# プログラミングC演習報告書

# プログラミングC第2回レポート課題

【担当教員】　長谷川 亨、内山 彰、義久 智樹　教員

【提出者】　ブアマニー　タンピモン　(09b20065)
ソフトウェア科学コース・2年
u946641i@ecs.osaka-u.ac.jp
【提出日】　2021年8月2日

# 1　課題内容

この課題にはC言語を用いて有したシェル（コマンドインタプリタ）を作成する。内部コマンドとして、このプログラムはディレクトリの管理機能の「cd コマンド, pushd コマンド, dirs コマンド, popd コマンド」、ヒストリ機能の「history コマンド, !!コマンド, !string コマンド」、ワイルドカード機能の「*コマンド」、プロンプト機能の「prompt コマンド」、そしてエイリアス機能のと「alias コマンド、unalias コマンド」が持っている。自分で考えた機能として削除機能の「rm コマンド」作成した。もっとも、ディレクトリの管理機能の「pwd コマンド」を追加した。内部コマンドだけでなく、このシェルには外部コマンドの実行を対応するようにプログラムされた。

# 2　プログラム全体の説明

## 2.1　仕様

このプログラムはスクリプト機能を通して 1 行に 1 つのコマンドを記入したテキストファイルを標準入力から読んで実行する。入力形は「mysh ¡ script_file 」である。但し、mysh じゃシェルのファイル名で、script_file はテキストファイルのファイル名で 1 行に 1 つのコマンドを記述される。ファイルは一つのコマンドをてに入れて、別名など実行できるような形に変更してからどの内部コマンドかを確認して、実行する。内部コマンドではない場合、外部コマンドを実行してみて、できないならこのコマンドが存在しないとまとめて、次のコマンドに移動する。

## 2.2　処理の流れ（処理概要）

1. プログラムはテキストファイルからコマンドの char 型の配列で持って、parse 関数を使てスペースで分けた char 型の配列の配列 args が作られる。
2. 実行の前処理として、実行できるコマンドに変更する。はじめには wildcard_command で、*の記号からカレントディレクトリにある条件を満たすファイルやディレクトリに変更する。ヒストリに関連する記号で「！」がもったらヒストリにある前のコマンドに変更する。最後に alias の別名から本当のコマンド言葉に変更する。
3. すべての別名がコマンドに変更したので、save_history 関数でコマンドを history 変数に保存する。
4. 引数 1 にある「コマンド」が内部コマンドと比較されて、それぞれのコマンドを実行始まる。しかし、内部コマンドと比較して、同じではない場合、external_command を実行する。

### 2.3 実装方法

1. parse 関数を使てスペースで分けた char 型の配列の配列 args が作られるだけでなく、command_status というコマンドの状態も表すようになる。数字は何の意味か以下のように確認することができる。コマンドが args という形になったら実行する。
   0 : フォアグラウンドで実行
   1 : バックグラウンドで実行
   2 : シェルの終了
   3 : 何もしない

2. 実行の前処理として、実行できるコマンドに変更する。はじめには wildcard_command で、*の記号からカレントディレクトリにある条件を満たすファイルやディレクトリに変更する。次はヒストリに関連する記号で、はじめに「!!コマンド」で!!の記号からヒストリにある前のコマンドに変更する。この後「!!ではない他の!」を見つけてヒストリで前に実行したコマンドを入れ替わる。最後に alias の別名から本当のコマンド言葉に変更する。入れ替わるとき、時々args 内に言葉が増やしたり、スペースだわけられたりするので、connect_args を使ってもう一度全ての string の配列が string に戻して、parse を使って新しい args を作成した。

3. すべての別名がコマンドに変更されるようになるので、save_history 関数でコマンドを history 変数に保存する。このようにヒストリで保存されるコマンドは記入と同じのデータではなく、どの状態でも実行可能コマンドである。

4. execute_command の中には内部コマンドを表す配列 command[] = { ”cd”, ”pushd”, ”dirs”, ”popd”, ”history”,”!!”,”alias”,”unalias”,”prompt”,”pwd”,”rm”,NULL} があった。引数 1 にある「コマンド」がこの内部コマンドと比較されて、どの番号の要素にあるか command_non い保存される。見分けやすいように、switch 関数を使って、コマンドを実行始まる。しかし、内部コマンドと比較して、同じではない場合、external_command を実行する。

## 3 外部コマンドの実行機能

### 3.1 仕様

このシェルは内部コマンド対応するだけでなく、外部コマンドも対応する。進み方として、もしコマンドは内部コマンドではないと確認できたら external_command という関数を通して、実行する。
external_command の関数には入力が 2 つあって、args という部分に分けされたコマンドと command_status というコマンドの状態を表す値である。外部コマンドを実行するとき、fork を使って、子プロセスを作成した。子プロセスで外部コマンドを実行する。もし command_status=0 の場合、フォアグラウンドで実行するため、親プロセスは子プロセスを待たなくて、リターンして関数を終わらせる。しかし、command_status=1 でバックグラウンドで実行する場合、wait 関数を使って親プロセスが子プロセスが終了するまで待つ。

### 3.2 処理の流れ（処理概要）

1. コマンドが内部コマンドと比較して、全て違うの場合、default として、外部コマンドと仮定されて、external_command に入れる。 external_command 関数は args という関数の配列と command_status というコマンドの状態を表す値を入力される。

2. External_command で外部コマンドを実行する。バックグラウンドコマンドを実行する場合はすぐにリターンするがフォアグラウンドの場合はコマンドが実行終了まで待つ。

## 3.3 実装方法

1. 外部コマンドを実行する方法

   **(関数)** external_command

   **(入力)** スペースで分けたコマンドの配列 args[], コマンドの状態を表す値 command_status

   **(出力)** (void)

   (a) 外部コマンドを実行する時、別のプロセスで実行する必要がある。fork() 関数を使って子プロセスと親プロセスに分ける。子プロセスの pid は 0 で、親プロセスの pid は 0 より大きい数字である。

