not appear and the specified function is performed. For example, pressing the up arrow key displays previously entered commands. However, after running the program, if you use keys such as the arrow keys, you may see strange symbols appear as shown in the picture above. In the example that deals with `<term.h>,` we will write code that changes the key operation so that the user application can perform the desired action using terminal setting values and `NCURSES .`

In the presented code , the settings for `standard input will be changed, and the standard input` that follows the existing `Canonical` method will be changed to `a Non-Canonical` method to obtain the desired result. `Canonical` in `standard input` means that the maximum length of characters that can be processed at one time is set to `255 and input is processed one line at a time.` When changing this to `Non-Canonical, local flags` such as `ICANON` and `control characters` such as `VMIN` and `VTIME` are mainly used.

`The term Canonical` means following the rules, and conversely, `Non-Canonical` means not following the rules.

http://www.terms.co.kr/canonical.htm

Additional examples of flags that can be manipulated by changing existing terminal settings and making them `Non-Canonical can be found in the link below.`

The Linux Serial Programming HOWTO: Program Examples

Next Previous Table of Contents All examples here are taken from miniterm.c. The maximum length of characters that can be processed in Canonical input processing is 255 (or defined in

https://wiki.kldp.org/HOWTO/html/Serial-Programming/Serial-Programming-HOWTO-3.h...

Please refer to the links below for flag setting values of `the termios structure and values that can be used in TermCap queries.`

linux.die.net

https://linux.die.net/man/3/tcgetattr

termcap(5) man page

Because many escape sequences are provided as capabilities for string values, it is easy to encode

https://nxmnpg.lemoda.net/ko/5/termcap

```c
#include <ctype.h>
#include <stdbool.h>
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include <term.h>
#include <unistd.h>

#define ERROR     -1
#define KEY_LEFT    4479771
#define KEY_RIGHT   4414235
#define KEY_UP      4283163
#define KEY_DOWN    4348699
#define KEY_BACKSPACE 127
#define CURPOS      "\033[6n"

const char        *g_cm;
const char        *g_ce;
```

```
bool  init_term(struct termios *t)
{
  if (tcgetattr(STDIN_FILENO, t) == ERROR)
    return (false);
  t->c_lflag &= ~(ICANON | ECHO);
  t->c_cc[VMIN] = 1;
  t->c_cc[VTIME] = 0;
  if (tcsetattr(STDIN_FILENO, TCSANOW, t) == ERROR)
    return (false);
  return (true);
}

bool  init_query(void)
{
  if (tgetent(NULL, "xterm") == ERROR)
    return (false);
  g_cm = tgetstr("cm", NULL);
  g_ce = tgetstr("ce", NULL);
  g_dc = tgetstr("dc", NULL);
  if (!g_cm || !g_ce)
    return (false);
  return (true);
}

int   putchar(int c)
{
  if (write(STDOUT_FILENO, &c, 1) == ERROR)
    return (0);
  return (1);
}

bool  get_position(int *col, int *row)
{
  int         i;
  int         ret;
  char        buf[1024];

  if (write(STDOUT_FILENO, CURPOS, strlen(CURPOS)) == ERROR)
    return (false);
  ret = read(STDIN_FILENO, buf, 1024);
  if (ret == ERROR)
    return (false);
  buf[ret] = '\0';
  i = 0;
  while (!isdigit(buf[i]))
    ++i;
  *row = atoi(&buf[i]) - 1;
```

```
  while (!isdigit(buf[i]))
    ++i;
  *col = atoi(&buf[i]) - 1;
  return (true);
}

bool  cur_left(int *col, int *row)
{
  if (*col)
  {
    --(*col);
    if (tputs(tgoto(g_cm, *col, *row), 1, putchar) == ERROR)
      return (false);
  }
  return (true);
}

bool  cur_right(int *col, int *row)
{
  ++(*col);
  if (tputs(tgoto(g_cm, *col, *row), 1, putchar) == ERROR)
    return (false);
  return (true);
}

bool  cur_up(int *col, int *row)
{
  if (*row)
  {
    --(*row);
    if (tputs(tgoto(g_cm, *col, *row), 1, putchar) == ERROR)
      return (false);
  }
  return (true);
}

bool  cur_down(int *col, int *row)
{
  ++(*row);
  if (tputs(tgoto(g_cm, *col, *row), 1, putchar) == ERROR)
    return (false);
  return (true);
}

bool  cur_backspace(int *col, int *row)
{
  if (*col)
```

```
    if (tputs(tgoto(g_cm, *col, *row), 1, putchar) == ERROR)
      return (false);
  }
  if (tputs(g_dc, 1, putchar) == ERROR)
    return (false);
  return (true);
}

bool  key_handle(int ch, int *col, int *row)
{
  if (ch == KEY_LEFT)
  {
    if (!cur_left(col, row))
      return (false);
  }
  else if (ch == KEY_RIGHT)
  {
    if (!cur_right(col, row))
      return (false);
  }
  else if (ch == KEY_UP)
  {
    if (!cur_up(col, row))
      return (false);
  }
  else if (ch == KEY_DOWN)
  {
    if (!cur_down(col, row))
      return (false);
  }
  else if (ch == KEY_BACKSPACE)
  {
    if (!cur_backspace(col, row))
      return (false);
  }
  else
  {
    ++(*col);
    if (!putchar(ch))
      return (false);
  }
  return (true);
}

bool  read_char(void)
{
  int       ch;
```

```
  int         row;

  while (true)
  {
    if (!get_position(&col, &row))
      return (false);
    ret = read(STDIN_FILENO, &ch, sizeof(ch));
    if (ret == ERROR)
      return (false);
    if (!ret)
      return (true);
    if (!key_handle(ch, &col, &row))
      return (false);
    ch = 0;
  }
}

int   main(void)
{
  struct termios    t;

  if (!init_term(&t) || !init_query() || !read_char())
    return (1);
  return (0);
}
```

```
 ✘ bigpel  ~/Desktop/minishell_test
❯ gco -lncurses test.c
 bigpel  ~/Desktop/minishell_test
❯ ./test
jseo is testing about the writing on the running program
```

Let's run the results obtained from the example code directly and check whether the cursor or keyboard values operate normally.

In the above code, the cursor position on the terminal is found by obtaining a value in the form of `[row;colR` through `the Escape Sequence` of `\033[6n and then parsing it.` If you run the above code, you can feel that it is quite different from the actual shell. Key operations that exceed the terminal size may not work as desired, and you can see that the value is overwritten when you place the cursor over the input and receive input. The former can be easily resolved by sending a request to the device through `ioctl` , which will be introduced later , to find out the terminal size, etc., and in the latter case , you can make good use of `TermCap 's query.`

4. on <sys/ioctl.h>, <sys.wait.h>

1) ioctl

1. Dependency

```
#include <sys/ioctl.h>
```

2. Function prototype

```
int ioctl(int fd, unsigned long request, ...);
```

3. Function description

The ioctl function is a function used when sending a request to a device and is a system call . When explaining streams in <dirent.h>, it was said that all devices in Unix are abstracted and manipulated as files. Therefore, since sending a request to a device through the ioctl function is done through file manipulation, fd becomes a file descriptor that refers to the device .

A typical device manipulated through ioctl is a terminal. At this time, fd entered as an argument to ioctl becomes a file descriptor obtained through open , but sometimes unexpected results may occur as a side effect of the function. Accordingly, it is recommended to also use the O_NONBLOCK flag when performing open .

The argument called request is a code provided by the device to be sent to the device corresponding to fd , and the last argument is a pointer that refers to a specific memory space . The reason for using a variable argument even though the last argument is a pointer type is to not specify the type of the pointer in the function prototype

Generally, when calling a function, the char * type is used as the last variable argument .