   (b) pid が 0 より小さい場合、これは親プロセスでも子プロセスでもないので「ERROR: Failed to fork child process」を発言して、プロセス終了する。

   (c) pid が 0 では子プロセスの場合なので execvp 関数を使って args に保存する外部コマンドを実行する。戻し値として 0 より小さい場合、コマンドが見つけないで実行できないという意味なので、「command doesnt exist」を発言して、子プロセスを終了する。

   (d) pid が 0 より大きいでは親プロセスの意味なので、コマンドの状態を表す値 (command_status) を確認する。command_status が 0 の場合、フォアグラウンドであるため子プロセスの終了を待つ。しかし、command_status が 1 の場合、バックグラウンドであるため子プロセスが終わるかどうか関係なく関数を抜ける。

## 3.4 テスト

### 3.4.1 テスト方法

External コマンドでは実行できるのかを確認する。もっとも、フォアグラウンドとバックグラウンドに正しく実行できるのかを確認する。

| 条件 | テストケース |
|---|---|
| バックグラウンドの実行確認 | firefox & |
| フォアグラウンドの実行確認 | firefox |

### 3.4.2　テスト結果

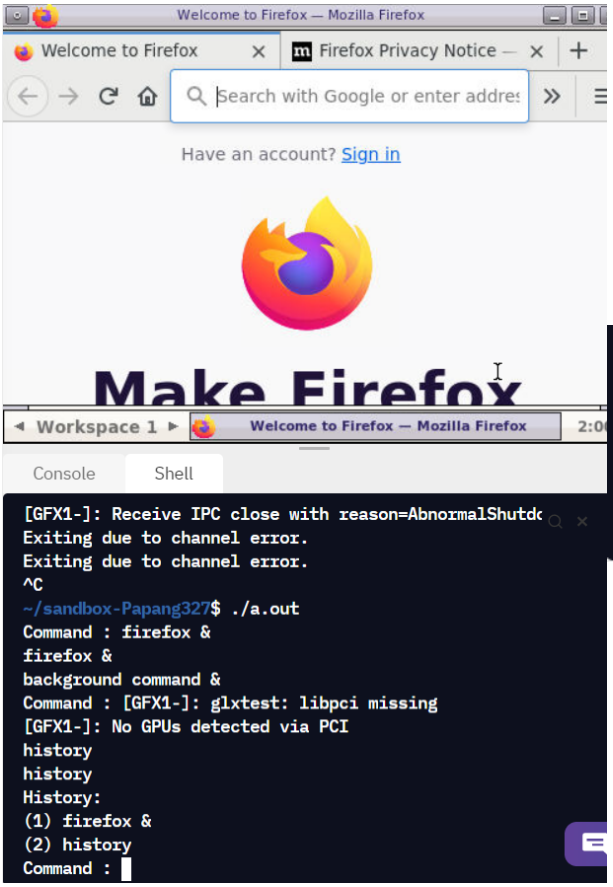


図 1: firefox-バックグラウンド

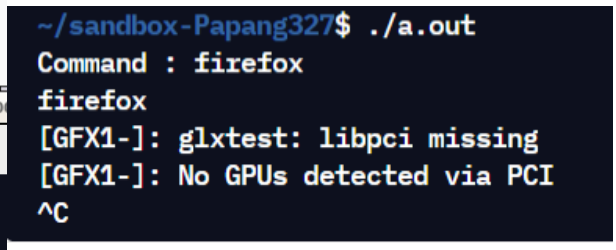


図 2: firefox-フォアグラウンド

図 1 はバックグラウンドで firefox を実行することで、図 2 はフォアグラウンドで firefox を実行する結果である。この 2 つの図で見れるように、フォアグラウンドで実行するときは外部コマンドが実行中の限り新しいコマンドが実行できないが、バックグラウンドで実行する時は外部コマンドがまだ実行終わらなくても他の実行を重ねる。

# 4　ディレクトリの管理機能

## 4.1　仕様

ディレクトリの管理機能では４つのコマンドを実行できた。
**cd コマンド:**　cd は change directory の略として、カレントディレクトリを変更できるようにするコマンドである。このコマンドを実行仕方とは「cd [ディレクトリ名]　」であり、ディレクトリは相対パスか絶対パス関係なくそのディレクトリへカレントディレクトリを移動する。しかし、指定されてない場合、環境変数 HOME に指定されたディレクトリに移動する。そのコマンドを実行する時作成した cd_command 関数が使われる。
**Pushd コマンド：**　pushd は push directory の略として、stack_dir というディレクトリのスタックにカレントディレクトリを保存する。そのコマンドを実行する時作成した pushd_command 関数が使われる。
**Dirs コマンド：**　dirs とは directory stack の略として、stack_dir に保存されるディレクトリをすべて表す。表す順番としてはスタックの上（一番最近入れられて、最初に取り出される方）が一番上にして、先に表す。そのコマンドを実行する時作成した dirs_command 関数が使われる。
**Popd コマンド：**　popd とは pop directory の略として、一番上（一番最近入れられた）のディレクトリを取り出されて、カレントディレクトリがそのディレクトリにする。そして、取り出したディレクトリはスタックから消す。そのコマンドを実行する時作成した popd_command 関数が使われる。

## 4.2　処理の流れ（処理概要）

1. cd コマンド

   (a) cd コマンドが実行っされた場合、まず引数の個数を確認する。cd コマンドの引数の個数は最低１個（コマンドのみ)、最大 2 個（コマンドとディレクトリ）であるので、3 個以上の場合はデータがありすぎて、「Too much data for command [cd]」を出力されて関数終了する。

   (b) しかし、引数が１個の場合、環境変数 HOME にはディレクトリが指定されるのか、またはそのディレクトリが実際に存在するか確認して、問題がある場合には「”The location is not valid」を出力して関数終了する。しかし、問題がない場合はカレントディレクトリを移動される。

   (c) 引数が２個の場合、移動したいディレクトリを実際に存在するかどうかを確認して、存在しない場合には「”The location is not valid」を出力して関数終了する。しかし、問題がない場合はカレントディレクトリを移動される。

   (d) まだ関数終了しない場合にはカレントディレクトリが移動されるという意味なので、新しいカレントディレクトリを出力して、ｃｄコマンドを終了する。

2. pushd コマンド

   (a) pushd コマンドを実行する時、引数の個数は 1 個で、コマンドだけである。しかし、引数が２個以上場合、「Too much data for command [pushd]」が出力されて、関数終了する。

   (b) stack_dir には MAXSTACK(初期=1000) のディレクトリしか保存できないため、保存ディレクトリが MAXSTACK になる場合、もう保存できないので、「The stack is full」を出力して、プログラムを終了する。

   (c) stack_dir がまだ満たさない場合、最後に、入れられたディレクトリはにあるので、カレントディレクトリをスタックに入れる。

3. dirs コマンド

(a) dirs コマンドを実行する時、引数の個数は 1 個で、コマンドだけである。しかし、引数が 2 個以上場合、「Too much data for command [dirs]」が出力されて、関数終了する。

(b) 新しい順からスタックのディレクトリを表示する。

4. popd コマンド

(a) popd コマンドを実行する時、引数の個数は 1 個で、コマンドだけである。しかし、引数が 2 個以上場合、「Too much data for command [popd]」が出力されて、関数終了する。

(b) スタックにはディレクトリがない（last_stack = 0）の場合、「Too much data for command [popd]」が出力されて、関数終了する。

(c) スタックで一番上のディレクトリがカレントディレクトリにして、設定できなかったら「The uppermost directory in slack is not valid」を出力する。

(d) 一番上のディレクトリをスタックから削除する。そして、メモリを解放する。

(e) 確認として、新しいカレントディレクトリを出力する。

## 4.3 実装方法

1. cd コマンド

(a) 環境変数を手に入れるようにする方法 Getenv を使って、環境変数を手に入れる。

**(関数)** Getenv

**(入力)** 環境変数のキー（例：”HOME ”）

**(出力)** キーと繋がってる変数の値（HOME の場合は設定した HOME のディレクトリ）。設定されない場合、NULL を出力する。

(b) カレントディレクトリを移動する方法 Chdir を使って、カレントディレクトリを移動する。

**(関数)** Chdir

**(入力)** 移動したいディレクトリ

**(出力)** ディレクトリの場所が移動できない、存在しない場合、-1 を出力する。

2. pushd コマンド

(a) カレントディレクトリが手に入れる方法
カレントディレクトリの絶対パスを一つ目の引数に入れると共に出力する。二つ目の引数は一つ目の引数の長さである。この行しかカレントディレクトリを使うつもりないので、一つ目の引数が NULL にして、二つ目の引数は長さ 0 で、保存しないようにする。このようにディレクトリがリターンの結果として出力されて、stack_word1 という string に保存される。

**(関数)** Getcwd

**(入力)** カレントディレクトリの絶対パスが保存してほしい変数と変数のサイズ

**(出力)** カレントディレクトリの絶対パスの char 型の配列

(b) カレントディレクトリをスタックに入れる方法

i. ディレクトリのスタック stack_dir はただのポインターの配列なので、ディレクトリを保存する動的メモリが必要になる。ディレクトリの長さは 256 個にすぎないと仮定して、malloc を使って、stack_word1 のポインターで 256 個の char 型のメモリを確保する。

**(関数)** Malloc

**(入力)**　確保したいメモリーのサイズ

**(出力)**　メモリーを確保したポインター

ii. Last_stack ではスタックでディレクトリの個数を数える関数で、stack_dir[last_stack] に保存する。

iii. スタックにあるディレクトリの数が増えたので last_stack の数が１に増える。

3. dirs コマンド

(a) 新しい順からスタックのディレクトリを表示する方法
Stack_dir は一番古いのポジションが 0 であって、一番新しいディレクトリは stack_dir の last_stack-1 にあるため、表す順番としてはスタックの上（一番最近入れられて、最初に取り出される方）が一番上にできるように last_stack-1 から 0 まで出力する。

4. popd コマンド

(a) 一番上のディレクトリをスタックから削除して、メモリを解放する方法。

i. last_stack はスタックにあるディレクトリの数なので、削除するように last_stack から 1 を消して、そのポジションが使わないようにする。

ii. メモリを解放するように free 関数を使う。

**(関数)**　Malloc

**(入力)**　確保したいメモリーのサイズ

**(出力)**　メモリーを確保したポインター

## 4.4　テスト

### 4.4.1　テスト方法

ディレクトリの管理機能のテストできるように、cd,pushd,dirs,popd を確認しないといけない。
cd コマンド：cd コマンドはカレントディレクトリを変更できるようにするコマンドで、相対パスでも絶対パスでも対応する。もっとも、パスが書かなければ環境変数 HOME に移動する。

| 条件 | テストケース |
|---|---|
| 相対パス | cd folder |
| 絶対パス | cd /home/runner/sandbox-Papang327 & |
| パスが書いていない | cd |