The act of not specifying the pointer type in the function prototype is to ensure that the argument can be used as a variety of pointer types, and the pointer type consistent with this is void * . The reason void * is not used in the ioctl function is simply because void * was an invalid type at the time the ioctl function was defined . In some recent systems, the prototype of ioctl is void * .

If there is no problem executing the ioctl function, it returns 0 , otherwise it returns -1 . If the argument called request is used under user definition, the return value of the ioctl function can be used while performing the task according to the request . Therefore, in this case, a positive number is used as the return value when there is no problem in executing the ioctl function.

Because the semantics of the ioctl function, including arguments and return values , may vary depending on the device driver, it may not operate precisely on streams corresponding to Unix I/O .

```c
/*
 * Ioctl's have the command encoded in the lower word, and the size of
 * any in or out parameters in the upper word.  The high 3 bits of the
 * upper word are used to encode the in/out status of the parameter.
 */
#define IOCPARM_MASK    0x1fff          /* parameter length, at most 13 bits */
#define IOCPARM_LEN(x)  (((x) >> 16) & IOCPARM_MASK)
#define IOCBASECMD(x)   ((x) & ~(IOCPARM_MASK << 16))
#define IOCGROUP(x)     (((x) >> 8) & 0xff)

#define IOCPARM_MAX     (IOCPARM_MASK + 1)      /* max size of ioctl args */
/* no parameters */
#define IOC_VOID        (__uint32_t)0x20000000
/* copy parameters out */
#define IOC_OUT         (__uint32_t)0x40000000
/* copy parameters in */
#define IOC_IN          (__uint32_t)0x80000000
/* copy parameters in and out */
#define IOC_INOUT       (IOC_IN|IOC_OUT)
/* mask for IN/OUT/VOID */
#define IOC_DIRMASK     (__uint32_t)0xe0000000

#define _IOC(inout, group, num, len) \
    (inout | ((len & IOCPARM_MASK) << 16) | ((group) << 8) | (num))
#define _IO(g, n)           _IOC(IOC_VOID, (g), (n), 0)
#define _IOR(g, n, t)       _IOC(IOC_OUT, (g), (n), sizeof(t))
#define _IOW(g, n, t)       _IOC(IOC_IN,  (g), (n), sizeof(t))
/* this should be _IORW, but stdio got there first */
#define _IOWR(g, n, t)      _IOC(IOC_INOUT,   (g), (n), sizeof(t))
```

As mentioned in the return value, it is possible to directly define the task for `the request , which` can be defined through the macro function provided in `<sys/ioctl.h> .` Within `<sys/ioctl> , there is a library` called `<sys/ioccom.h>` . As shown in the picture above, the behavior of the device when it receives `a request` can be defined through the `_IO` , `_IOR` , `_IOW` , and `_IOWR functions.`

The command that executes `ioctl consists of 32 bits` , and they have a specific structure. `2 Bit` refers to task classification such as `R/W , 14 Bit` refers `to size` , `8 Bit` refers to `number` , and `8 Bit` refers to `type .` These correspond to `_IO(type, number)` and `{_IOR, _IOW, _IOWR}(type, number, size) in the figure presented shortly.`

Operations such as `R/W` used in `2 Bit` mean `00: None` , `01: Write` , `10: Read` , `11: Read/Write .` Also, since `size` is a name defined in the manual, certain systems also use that name, which is a misnomer. Since the value used in `size is sizeof(size)` , the need to rename each argument as a whole was suggested. Therefore , `a separate name` is designated for

is low when using structure or `legacy values, so it must be used with caution.`

`Please check the link below for an example of using the macro function of the ioctl` function, and the example presented here is checking the terminal size using `the ioctl` function and the `winsize` structure.

In old systems, a structure called `ttysize` was used, but with the creation of a structure called `winsize , ttysize` became `Obsolete` . The structure also exists in `<sys/ioctl.h> .` At this time, `ws_xpixel` and `ws_ypixel` of `winsize` are unused member variables.

(Embedded) Linux Kernel Module - IOCTL

ioctl configuration consists of 32 bits [2] [14 (data size)] [8 (magic number)] [8 (division number)]

https://richong.tistory.com/254

4. Example

```c
#include <stdio.h>
#include <sys/ioctl.h>
#include <unistd.h>

int main(void)
{
  struct winsize  size;

  if (ioctl(STDIN_FILENO, TIOCGWINSZ, &size) == -1)
    return (1);
  printf("Terminal Row Size:\t%d\n",size.ws_row);
  printf("Terminal Col Size:\t%d\n",size.ws_col);
  return (0);
}
```

```
 bigpel  ~/Desktop/minishell_test
❯ gco test.c
 bigpel  ~/Desktop/minishell_test
❯ ./test
Terminal Row Size:      23
Terminal Col Size:      98
```

2) wait3

1. Dependency

```
#include <sys/wait.h>
```

2. Function prototype

```
pid_t wait3(int *status, int options, struct rusage *rusage);
```

3. Function description

The wait3 function is a function that is obsolete when the waitpid or waitid function appears . The function of the function itself is the same as the waitpid function, but if you still want to understand the wait3 function, it is recommended that you first understand status and options based on the contents described in wait and waitpid in pipex through the link below .

The call wait3(status, options, rusage) is equivalent to waitpid(-1, status, options) .

pipex

PC (Program Counter)? A PC is one of the registers inside a central processing unit such as a CPU and

https://bigpel66.oopy.io/library/42/inner-circle/8

```
struct rusage {
    struct timeval ru_utime;        /* user time used (PL) */
    struct timeval ru_stime;        /* system time used (PL) */
#if __DARWIN_C_LEVEL < __DARWIN_C_FULL
    long    ru_opaque[14];          /* implementation defined */
#else
    /*
     * Informational aliases for source compatibility with programs
     * that need more information than that provided by standards,
     * and which do not mind being OS-dependent.
     */
    long    ru_maxrss;              /* max resident set size (PL) */
#define ru_first        ru_ixrss        /* internal: ruadd() range start */
    long    ru_ixrss;               /* integral shared memory size (NU) */
    long    ru_idrss;               /* integral unshared data (NU) */
    long    ru_isrss;               /* integral unshared stack (NU) */
    long    ru_minflt;              /* page reclaims (NU) */
    long    ru_majflt;              /* page faults (NU) */
    long    ru_nswap;               /* swaps (NU) */
    long    ru_inblock;             /* block input operations (atomic) */
    long    ru_oublock;             /* block output operations (atomic) */
    long    ru_msgsnd;              /* messages sent (atomic) */
    long    ru_msgrcv;              /* messages received (atomic) */
    long    ru_nsignals;            /* signals received (atomic) */
    long    ru_nvcsw;               /* voluntary context switches (atomic) */
    long    ru_nivcsw;              /* involuntary " */
#define ru_last         ru_nivcsw       /* internal: ruadd() range end */
#endif  /* __DARWIN_C_LEVEL >= __DARWIN_C_FULL */
};
```

As you can see by looking at the wait function through the pipex link , the existing wait function could not use options in waitpid , so the wait3 function was used to perform a specific function corresponding to option . Another reason for using the wait3 function instead of the wait function is because of the structure called rusage . The argument rusage is an abbreviation for Resource Usage and is a structure that indicates the amount of resource usage. If the value of the structure called rsuage of wait3 is not NULL , various information about the resources of the child process are recorded in rusage while performing the function of wait3 . You can see that the member variables of rusage are maintained as shown in the picture above.

The reason the wait3 function was obsolete is because of standardization. The wait3 function corresponds to a system call in BSD- based systems, and as it was standardized by POSIX , it was recommended to use waitpid . At this time , if you look closely at the waitpid function, you can see that it is possible to use option as wait3

purpose of the function . For this reason, `rusage` can be obtained through a function called getrusage `.` In the case of `minishell ,` `getrusage is not an allowed function, so if there is a situation where you need to obtain resource usage,` you can use the `wait3 function` at least in this case .

You can learn more about `getrusage from the link below.`

getrusage(2) - Linux man page

getrusage - get resource usage #include #include getrusage() returns resource usage measures for who, which can be one of the following: RUSAGE_SELF Return resource usage

https://linux.die.net/man/2/getrusage

The return value and error handling of the `wait3 function` are the same as `wait` and `waitpid .` The return value of `the pid_t type refers to the pid` of `the child process , and` returns `-1` if there is a problem with function execution or `the child process` is terminated by a signal .

In the given example, no separate `options` will be used and no `status will be obtained.` Since this information can be identified through `waitpid` in `pipex , in the example below` , we will understand the code that determines resource usage for an arbitrary `child process .`

4. Example

```
#include <stdio.h>
#include <sys/wait.h>
#include <unistd.h>
#include <sys/time.h>

int main(void)
{
  struct rusage ru;
  pid_t   pid;

  pid = fork();
  if (pid == -1)
    return (1);
  else if (!pid)
```

```
        return (0);
      }
      else
      {
        wait3(NULL, 0, &ru);
        printf("Parent Process\n");
        printf("=========Resource Usage of Child========\n");
        printf("Number of Context Switch (Voluntary)\t%ld\n", ru.ru_nv
        printf("Number of Context Switch (Involuntary)\t%ld\n", ru.ru_
        printf("Number of Page Swap\t%ld\n", ru.ru_nswap);
        printf("Number of Page Fault\t%ld\n", ru.ru_majflt);
        printf("Signal Received\t%ld\n", ru.ru_nsignals);
      }
      return (0);
    }
```

```
 bigpel  ~/Desktop/minishell_test
❯ gco test.c
 bigpel  ~/Desktop/minishell_test
❯ ./test
Child Process
Parent Process
=======Resource Usage of Child======
Number of Context Switch (Voluntary)    0
Number of Context Switch (Involuntary)  1
Number of Page Swap     0
Number of Page Fault    0
Signal Received 0
```

3) wait4

1. Dependency

```
#include <sys/wait.h>
```

2. Function prototype

```
pid_t wait4(pid_t pid, int *status, int options, struct rusage *ru
```

3. Function description

The wait4 function, like the wait3 function, is a function that is obsolete when the waitpid or waitid function appears . The function of the function itself is the same as the waitpid function, and the difference from the wait3 function is that the function's work can specify a certain child process .

The call wait4(pid, status, options, rusage) is the same as waitpid(pid, status, options) .

The wait4 function works the same as the wait3 function except that it can specify a child process , so it is recommended to read the wait3 function above.

The return value and error handling of the wait4 function are the same as for wait and waitpid as for the wait3 function . The return value of the pid_t type refers to the pid of the child process , and returns -1 if there is a problem with function execution or the child process is terminated by a signal .

In the given example, no separate options will be used and no status will be obtained. Since this information can be identified through waitpid in pipex , in the example below , we will understand the code that determines resource usage for an arbitrary child process .

4. Example

```
#include <stdio.h>
#include <sys/wait.h>
#include <unistd.h>
#include <sys/time.h>

int main(void)
{
  struct rusage ru;
  pid_t     pid;
```

```
        return (1);
      else if (!pid)
      {
        printf("Child Process\n");
        return (0);
      }
      else
      {
        wait4(pid, NULL, 0, &ru);
        printf("Parent Process\n");
        printf("=========Resource Usage of %d========\n", pid);
        printf("Number of Context Switch (Voluntary)\t%ld\n", ru.ru_nv
        printf("Number of Context Switch (Involuntary)\t%ld\n", ru.ru_
        printf("Number of Page Swap\t%ld\n", ru.ru_nswap);
        printf("Number of Page Fault\t%ld\n", ru.ru_majflt);
        printf("Signal Received\t%ld\n", ru.ru_nsignals);
      }
      return (0);
    }
```

```
bigpel ~/Desktop/minishell_test
❯ gco test.c
bigpel ~/Desktop/minishell_test
❯ ./test
Child Process
Parent Process
=========Resource Usage of 20205========
Number of Context Switch (Voluntary)    0
Number of Context Switch (Involuntary)  1
Number of Page Swap     0
Number of Page Fault    0
Signal Received 0
```

5. on <unistd.h>, <stdlib.h>, <signal.h>

1) getcwd

1. Dependency

```
#include <unistd.h>
```

2. Function prototype

```
char *getcwd(char *buf, size_t size);
```

3. Function description

`getcwd` is a function that obtains the absolute path where the program that called `getcwd` is running as a string. `The absolute path` is recorded in a `char * type argument` called `buf , and size` means the size of `buf .` Since the string written to `buf ends` with `the null character '\0' ,` you should keep in mind that `size` is the size including `the null character .` If the string length of the `absolute path` to be recorded in `buf ,` including null characters , exceeds `size , NULL` is returned. If recording of `the absolute path is successful, the address of buf` is returned.

The unique thing about `getcwd is that if NULL is entered` as an argument for `buf , it receives a dynamic allocation` internally of size and `returns` the address of the space. If it is impossible to record `the absolute path` by receiving `dynamic allocation` with a size of `size ,` it will receive `dynamic allocation` with a size large enough to record `the absolute path .` In this case, the user must directly call `free on the dynamically allocated space.` If there is a problem executing the function, `NULL` is returned.

4. Example

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

int main(void)
{
  char  *path;
```

```
        return (1);
    printf("%s\n", path);
    free(path);
    path = NULL;
    return (0);
}
```

2) right

1. Dependency

```
#include <unistd.h>
```

2. Function prototype

```
int chdir(const char *path);
```

3. Function description

`The chdir function` is used to change the path of the currently running program to `path .` If the path change is successful, `0` is returned, and if an error occurs, `-1` is returned.

4. Example

```
#include <unistd.h>
```

```c
int main(void)
{
  char  *path;

  path = getcwd(NULL, 0);
  if (!path)
    return (1);
  printf("Before:\t%s\n", path);
  free(path);
  path = NULL;
  if (chdir("../") == -1)
    return (1);
  path = getcwd(NULL, 0);
  if (!path)
    return (1);
  printf("After:\t%s\n", path);
  free(path);
  path = NULL;
  return (0);
}
```

```
✘ bigpel ~/Desktop/minishell_test
❯ gcc test.c
 bigpel ~/Desktop/minishell_test
❯ ./test
Before: /Users/bigpel/Desktop/minishell_test
After:  /Users/bigpel/Desktop
```

3) isatty

1. Dependency

```c
#include <unistd.h>
```

2. Function prototype

```c
int isatty(int fd);
```

3. Function description

It returns whether `the file descriptor` called `fd` received as an argument refers to the terminal. `Returns 1` if it is referencing a terminal, otherwise returns `0 .`

There are two types of errors that can occur when calling `isatty : EBADF` and `ENOTTY` . For each, these are values that occur when an unusable `fd` is used or because `fd` does not refer to the terminal. `For situations where 0` is returned, both of the presented errors are included, so there is no need to detect errors by comparing `errno` before and after calling `isatty .` This is because the two errors presented do not obscure the purpose of the function itself, which is to determine whether or not it refers to a terminal. To put it a little differently, no matter which of the two errors occurs, it is clear that `the fd received as an argument does not refer to the terminal.`

`In the example of IBM` 's `isatty function` , which can be seen in the link below, you can see that no separate error checking is performed.

isatty() - Test if descriptor represents a terminal

https://www.ibm.com/docs/en/zos/2.2.0?topic=functions-isatty-test-if-descriptor-represents...

4. Example