Pushd コマンド：pushd コマンドは stack_dir というディレクトリのスタックにカレントディレクトリを保存する。動作確認としてに pushd を使って保存して、結果が正しく動くのかは dirs で確認する。
popd コマンド：popd コマンドは一番上（一番最近入れられた）のディレクトリを取り出されて、カレントディレクトリがそのディレクトリにする。動作確認としてに pushd を使って保存して、結果が正しく動くのかは dirs で確認する。
Dirs コマンド：dirs コマンドは stack_dir に保存されるディレクトリをすべて表す。

| 条件 | テストケース |
|---|---|
| Stack_dir にディレクトリを保存できる | pushd |
| Stack_dir が順番に保存できて、表示できる | cd folder<br>pushd<br>dirs |
| Popd で stack_dir から一番上のディレクトリ<br>を取り出して、カレントディレクトリにする | cd ..<br>popd |
| Stack_dir にディレクトリがない場合、popd コマンド呼ばれても使えない | popd |

#### 4.4.2 テスト結果

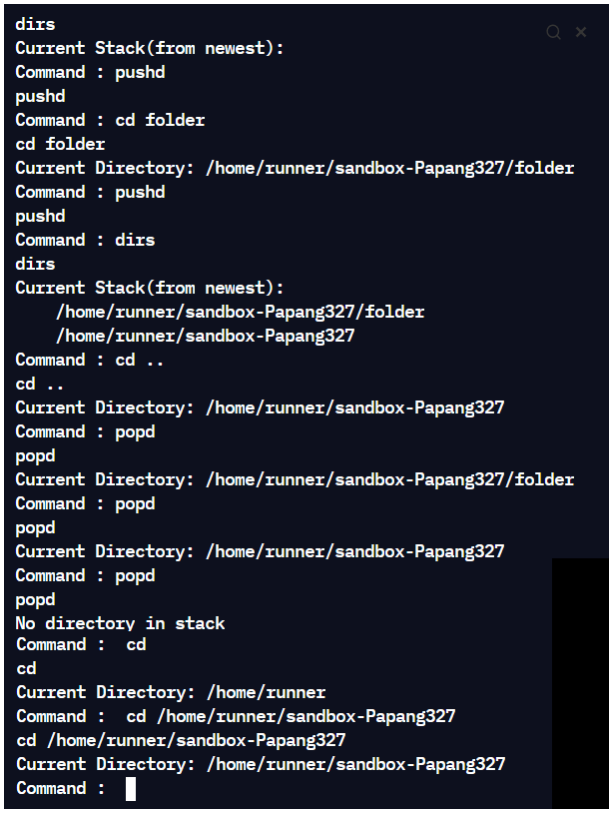


図 3: ディレクトリ機能をテストする結果

図 3 で見れるように、全てのコマンドの実行結果がこのようになる。まとめとして、この結果から cd コマンドは相対パス、絶対パスでも対応して、パスが書かなければ環境変数 HOME に移動できると保証する。stack_dir の動作の確認でも全てがコマンドの説明通りに、実行できる。

# 5 ヒストリー機能

## 5.1 仕様

ヒストリー機能では 3 つのコマンドを実行できた。

**History コマンド：**　history コマンドとはこれまでシェルで実行したコマンドが 32 個まで保存して、実行した順番に表示する。表示形としては番号を付けて古く実行したコマンドから表示する。
**!!コマンド：**　!!コマンドとは１つ前に実行したコマンドをもう一度実行するというコマンドである。再実行コマンドでもヒストリに保存される。 ヒストリ機能の拡張として、!n コマンドでは n の数字で指定した順番に実行したコマンドを再度実行して、!-n コマンドでは n 個前のコマンドを再度実行できるように作成した。
**!string コマンド：**　!string コマンドとは string という文字列で始まる前に実行したコマンドをもう一度実行する。但し、複数ある場合、最新のコマンドを実行する。再実行コマンドでもヒストリに保存される。

## 5.2　処理の流れ（処理概要）

まず、history 配列について紹介する。history 配列とは 32 要素の配列で、それぞれは char 型のポインターである。historyend は int 型の変数で、history に入ったディレクトリは何個目かを数える。はじめに全部は NULL であり、それぞれのメンバーは 0 から 31 個目まで history[0] から history[31] に順番にディレクトリのヘッダーを保存される。しかしディレクトリの 33 個目が入ると、history[0] はのデータはもう要らないので、33 個目のデータが history[0] に保存される。同じ、n 個目のデータが history[(n-1)%32] に入るので、メモリと計算時間が少なくなる。
以下からはプログラムの順番である。

1. まず、コマンドの入力が引数に変更されて実行コマンドが execute_command に入るとき、コマンドは「!!」であるかを確認して、「!!」コマンドだったらコマンドが前に実行したコマンドに入れ替わる。
2. しかし、コマンドは「!!」ではないが「！」から始まる場合、「!string、!n、!-n」コマンドであると仮定して、history_position 関数を使って history にあるどのコマンドと入り変わるのかを決めて、場所が his_po にリターンする。しかし、his_po は-1 の場合、string が始まる 32 個前の history がないという意味で、「Past command according to the condition [%s] cannot be found」を出力して、リターンする。
3. His_po が-1 ではない場合、条件に満たすヒストリがあるので history[his_po] が args に入れ替わることになる。
4. Alias で設定した別名や!!などすべてがコマンドが実行できる形に変更されると、ヒストリを保存する。ヒストリを保存できるように save_history 関数を使って、history 配列に保存される。
5. 次はコマンドを実行する、その時 history コマンドの場合、history_command の関数を使って最大 32 個に実行したコマンドが古い順番に表す。

## 5.3　実装方法

1. !!、!n、!-n、!string に対応するするヒストリの位置を見つかる方法

   **(関数)**　History_position

   **(入力)**　!から始まるコマンド　（例：!-1 !5 !ali）

   **(出力)**　それぞれのコマンドに対応するヒストリの位置。見つけない場合、-1 を出力する。

   History_position 関数の仕方：

   (a) まず、コマンドが string というポインターに指定される。全ての入力は!から始まるので、!を消せるように string++をして、ポインターが次のビットに指定する。

(b) まず、!string を指定するように string とヒストリにあるコマンドを比較する。保存されたコマンドが 32 個より小さい場合、current_position から 0 の位置まで比較する。しかし、32 個以上の場合、一番新しいヒストリの位置は (historyend-1)%32 であるため、current_position =(historyend-1)%32 にして、この位置から 32 個を段々上がりつつある。しかし、0 の次は-1 にしたら問題があるので current_position が３２を足して、３１から (current_position+1) まで比較される。strstr（history[current_position]、string）で比較すると右の引数（string）が左（ヒストリのコマンド）に存在する場合、どの文字から見つかるかをリターンするので、この結果を history[current_position] と引くと０が出したらコマンドのはじめから見つかるということで、!string に対応するヒストリがみつかる。このように current_position をリターンする。

(c) !string で見つけない場合、「!-n」であるかどうかを決める。「!-n」のコマンドになるようにはじめの文字が「-」であることを確認して、string++を使って、残りの文字が数字になる。数字の文字が数字に変更するように string_to_num 関数を使って、num に出力する。しかし、数字以外の文字があれば-1 に提出するので num が-1 の場合、-1 をリターンする。num は 32 より大きいでもヒストリが 32 個まで保存するなので、-1 をリターンする。そうすると、「!-n」が使ってほしいコマンドは今のコマンドより n 個前なので、ヒストリに (historyend-num) というヒストリ個の位置にあると分かる。但し、もし num が historyend より大きい（例：保存されたコマンドは 7 個で n は 20）は対応できないので、-1 をリターンする。これ以外は出力できそうなので、(current_positon%32) をヒストリの位置としてリターンする。

(d) 「!string」ではないし、「!-n」ではない場合、「!n」であるかどうかを確認する。数字の文字が数字に変更するように string_to_num 関数を使って、num に出力する。しかし、数字以外の文字があれば-1 に提出するので num が-1 の場合、-1 をリターンする。num が 32 より大きいでも対応しないので-1 を出力する。historyend はコマンドの全ての個数を数えるので num は historyend より大きいの場合でも対応できないで、-1 をリターンする。h istroyend は 32 より小さい場合、ヒストリは history[0] から始まるので、ヒストリの位置は「num-1」でリターンする。しかし、historyend が 32 より大きいなら (historyend)%2 から始まるので、ヒストリの位置が「(historyend+num-1)%32」でリターンする。

**(関数)** string_to_num

**(入力)** 数字を表す文字

**(出力)** 数字、問題があれば-1

String_to_num の仕方：

(a) 引数が string にして、リターンを保存する変数が num にして、0 を初期化する。

(b) string を文字ずつ読んで、「0-9」である場合、元に num に保存する数が一桁上がるように 10 を掛けて、(string の文字-  ’0  ’) で数字化して、次の string 文字を読み込む。string が全て読み込むと num を出力する。

(c) 「0-9」以外の文字があれば-1 をリターンする。

2. ヒストリを保存する方法

**(関数)** Save_history

**(入力)** スペースで分けたコマンドの配列 args[], コマンドの状態を表す値 command_status

**(出力)** (void)

Save_history の仕方：

(a) connect_arg 関数を使って、スペースで分けたコマンドの配列 args[] から一つの string に変更して、出力する。command_status ＝１というバックグラウンドのコマンドの場合は string の最後に、「&」を付ける。新しい string は buf_histroy に保存する。

(b) string を history[historyend%32] に保存する。

(c) 保存されたコマンドが増やすので、historyend が１を増やす。

3. ヒストリを表示する方法

**(関数)** History_command

**(入力)** コマンド本体　args[]

**(出力)** (void)

History_command の仕方

(a) 引数がコマンド「history」だけで、1 かどうかを確認する。他の引数がある場合、「Too much data for command [history]」を出力して、コマンド実行終了する。

(b) 一番古い値を start にする。基本、start=0 だが、historyend という実行したコマンドこ数が 32 以上の場合、start が historyend%32 になる。

(c) 表示するように、まず「History:」を出力してから、1 行 1 コマンド history[start] から historyend(32 以下の場合)、または 32 個を出力する。コマンドの前に個数を表す。

## 5.4 テスト

### 5.4.1 テスト方法

ヒストリー機能の実行が確認できるように History、!!、!string、!-n、!n コマンドを確認する。
**History コマンド：**history コマンドとはこれまでシェルで実行したコマンドが 32 個まで保存する。
**!!コマンド：**!!コマンドとは１つ前に実行したコマンドをもう一度実行するというコマンドである。再実行コマンドでもヒストリに保存される。
**!string コマンド：**!string コマンドとは string という文字列で始まる前に実行したコマンドをもう一度実行する。但し、複数ある場合、最新のコマンドを実行する。再実行コマンドでもヒストリに保存される。
**!n !-n コマンド：**ヒストリ機能の拡張として、!n コマンドでは n の数字で指定した順番に実行したコマンドを再度実行して、!-n コマンドでは n 個前のコマンドを再度実行できるように作成した。
以下の条件に満たすと確認するように、テストを確認する。

| 条件 | テストケース |
|---|---|
| ヒストリが正しく保存されて、表せる | history |
| !!のコマンドが動ける | !! |
| !-n に問題があるとエラーを出す | !-n (n はヒストリ欄より大きい) |
| !-n のコマンドが動ける | !-n (n はヒストリ欄より小さい) |
| !n に問題があるとエラーを出す | !n (n はヒストリ欄より大きい) |
| !n のコマンドが動ける | !n (n はヒストリ欄より小さい) |
| !string が実行したら一番<br>最近条件を満たすコマンドが出る | !p |
| !string が実行しで条件を満たすコマンドがないとエラーをでる | !qwer |