```
#include <fcntl.h>
#include <stdio.h>
#include <unistd.h>

void  censor(int fd, const char *s)
{
  if (isatty(fd))
  {
    if (s)
      printf("%s is referring to a terminal\n", s);
    else
      printf("File Descriptor %d is referring to a terminal\n", f
  }
  else
  {
    if (s)
```

```
            printf("File Descriptor %d is not referring to a terminal\n'
        }
    }

    int main(void)
    {
        int fd;

        fd = open("test", O_RDONLY);
        if (fd < 0)
            return (1);
        censor(STDIN_FILENO, "STDIN");
        censor(STDOUT_FILENO, "STDOUT");
        censor(STDERR_FILENO, "STDERR");
        censor(fd, NULL);
        censor(42, NULL);
        close(fd);
        return (0);
    }
```

```
 bigpel  ~/Desktop/minishell_test
❯ gcc test.c
 bigpel  ~/Desktop/minishell_test
❯ ./test
STDIN is referring to a terminal
STDOUT is referring to a terminal
STDERR is referring to a terminal
File Descriptor 3 is not referring to a terminal
File Descriptor 42 is not referring to a terminal
```

4) ttyname

1. Dependency

```
#include <unistd.h>
```

2. Function prototype

```
char *ttyname(int fd);
```

3. Function description

If the file descriptor called fd received as an argument refers to a terminal, the path to the terminal is returned as a string that ends with the null character '\0'. If there is a problem executing the function or fd does not refer to the terminal, it returns NULL. Since the returned string is internally allocated in static form, its value may be overwritten by subsequent calls to ttyname. Also, since it is allocated in static form, there is no need to make a separate free call.

4. Example

```
#include <fcntl.h>
#include <stdio.h>
#include <unistd.h>

void  censor(int fd, const char *s)
{
  if (isatty(fd))
  {
    if (s)
      printf("%s is referring to a terminal\n", s);
    else
      printf("File Descriptor %d is referring to a terminal\n", f
  }
  else
  {
    if (s)
      printf("%s is not referring to a terminal\n", s);
    else
      printf("File Descriptor %d is not referring to a terminal\n'
  }
  printf("TTYNAME:\t%s\n", ttyname(fd));
}

int main(void)
{
  int fd;

  fd = open("test", O_RDONLY);
  if (fd < 0)
    return (1);
```

```
        censor(fd, NULL);
        censor(42, NULL);
        close(fd);
        return (0);
    }
```

5) ttyslot

1. Dependency

```
#include <unistd.h>
```

2. Function prototype

```
int ttyslot(void);
```

3. Function description

Returns `the index` of the terminal referenced by the program that called `ttyslot .` If there is a problem executing the function, `0` or `-1` is `the DB` entry `index for the terminal.` Please note that `the ttyslot` function is a `Legacy function.`

```c
#include <fcntl.h>
#include <stdio.h>
#include <unistd.h>

void  censor(int fd, const char *s)
{
  if (isatty(fd))
  {
    if (s)
      printf("%s is referring to a terminal\n", s);
    else
      printf("File Descriptor %d is referring to a terminal\n", f
  }
  else
  {
    if (s)
      printf("%s is not referring to a terminal\n", s);
    else
      printf("File Descriptor %d is not referring to a terminal\n'
  }
  printf("TTYNAME:\t%s\n", ttyname(fd));
}

int main(void)
{
  int fd;

  printf("TTYSLOT:\t%d\n", ttyslot());
  fd = open("test", O_RDONLY);
  if (fd < 0)
    return (1);
  censor(STDIN_FILENO, "STDIN");
  censor(STDOUT_FILENO, "STDOUT");
  censor(STDERR_FILENO, "STDERR");
  censor(fd, NULL);
  censor(42, NULL);
  close(fd);
  return (0);
}
```

1. Dependency

```
#include <stdlib.h>
```

2. Function prototype

```
char *getenv(const char *name);
```

3. Function description

Returns a string for the value of the environment variable corresponding to `name .` If the value corresponding to the environment variable is not found or a problem occurs in function execution, `NULL` is returned. You should keep in mind that the values referenced by `getenv should not be freed` because they are internally allocated in `static form.`

Should I free/delete char* returned by getenv()?

Thanks for contributing an answer to Stack Overflow!
Please be sure to answer the question. Provide details

https://stackoverflow.com/questions/4237812/sho...

4. Example

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

int main(void)
{
  char  *term;

  term = getenv("TERM");
  if (!term)
    return (1);
```

```
    return (0);
}
```

```
 bigpel  ~/Desktop/minishell_test
❯ gco test.c
 bigpel  ~/Desktop/minishell_test
❯ ./test
Term Type is xterm-256color
```

7) signal

1. Dependency

```
#include <signal.h>
```

2. Function prototype

```
typedef void (*sighandler_t)(int);
sighandler_t signal(int signum, sighandler_t handler);
```

3. Function description

A signal is a type of `IPC` and `an interrupt at the same time` . Signals are called `interrupts that are generated from the software perspective, and when the concept of interrupts` was discussed in `Philosophers` , the subject of `interrupts was said to be hardware.` Accordingly, you may think that the signal violates the definition of `Interrupt , but` generating an `Interrupt by entering the ctrl` combination key in software uses terminal control, resulting in an `Interrupt using hardware` . In other words, the occurrence of `an interrupt` on the software side of a signal can be seen as a type of `interrupt delegation .`

```
/*
 * Language spec sez we must list exactly one parameter, even though we
 * actually supply three.  Ugh!
 * SIG_HOLD is chosen to avoid KERN_SIG_* values in <sys/signalvar.h>
 */
#define SIG_DFL         (void (*)(int))0
#define SIG_IGN         (void (*)(int))1
#define SIG_HOLD        (void (*)(int))5
#define SIG_ERR         ((void (*)(int))-1)
#else
```

The signal function is used to define the action to be performed for a specific signal. I previously mentioned signal processing in the pipex article. Signal processing operations are divided into SIG_IGN (ignore) , SIG_DFL (default definition) , and handler (user definition) . As can be seen from the fact that the types of each operation presented in the above figure are the same, the processing operation of a specific signal can be determined through the signal function. You can.

The SIG_DFL operation of each signal can be checked through the Default Action in the link below. Signals whose actions can be defined through the signal function are signals excluding SIGKILL , SIGSTOP , and SIGCONT .

Signal (IPC) - Wikipedia, the free encyclopedia

Signals are standardized messages send to a running program to trigger specific behavior (such quitting or

W https://en.wikipedia.org/wiki/Signal_(IPC)

signum of the signal function means the signal number to be defined, and handler means the function to be defined of type sighandler_t . Even though handler refers to a function , it may appear that it is not a function pointer type. This question can be resolved by following the definition of the sighandler_t type. It can be confirmed that sighandler_t is a function pointer that refers to a function that returns void and takes an int type argument to receive signum .

If you look at the return type of the signal function , you can see that it is sighandler_t . As specified in the return type, the signal function returns a specific handler. At this time, the returned handler is not the handler used to call the signal function, but refers to the handler defined before the signal processing operation was defined as a handler .

of the operation of a specific signal. If a problem occurs while executing the signal function, the signal function returns SIG_ERR rather than the previously registered handler, so it can be detected.

sighandler_t is a type defined in the GNU Extension , and in glibc , in addition to the GNU Extension, the signal function is also specified as a type defined in the BSD Extension , such as sig_t . If we express the signal function without this type, it will be void (*signal(int signum, void (*handler)(int))) (int); You can see that it appears in difficult-to-read syntax like .