### 5.4.2 テスト結果

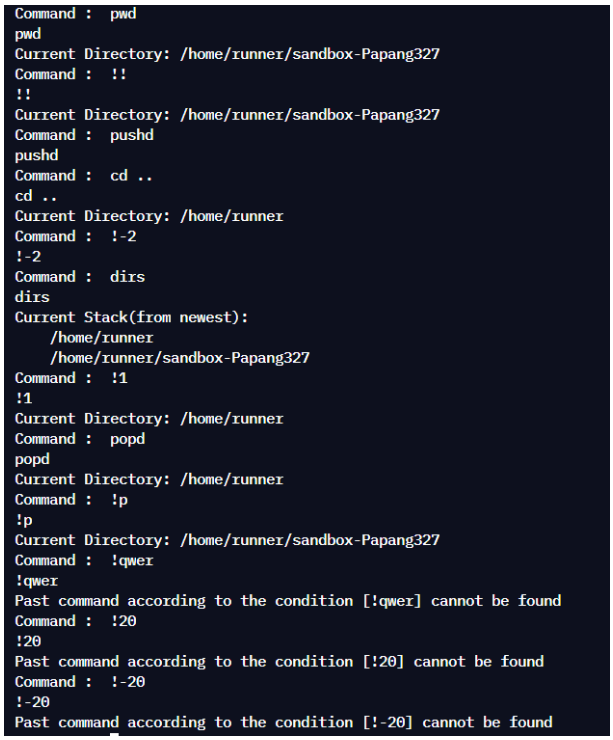


図 4: ヒストリ機能をテストする結果

図 4 で見れる通りに、実行するとコマンドがうまく動ける。ヒストリは数字を付けて、全てのヒストリを古い順で表す。!n,!-n,!string という元に実行したヒストリをもう一度実行するコマンドでも、条件を満たすコマンドがないとエラーを出ることができて、正しく元のコマンドを選んで、再実行できる。

# 6　ワイルドカード機能

## 6.1　仕様

ワイルドカード機能（*）とはコマンドではなく、記号で、その記号が表すと、ディレクトリにあるファイル名に変更する。ワイルドカード機能は 3 つの種類を持つ。**[*]** :　カレントディレクトリにある全てのファイルとディレクトリ名に入れ替わる **[*strings]**:　strings は文字列で、後ろの部分が strings と同じファイル名またはディレクトリ名だけ入れ替わる。**[strings*]**:　strings は文字列で、前の部分が strings と同じファイル名またはディレクトリ名だけ入れ替わる。

## 6.2　処理の流れ（処理概要）

1. スペースで分けたコマンド args のなかで、「*」の文字があるかどうか部分ずつ確認する。
2. 「*」を見つけた args の部分を wildcard_command に入れて、条件が満足したファイル名とディレクトリ名が出力されて、同じ args の部分に入れ替わる。wildcard_command の流れとは先に「*」「string*」「*string」どちらかを*の場所から決めて、string をファイル名に比較する。もし条件内だったら new_args という string に追加して、全てが確認したら new_args を出力する。
3. ファイル名は一つ以上の場合、ファイル名の間がスペースがあるので見分けないといけない。このように connect_args 関数を使って、コマンドが 1 行に戻して、parse 関数を使って、もう一度スペースに分ける。

## 6.3　実装方法

**(関数)**　wildcard_command

**(入力)**　*を持つコマンドの一部

**(出力)**　カレントディレクトリにある条件を満たすファイル名が全てスペースで分ける string

Wildcard_command の仕方

1. まず入力されたコマンドの部分 args は「*」「string*」「*string」であるかを決めて、position 変数が順番に 0,1,2 である。まず strcmp を使って、「*」の場合なら ”* ”と同じであるため、区別できる。しかし「*string」の場合、*は一つ目文字であるため strstr を使って出力が args のポインターの部分である。最後に、前の条件を満たさないが、*が持つには「string*」である。
2. readdir(opendir(“ . ”)) を使って、一つずつのディレクトリ内にあるファイル名やディレクトリが directory に表す。
3. directory-¿d_name はディレクトリまたはファイル名であるため、隠しファイルが表してほしくないので strcmp を使って、「.」がファイルのはじめの文字にある場合、continue で次のファイルをみる。
4. new_args が条件を満たすファイル名を保存する。はじめには何も入れていない。問題がなければ、position=0 の場合はすぐに new_args に追加する。position=1 の場合、見つけた初めの文字の場所と directory-¿d_name のはじめのアドレスではファイル名の長さと args の長さの差と同じ場合、、これは正しく後ろの部分は同じなので new_args に追加する。しかし、position=2 の場合ファイル名のはじめのアドレスと strstr を使って比較して見つかったアドレスは同じ場合、前の方と同じであるので new_args に追加する。

5. 毎回ファイル名を追加すると、スペースを追加する。

6. 全てのファイルが確認したら new_args をリターンする。

## 6.4 テスト

### 6.4.1 テスト方法

Wildcard 機能とは*という記号が表すと、ディレクトリにあるファイル名に変更する。[*] そのままの場合はカレントディレクトリにある全てのファイル名に入れ替わる。しかし、*の前や後ろに string が付いた場合、前または後ろの部分が strings と同じファイルとディレクトリ名だけ入れ替わる。以下の条件に満たすように確認する。rm と共に確認して、条件を満たすファイルを消す。

| 条件 | テストケース |
|---|---|
| *があったら気づけて、消す。 | rm * |
| string*のコマンドが動ける | rm fil* |
| *string のコマンドが動ける | rm *1.txt |

### 6.4.2 テスト結果

```
Command : cd folder
cd folder
Current Directory: /home/runner/sandbox-Papang327/folder
Command : rm *
rm *
 [file1] is deleted successfully
 [file2] is deleted successfully
 [file3] is deleted successfully
 [file4] is deleted successfully
Command : cd ..
cd ..
Current Directory: /home/runner/sandbox-Papang327
Command : ls
ls
a.out      file2.txt  film1.txt  main.c
bile1.txt  file3.txt  folder     script.txt
Command : rm fil*
rm fil*
 [file3.txt] is deleted successfully
 [file2.txt] is deleted successfully
 [film1.txt] is deleted successfully
Command : ls
ls
a.out  bile1.txt  folder  main.c  script.txt
Command : rm *1.txt
rm *1.txt
 [bile1.txt] is deleted successfully
Command :
```

図 5: wildcard 機能をテストする結果

図 5 で見れる通り、rm *を実行するとき、フォルダーにある「file1,file2, file3, file4」の全てが消された。rm fil*を実行するとき ls で表す通り、フォルダーには他のファイルがあっても「file3.txt file2.txt file1.txt」だけ消される。rm *1.txt を実行するとき ls で表す通り、フォルダーには他のファイルがあっても「bile1.txt」だけ消される。このように、wildcard 機能が正しく実行されると評価できる。

# 7　プロンプト機能

## 7.1　仕様

プロンプト機能にはプロンプトコマンドを実行できた。Prompt コマンドは指定した言葉がプロンプトを変更する。はじめや文字列が指定しない場合、デフォルトとして ”Command: ” に変更する。

## 7.2　処理の流れ（処理概要）

1. コマンドが「prompt」の場合、prompt_command を実行はじめる。word_command にはプロンプトの言葉が保存する。
2. コマンドの内容「prompt」が 2 個以上の場合、データが入力すぎるなので、リターンする。
3. コマンドは 1 個「prompt」だけの場合、指定した言葉がないのでデフォルトを使う。word_command が ”Command : ”に変更する。
4. コマンドは 2 個、「prompt (word)」の場合、指定した言葉があるので、指定した言葉が word_command に入れ替わる。

## 7.3　実装方法

1. プロンプトを呼ばれるのは main 関数にあるため word_command というプロンプトの言葉が保存する変数はグローバル変数にする。コマンドが「prompt」の場合、prompt_command を実行はじめる。
2. prompt_command の引数はスペースで分けるコマンドの内容 args で、「prompt」には args[0] にあって、指定した言葉が args[1] にあるため、args[2] が NULL ではない場合、データが入力すぎるという意味なので、「Too much data for command [prompt]」を出力する。
3. コマンドは 1 個の場合、デフォルトに設定するという意味なので word_command が ”Command : ” に変更する。
4. コマンドは 2 個の場合、指定した言葉があるので、指定した言葉が word_command に入れ替わる。

## 7.4　テスト

### 7.4.1　テスト方法

Prompt コマンドは指定した言葉がプロンプトを変更する。確認するケースは以下の三つである。

| 条件 | テストケース |
|---|---|
| 指定した言葉がプロンプトに変更できる | prompt cat>> |
| 指定しない場合、「command :」になる | prompt |
| 引数がありすぎて、エラーをだして実行終了 | prompt cat dog |

#### 7.4.2　テスト結果

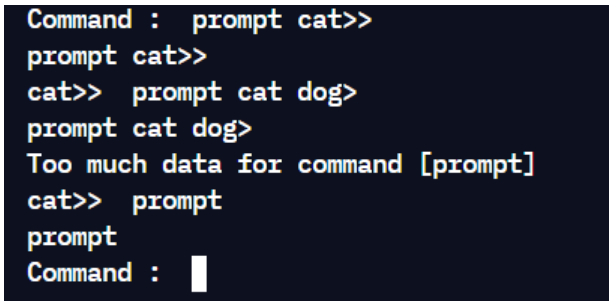


図 6: プロンプト機能をテストする結果

図 6 で見れる通りに、prompt コマンドと共に言葉が出したらその言葉がプロンプトになる。しかし、言葉が出さない場合、プロンプトがデフォルトの「command: 」になる。引数がありすぎるとエラーで実行終了になる。このように prompt が正しく実行できるとまとめられる。

# 8　スクリプト機能

## 8.1　仕様

スクリプト機能とはファイルを標準入力から読み込んで実行することである。ファイルはテキストファイルで、1 行ごとに 1 つのコマンドを記入する。全部読み込むと、正常終了する。これは他の関数と違って関数ではないので、実行する方法としてじゃシェルを実行するとき作成したシェル [例：a.out] の時からテキストファイルをいれる。
形は次のようにである。

「./a.out <file1.txt」

但し、file1.txt はコマンドをもったテキストファイルである。

## 8.2　処理の流れ（処理概要）

1. EOF（ファイルの終了）が毎回確認できるように、読み込めるように標準入力と同じく 1 行を 読み込むではなく、一文字ずつを読み込む。EOF になったらファイル全体の終了である。文字が command_buffer に入れられて、行に変更される。common_len はポインターで一番最後にまだ使わない箱である。

2. 文字が行の終了を表す文字ではない場合、command_buffer[command_len] に入れられて command_len が一個増やすので command_len に一を足す。

3. 行終了を表す文字「\0」「\n」がある場合、余裕としてもう一つのスペースを入れてから ’\n ’を入れて 1 行にできるようになる。文字が「EOF」でもこれは行終了を表す文字と同じく一つのスペースを入れてから ’\n ’を入れて 1 行にできるようになるが、次の行がなくて、実行終了する。できた行は command の string として前と同様に使われている。

## 8.3 実装方法

1. EOF（ファイルの終了）が毎回確認できるように、読み込めるように標準入力同じく 1 行を 読み込むではなく、一文字ずつを読み込む。EOF になったらファイル全体の終了である。文字が command_buffer に入れられて、行に変更される。common_len はポインターで一番最後にまだ使わない箱である。