There is also a problem with the signal function itself, which is that the use of the signal function in a multi-threading environment is Unspecified Behavior and the use of the signal function is not a very good function in terms of portability. If we look at the latter a little more, the main reason for using handlers other than SIG_IGN and SIG_DFL is that the semantics defining them are very different for each system. For example, in past Unix- based systems, once a signal was processed by a handler registered through the signal function, the signal processing operation was reset to SIG_DFL . Accordingly, it became necessary to define a handler for the signal again within the handler, and the system did not block signals that occurred before reaching the signal function within the handler , resulting in situations where unexpected results were obtained as they were processed as SIG_DFL . will be. Of course , this is a problem that only occurs on some Unix systems, but it has clearly become a problem in terms of portability.

In POSIX , this problem was solved by using a function called sigaction instead of the signal function. Sigaction provides more functions than the signal function through a structure called sigaction . As a result, the function called signal is avoided and the use of sigaction is recommended. In minishell , the problem that occurs in signal is not major and it does not allow sigaction and various functions related to it, so you must use the signal function, but you should keep in mind that there is a function that can be used instead of the signal function.

There are also things to be careful about when using the signal function. First, you must use only Async-Signal-Safe functions within

reentrant is not possible is printf , but printf does not perform Async-Signal-Safe . If you receive a signal while it is being output by calling printf and try to call printf during the signal processing operation , the desired output result may not be produced. Since the output of printf itself is output through Buffer Management , writing to memory is performed based on a separate internal buffer . In the presented situation, the contents of the existing buffer used for writing to memory may be lost. am. Therefore, to ensure that the operations processed within the handler are problem-free, only Async-Signal-Safe functions must be used. These functions correspond to most system calls that can be Reentrant , and the functions available within the signal function can be found at the link below.

signal-safety(7) - Linux manual page

An async-signal-safe function is one that can be safely called from within a signal handler. Many functions are not async- signal-safe. In particular, nonreentrant functions are generally

https://www.man7.org/linux/man-pages/man7/signal-safety.7.html

The second thing to note is that the processing operation should not be defined using the signal function for signals such as SIGFPE , SIGILL , SIGSEGV, etc., which are not signals generated by functions such as kill or raise . Ignoring or customizing these signals can result in a program that continues indefinitely in a problem situation because the program does not terminate even in an error situation.

4. Example

```c
#include <signal.h>
#include <stdbool.h>
#include <unistd.h>

void  handler(int signum)
{
  (void)signum;
  write(STDOUT_FILENO, "write From Signal\n", 18);
}

int main(void)
{
```

```
        ;
    return (0);
}
```

```
✘ bigpel ~/Desktop/minishell_test
❯ gco test.c
 bigpel ~/Desktop/minishell_test
❯ ./test
^Cwrite From Signal
^Cwrite From Signal
^Cwrite From Signal
^\[1]    157 quit       ./test
```

6. on <readline/readline.h>, <readline/history.h>

As you can see from the names of `<readline/readline.h>` and `<readline/history.h>`, both `libraries` belong to the `readline directory.` In `Mac OS X, a directory called readline` already exists, so you can use it.

```
 bigpel ~/Desktop/minishell_test
❯ gco test.c
test.c:11:6: error: use of undeclared identifier 'rl_replace_line'
        f = rl_replace_line;
            ^
1 error generated.
```

If you call and compile each function using the `readline directory` provided by default in `Unix systems,` you can see that functions such as `rl_replace_line do not compile properly as shown in the picture above.` This is because `Unix-` based `readline` provides only basic functions, and `the GNU Library` is required to use additional functions such as `rl_replace_line .` The functions provided by `readline` in `the GNU Library can be found at the link below.`

Depending on `the Mac OS`

GNU Readline Library

This document describes the GNU Readline Library, a utility which aids in the consistency of user interface across discrete programs which provide a command line interface. The Readline

https://tiswww.case.edu/php/chet/readline/readline.html

```
✘ bigpel ~/Desktop/minishell_test
❯ gco -lreadline -L/usr/local/opt/readline/lib -I/usr/local/opt/readline/include test.c
```

specifying the correct path as shown in the picture above can you see that it compiles without problems. If you do not want to use the readline functions provided additionally by the GNU Library , it is possible to use readline built into the system without providing separate compilation options.

Since GCC is installed by default when installing Xcode on Mac OS _ _. This can be easily installed using the command specified below.

```
brew install readline
```

As shown above, there are some things to be careful about when using readline of the GNU Library . The GNU Library 's readline uses a structure called FILE , and the structure exists in <stdio.h> . It would be nice if <stdio.h> was included in GNU Library 's <readline/readline.h> , but since this is not the case, <stdio.h> must be included before including GNU Library 's <readline/readline.h>. Only then can the functions you want to call use the FILE structure appropriately.

1) readline

1. Dependency

```
#include <readline/readline.h>
```

2. Function prototype

```
char *readline (const char *prompt);
```

The readline function itself is similar to the function implemented by get_next_line targeting standard input . Prompt received as an argument refers to the text to be displayed before receiving standard input . If the prompt argument is a null string or NULL , no separate text is output. The string used as a prompt is internally registered and used as a prompt value.

As with get_next_line , the returned char * type string is created through dynamic allocation , so direct free by the user who called readline must be supported. The string from the input received from the user up to the newline character is used as the return value of readline , but the returned string does not include the newline character. If only a space character is entered and then a newline character is entered, only a null string is returned.

The handling of EOF in the return value of readline is divided into two cases. If EOF is encountered but there is no string to process before , NULL is returned. If EOF is encountered but there is a string to process before, EOF is treated like a newline character and the string is returned appropriately.

The reason why the readline function is similar to the get_next_line function rather than the same is because among the functions of readline , there is a function that supports editing of vi or emacs . For example, as mentioned above, in order to reproduce the case where there is a string to process when EOF is encountered , you must enter Ctrl + D after entering the string. In this case, you can see that even if you enter Ctrl + D, it is not recognized as EOF . You can. This is because readline supports Key Binding according to the editing method.

```
end-of-file (usually C-d)
       The character indicating end-of-file as set, for example,
       by ``stty''.  If this character is read when there are no
       characters on the line, and point is at the beginning of
       the line, Readline interprets it as the end of input and
       returns EOF.
delete-char (C-d)
       Delete the character at point.  If this function is bound
       to the same character as the tty EOF character, as C-d
```

If you check the manual to learn more about the phenomenon, you will see something like the picture above. To enter `EOF` in `readline`, you must enter `Ctrl + D when there are no characters, and Ctrl + D when a character exists` acts as `a key binding` to delete one character. `Therefore, in order to enable EOF to be sent` when a string exists, the basic setting value must be changed separately. Setting values can be changed in `~/.inputrc`, and if the file does not exist, they can be changed in `/etc/inputrc .`

If both files do not exist, you can create them separately and operate them. The file called `inputrc is used to initialize readline`, and the file can be written very similarly to writing `vimrc .`

`Key Bindings` that can be used in `readline and Directives` that can be used in the mentioned configuration files can be checked and set at the link below. You can use them appropriately as needed.

readline(3) - Linux manual page

readline will read a line from the terminal and return it, using prompt as a prompt. If prompt is NULL or the

https://man7.org/linux/man-pages/man3/readline.3.html

`Since the readline` function supports specific `editing ,` it also provides an auto-completion function using `Tab .` This auto-completion function does not show as much completeness as auto-completion such as `Oh-My-Zsh , but it guarantees the auto-completion function of the basic shell. Therefore, if you customize` the function, it will be possible to create a highly complete `minishell` .

2) rl_on_new_line

1. Dependency

```
#include <readline/readline.h>
```

```
int rl_on_new_line(void)
```

3. Function description

The `rl_on_new_line` function is used to notify `Update-` related functions within the readline directory that the cursor has moved to the next line through a newline character. Since this is a function for notification purposes, the `rl_on_new_line` function does not directly perform newline characters. Therefore, the `rl_on_new_line` function is used after outputting a newline character.

`Update` -related functions also include `rl_redisplay` .

If there is no problem with the function execution of `rl_on_new_line` , `0` is returned; otherwise, `-1` is returned.

3) rl_replace_line

1. Dependency

```
#include <readline/readline.h>
```

2. Function prototype

```
void rl_replace_line(const char *text, int clear_undo)
```

3. Function description

```
extern const char  *rl_library_version;
extern int     rl_readline_version;
extern char    *rl_readline_name;
extern FILE    *rl_instream;
extern FILE    *rl_outstream;
extern char    *rl_line_buffer;
extern int      rl_point, rl_end;
extern int      history_base, history_length;
extern int      max_input_history;
extern char    *rl_basic_word_break_characters;
extern char    *rl_completer_word_break_characters;
extern char    *rl_completer_quote_characters;
extern Function    *rl_completion_entry_function;
extern char    *(*rl_completion_word_break_hook)(void);
extern CPPFunction *rl_attempted_completion_function;
extern int      rl_attempted_completion_over;
extern int      rl_completion_type;
extern int      rl_completion_query_items;
extern char    *rl_special_prefixes;
extern int      rl_completion_append_character;
extern int      rl_inhibit_completion;
extern Function    *rl_pre_input_hook;
extern Function    *rl_startup_hook;
extern char    *rl_terminal_name;
extern int      rl_already_prompted;
extern char    *rl_prompt;
```

`The rl_replace_line` function uses a variable called `rl_line_buffer .` Functions in `the readline directory` can use variables provided globally from `the readline directory, and rl_line_buffer` is one of them, as shown in the picture above. At this time , `rl_line_buffer` separately maintains the string entered by the user.

You can check the global variables used by functions in the `readline directory at the link below.`

GNU Readline Library

This document describes the GNU Readline Library, a utility which aids in the consistency of user interface across discrete programs which provide a command line interface. The Readline

https://tiswww.case.edu/php/chet/readline/readline.html#SEC34

`The role of the rl_replace_line` function is to replace the content entered in `rl_line_buffer with a string called text .` At this time , `clear_undo` is a value that determines whether to initialize the internally maintained `undo_list .` If the value of `clear_undo is 0` , it is not initialized. If the value is other than `0 , undo_list` is initialized. There is no separate return value for `rl_replace_line .`

`undo_list` is used for `Undo operations` among `the editing functions` supported by `the readline function.`

`rl_replace_line` can be used in many ways, but it can also be used to initialize `rl_line_buffer by Flushing` it . In particular, in this case, it can be used in situations when a signal is received. You can understand exactly how to use it by looking at an example.

4) rl_redisplay

1. Dependency

```
#include <readline/readline.h>
```

2. Function prototype

```
void rl_redisplay(void);
```

3. Function description

Like `the rl_replace_line` function, `the rl_redisplay function also` uses a variable called `rl_line_buffer . When using the rl_redisplay function, the value of rl_line_buffer` entered by the user and maintained is displayed along with a prompt. At this time, the prompt value is used as the string given as `prompt` to `the readline function.`

If you want to change the prompt, you can do so through a specific function that exists in the `readline directory.` However, keep in mind that `minishell` does not allow functions related to prompt settings.

`You can see that the rl_redisplay` function does not receive any separate arguments nor does it have a separate return value. Accordingly, it may be difficult to understand why `rl_redisplay should be used in a general situation where the readline` function is called , but `rl_redisplay is mainly` used in situations when a signal is received. You

5) add_history

1. Dependency

```
#include <readline/history.h>
```

2. Function prototype

```
int add_history(const char *line);
void add_history(const char *line);
```

3. Function description

`add_history` is a function that retrieves the string entered by the user during the basic operation of `the readline function.` The string written as `line , the` argument of `add_history` , can be recalled during execution of `the readline function using the up and down arrow keys.`

When using the built-in `readline directory in Unix , the return value is created as an int` type. If there is no problem with the function execution, `0` is returned, otherwise `-1` is returned. If you use the `readline directory` of `the GNU Library` rather than `the readline directory` built into `Unix ,` you can see that a return value of type `void is created, unlike before.` It is a good idea to use the return value after checking exactly what `readline directory` you are using .

6) Example

It would be good to give an example of the `readline directory for each`

input from the user is output and the previously input string is retrieved using the up and down arrow keys. Also, just as a new prompt is displayed using `Ctrl + C` in an actual shell, we will use the `rl_on_new_line`, `rl_replace_line`, and `rl_redisplay` functions appropriately in the example program to operate similarly.

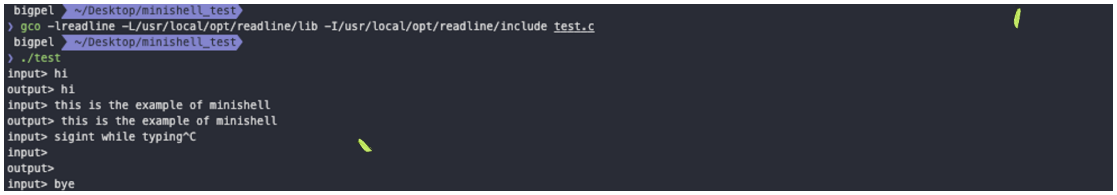
```
#include <signal.h>
#include <stdbool.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <readline/readline.h>
#include <readline/history.h>
#include <unistd.h>

void  handler(int signum)
{
  if (signum != SIGINT)
    return ;
  write(STDOUT_FILENO, "\n", 1);
  if (rl_on_new_line() == -1)
    exit(1);
  rl_replace_line("", 1);
  rl_redisplay();
}

int   main(void)
{
  int       ret;
  char      *line;

  signal(SIGINT, handler);
  while (true)
  {
    line = readline("input> ");
    if (line)
    {
      ret = strcmp(line, "bye");
      if (ret)
        printf("output> %s\n", line);
      add_history(line);
      free(line);
      line = NULL;
      if (!ret)
        break ;
```

```
            return (1);
        }
        return (0);
    }
```



7. Approach

Unfortunately, it was a pretty annoying task, and I really want to finish it quickly haha... I'll just write it down briefly.

1) Cooperation

The main repository used bigpel66/42-cursus , and team members contributed code in the form of `Issues` and `Pull Requests` on `GitHub .` Before merging a `Pull Request` , we made it a rule to conduct a code review supported by `the GitHub function.`

Although the collaboration period was not long, deciding and dividing the program structure and data format was not very difficult, perhaps because the team members thought about the structure for a long time during the gap period while catching up on progress.

In particular, in the case of `Makefile ,` `GNU` 's `readline` must be used, but there was a problem that this path may be different from that of the team member, and the path in the cluster environment may also be different. This has been deleted from the current repository, but it was written in the form below so that it can be proceeded with the `make MODE=$(NAME) command.`

```
MODE = EVAL
```

```
  LIB_HEADER  = /Users/jseo/.brew/Cellar/readline/8.1.1/include/
else ifeq ($(MODE), HYSON)
  LIB_HEADER  = /Users/hyson/.brew/Cellar/readline/8.1.1/include/
else ifeq ($(MODE), JSEO)
  LIB_HEADER  = /usr/local/opt/readline/include/
endif

ifeq ($(MODE), EVAL)
  LIB_FOLDER  = /Users/jseo/.brew/Cellar/readline/8.1.1/lib/
else ifeq ($(MODE), HYSON)
  LIB_FOLDER  = /Users/hyson/.brew/Cellar/readline/8.1.1/lib/
else ifeq ($(MODE), JSEO)
  LIB_FOLDER  = /usr/local/opt/readline/lib/
endif

%.o       : %.c
          @echo $(YELLOW) "Compiling...\t" $< $(EOC) $(LINE_CLEAR
          @$(CC) $(CFLAGS) -I $(HEADER) -o $@ -c $< -I $(LIB_HEADI

$(NAME)     : $(OBJ)
          @echo $(GREEN) "Source files are compiled!\n" $(EOC)
          @echo $(WHITE) "Building $(NAME) for" $(YELLOW) "Mandat
          @$(CC) $(CFALGS) -I $(HEADER) -o $(NAME) $(OBJ) -I $(LI
          @echo $(GREEN) "$(NAME) is created!\n" $(EOC)
```

`The Makefile` above is a partial excerpt, and when evaluating on a cluster, `readline was installed` using `brew` and then used `install_name_tool` to set the dynamic library path.

If you have problems using dynamic libraries such as `install_name_tool`, search for `install_name_tool in the link below.`

miniRT

Subjects

https://bigpel66.oopy.io/library/42/inner-circle/5

2) Implementation

I used about two major data structures. The first is the BB Tree, for

command execution .

I had previously implemented `a Set` using `RB Tree in the push_swap` task , but there were some things that I personally found disappointing. `For example, the Delete` logic did not work properly, or `the →` was so prevalent that it was difficult for even the user to recognize it later, making it unmaintainable. `So, there were times when I used RB Tree out` of personal greed , and in fact, there were times when I implemented it because I thought there would be more actions to find a specific value than the number of times I would print the entire list of environment variables. To output the entire list, when implemented as a list $O(n)$This is fine, but `RB Tree` searches for duplicate nodes.$O(nlogn)$demands. On the other hand, when looking for a specific node in the list$O(n)$However, in `RB Tree ,` $O(logn)$Because that is enough. Therefore, we decided to use `the RB Tree` in order to benefit from more frequent instructions even if we suffer some losses in the overall search .

Thinking about it now, I think it might be better to keep it as a list or `an RB Tree` , and use the list for overall search and `the RB Tree for specific node search.`

The overall process proceeds in the form `of replacement (Expand) → chunk division (Tokenize) → AS Tree form with environment variable values` . There is no problem on Mandatory, but there are bonuses such as `&&, ||` Considering the above, it was really disappointing to find out only later that the order `Tokenize → AS Tree → Expand did not fall into an edge case.`

`The reason I introduced AS Tree` was because other shells were using it, so I wanted to experience it, and since I heard that it was very easy to use in `Syntax Parsing` , I thought it would be easier to deal with various difficult edge cases that I would encounter in the future. am.

Among many cases, the combination of `Redirection` and `Pipe` ultimately benefited.

Most of the code had a recursive structure, so it was quite difficult to implement at first, but as time passed, it seemed to be quite good. I didn't

time for others, and I thought I benefited from comments, assertions, and structure.

Since this was a task to be carried out with a team member, I decided on an annotation style and set rules to record it thoroughly. When entering the repository, comments are added to each function, and it is decided to record without forgetting what purpose the function is for, what the return value is, what the arguments are, etc.

For assertions, `assert` was implemented and used. This was implemented and used as identically as possible to `assert used in general coding. In order to create clean code using assert`, the logic was structured as follows.

1. In terms of implementation, the number of dynamic allocations is quite high, but I thought that the process of checking this at every moment, returning `a bool , and doing a fallback was not desirable.`
2. The overall logic was designed to guarantee `NULL Safety (` operate safely on `NULL ).`
3. For the values required in the next logic, `assert` was used to verify that `NULL appeared in the previous logic.`

Therefore, if there is a problem in the implementation or debugging of the code below, `the mini_assert` function terminates the program and tells you which file, which line, and which function the problem occurred in, saving quite a lot of time.

```
/*
** loop ()        - Main Runtime Function of Minishell
**
** return       - void
** input        - Variable for a User Input
** chunks       - Variable for Tokens of User Input
** syntax       - Variable for a Syntax Tree from Chunks
** envmap       - Variable for Maps the Environment Variables
*/

void  loop(char *input, t_lst *chunks, t_as *syntax, t_rb *envmap
{
  while (true)
```

```
    input = readline(get_value(envmap, "PS1"));
    if (input == NULL)
    {
      jputendl("exit", STDOUT_FILENO);
      exit(VALID);
    }
    if (!jstrlen(input) || empty(input))
      continue ;
    add_history(input);
    if (!quotes(input) && set_rl(input, QUOTES, STDERR_FILENO, fa
      continue ;
    input = expand(input, envmap, false);
    mini_assert(input != NULL, \
      MASSERT "(input != NULL), " LOOP MLOOP_FILE "line 60.");
    tokenize(input, &chunks);
    mini_assert(chunks != NULL, \
      MASSERT "(chunks != NULL), " LOOP MLOOP_FILE "line 64.");
    execute(chunks, syntax, envmap);
    jlstclear(&chunks, jfree);
  }
}
```

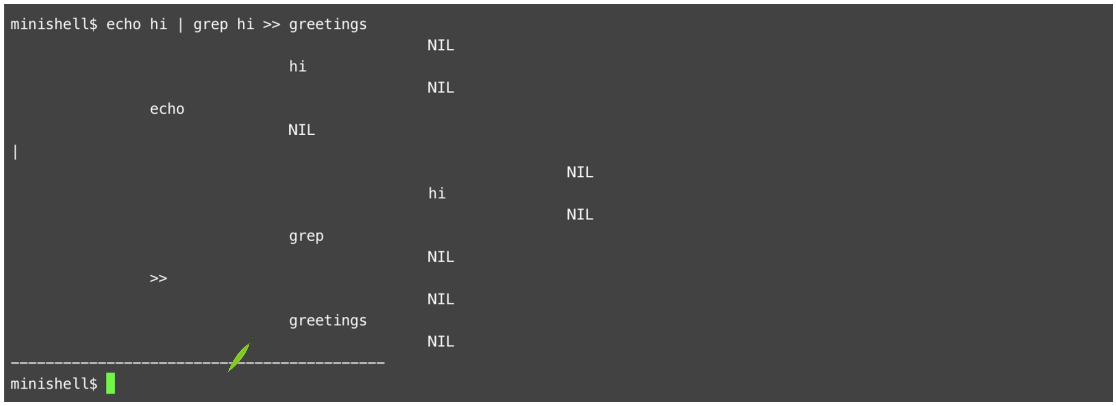
`assert` is implemented as follows:

```
/*
** mini_assert ()   - Assert Whether Condition True or False
**
** return       - void
** condition      - Condition to Check
** context        - Context Information in Runtime
*/

void  mini_assert(bool condition, char *context)
{
  if (condition)
    return ;
  jputendl(context, STDERR_FILENO);
  exit(GENERAL);
}
```

In particular, we put a lot of thought into reducing dependency on the program structure as much as possible and ensuring that each function can be used like a library. This helped to greatly reduce our dependence on each other when collaborating. I think this was possible because I defined the things needed for implementation in advance and wrote a lot of pseudo code. Accordingly, our team used five `includes as shown in the photo below.`

Additionally, I thought it would be of great help to be able to visually see the implemented contents, so I created several debugging functions while implementing each of the above functions. Therefore, even if you typed a single command, it was possible to follow it with your eyes and continuously check whether there were any problems with the structure created by the team and implement it. Thanks to this, even if a problem occurred in an edge case, it was confirmed quickly, and thanks to `the AS Tree,` we were able to respond flexibly to fixing it.



Although there were some disappointments and many difficulties during implementation, it was a good enough collaboration and I don't think it was

While organizing, I found out later that `stat` , `lstat` , and `fstat were missing. These are not that difficult, so I recommend looking for them yourself haha...`

8. Reference

Let’s learn about standard streams and standa...

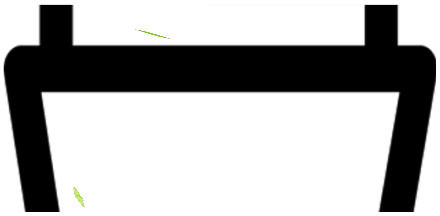
In programming, you sometimes hear talk about 'standard input/output' or 'standard stream'. This is

https://shoark7.github.io/programming/knowledge/wh...

Use of opendir(3), readdir(3), and closedir(3) -...

When developing a program, you often need to read the directory name or file name contained in a specific

https://www.it-note.kr/24



Links related to ncurses library

ncurses (new curses) is a programming library that provides an API that allows programmers to write text

https://neverapple88.tistory.com/28

A quick look at ncurses

Today we will learn about NCURSES, a terminal graphics library. According to Wikipedia, ncurses is a

https://minwook-shin.github.io/basic-ncurses/

[Linux] NCURSES Programming

Anyone who has ever used DOS probably knows a famous program called MDIRII. Using this program, you

https://anythink.tistory.com/entry/Linux-NCURSE...

[Unix C]NCURSE ?

What are NCURSES? This is a library that helps you easily use Window, Panel, Menu, Mouse, Color, etc. in

https://widian.tistory.com/58

How to configure gnome-terminal to use xterm...

https://superuser.com/questions/841016/how-to-...

Which terminal type am I using?

I have tested this with both Ubuntu 12.04 and Debian 7. When I do echo $TERM I get xterm But if I use the

https://unix.stackexchange.com/questions/93376...

What uses the TERM variable?

The TERM variable is used by programs running in a terminal. It is supposed to allow programs to determine

https://unix.stackexchange.com/questions/52832...

What is the difference between xterm-color & ...

According to the ncurses FAQ, xterm-color is long obsolete: Originally, xterm-color corresponded to the

https://stackoverflow.com/questions/10003136/w...

tcgetattr

The Open Group Base Specifications Issue 6 IEEE Std 1003.1, 2004 Edition Copyright © 2001-2004 The IEEE and The Open Group, All Rights reserved. A newer edition of this document

https://pubs.opengroup.org/onlinepubs/009604499/functions/tcgetattr.html

tcgetattr(3) man page

This section explains the functions used to control the general terminal interface. Unless specifically stated

https://nxmnpg.lemoda.net/ko/3/tcgetattr

BadproG.com

Programming with C, C++, Java SE, Java EE, Android, UNIX and GNU/Linux, PHP, MySQL, Symfony, Zend and much more!

https://www.badprog.com/unix-gnu-linux-system-calls-using-tgetflag

UNIX & GNU/Linux - System calls - Using tgete...

The tgetent() function is a Linux system call function. It is designed to be used with other termcap functions

https://www.badprog.com/unix-gnu-linux-system...

tgetent(3XCURSES) (man pages section 3: Curses Library Functions)

NAME tgetent, tgetflag, tgetnum, tgetstr, tgoto- emulate the termcap database SYNOPSIS #include PARAMETERS DESCRIPTION The tgetent() function looks up the termcap entry for

https://docs.oracle.com/cd/E19455-01/817-5436/6mkt3sbha/index.html

first, previous, next, last section, table of contents.The termcap library is the application programmer's interface to the termcap data base. It contains functions for the following

https://www.gnu.org/software/termutils/manual/termcap-1.3/html_chapter/termcap_2.html

The Termcap Library - tgoto

first, previous, next, last section, table of contents.The special case of cursor motion is handled by tgoto. There are two reasons why you might choose to use tgoto: For Unix

https://www.gnu.org/software/termutils/manual/termcap-1.3/html_node/termcap_16.html

curs_terminfo(3x) - Linux manual page

curs_terminfo(3X) curs_terminfo(3X) These low-level routines must be called by programs that have to deal

https://man7.org/linux/man-pages/man3/curs_terminf...

Termcaps : Get cursor position

I need to get my (Y, X) position of my terminal's cursor with the termcaps in C. With ioctl() i get the size of my

https://stackoverflow.com/questions/34746634/t...

(Embedded) Linux Kernel Module - IOCTL

ioctl configuration consists of 32 bits [2] [14 (data size)] [8 (magic number)] [8 (division number)]

https://richong.tistory.com/254

Mac OS X Manual Page For ioctl(2)

This document is a Mac OS X manual page. Manual pages are a command-line technology for providing documentation. You can view these manual pages locally using the man(1)

https://developer.apple.com/library/archive/documentation/System/Conceptual/ManPage...

ttyslot(3) - Linux manual page

https://man7.org/linux/man-pages/man3/ttyslot.3.html

Difference between structures ttysize and wins...

The ttysize was the original implementation for SunOS 3.0 (February 1986), and soon after made obsolete by

https://stackoverflow.com/questions/18878141/dif...

Return value of signal

Thanks for contributing an answer to Stack Overflow! Please be sure to answer the question. Provide details

https://stackoverflow.com/questions/23076909/r...

All about Linux signals

In most cases, when you want to process a signal within a program, you simply use the following syntax:

https://tdoodle.tistory.com/entry/All-about-Linux-...

[System Programming] Signals

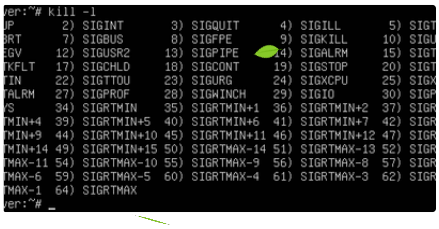
A signal is a software interrupt, which is a simple message sent asynchronously to a process notifying it

https://12bme.tistory.com/224

[Linux / Unix] What is a signal? SIGNAL types, ...

Since Twice's song Signal became popular(?), everyone knows that signal means signal... In IT, signal is not a

https://jhnyang.tistory.com/143



[C-Network] Multiprocess fork() - 2 (signal han...

Hello, this is Jjung. This time, we are configuring a multiprocess in C language, and let's do neat

https://m.blog.naver.com/joyangel93/220402541...

Why printf is not asyc signal safe function?

We know that printf is not async signal safe function.

https://unix.stackexchange.com/questions/60921...

Difference between structures ttysize and wins...

The ttysize was the original implementation for SunOS 3.0 (February 1986), and soon after made obsolete by

https://stackoverflow.com/questions/18878141/dif...

Terminal window size doesn't update using ioctl

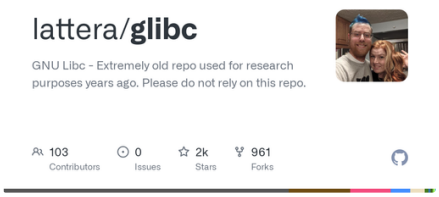
One problem in your code is obvious, after printf("\033[8;40;100t"); you haven't flushed stdout.

https://stackoverflow.com/questions/48568875/t...

glibc/ioctl-types.h at master · lattera/glibc

This repository has been archived by the owner. It is now read-only. You can't perform that action at this

https://github.com/lattera/glibc/blob/master/bits/i...



https://stackoverflow.com/questions/35316374/w...

using getrusage to get the time for parent and ...

The semantics RUSAGE_CHILDREN are pretty clearly explained in the man page: RUSAGE_CHILDREN Return

https://stackoverflow.com/questions/35100805/u...

Readline: Get a new prompt on SIGINT

I was confused at first by jancheta's answer, until I discovered that the purpose of siglongjmp is to unblock

https://stackoverflow.com/questions/16828378/re...

Today
1