2. 文字が「\0」「\n」「EOF」ではない場合、command_buffer[command_len] に入れられて command_len が一個増やすので command_len に一を足す。

3. 行終了を表す文字「\0」「\n」がある場合、余裕としてもう一つのスペースを入れてから ’\n ’を入れて 1 行にできるようになる。文字が「EOF」でもこれは行終了を表す文字と同じく一つのスペースを入れてから ’\n ’を入れて 1 行にできるようになるが、次の行がなくて、実行終了する。できた行は command の string として前と同様に使われている。

## 8.4 テスト

### 8.4.1 テスト方法

スクリプト機能とはファイルを標準入力から読み込んで実行することである。このように準備したファイルを読み込まれて、正しく動けるかどうかを確認する。

```
pushd
alias cdd cd
alias rekishi history
alias
cdd folder1
unalias rekishi
rekishi
history
dirs
popd
pwd
!!
!hi
*
prompt meirei>
!-10
!1
rm fi*
rm *1.txt
history
```

図 7: script.txt の内容

図 7 にあるのは script.txt の内容である。他のコマンドも共に確認できるようにこのプログラムを実行できたコマンドが全て入れる。

### 8.4.2　テスト結果

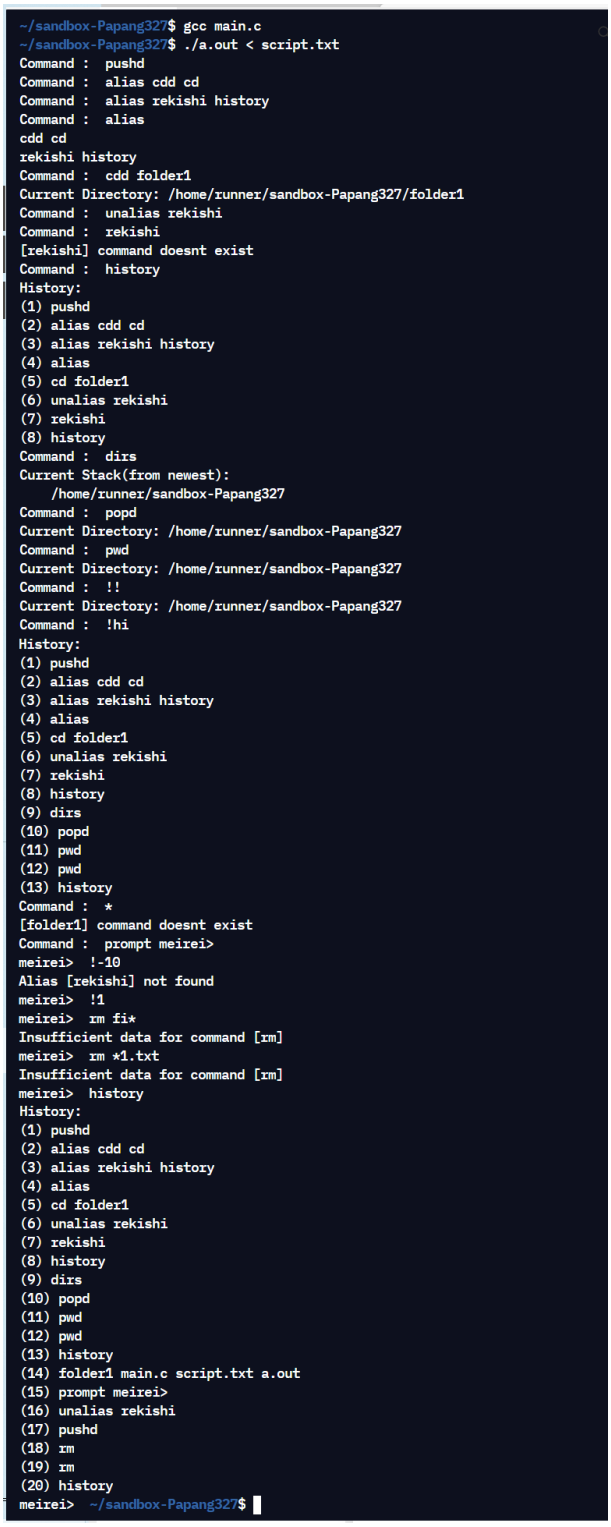


図 8: スクリプト機能をテストする結果

図 8 で見れる通り、プログラムは実行するときスクリプトファイルを読み込んで、正しく実行する。全て実行し終わると自動的にプログラム終了する。

# 9 エイリアス機能

## 9.1 仕様

エイリアス機能には alias と unalias という二つのコマンドが実行できる。**Alias コマンド：** コマンドの別名が設定できるようにするコマンドである。しかし、設定する言葉がない場合、現在登録された alias の言葉が一覧に表示する。**unalias コマンド：** 別名設定された言葉が解除するコマンドである。

## 9.2 処理の流れ（処理概要）

1. Alias コマンド
   (a) 入力されたのは [alias] のコマンドである場合、alias_command に入って、実行はじめる。
   (b) 引数がコマンドだけ (一個) の場合、現在登録した言葉が表す。alias の別名と元のコマンドが行ごとに表して、リターンする。
   (c) alias の引数が表示以外場合は３つが必要で、コマンド、別名、元の名前である。このように引数が２つである場合は「Insufficient data for command [alias]」を出力して、実行終了する。引数が 4 つの場合でも引数があり過ぎるなので「Too much data for command [alias]」を出力して、実行終了する。
   (d) last_alias は登録した alias の個数を数える変数で、もし last_alias はもう MAXALIAS と同じ場合、ストレージがもういっぱいなので「The alias storage is full: delete some aliases first」を出力して、実行終了する。
   (e) 以上の問題がなければ新しい alias 名を保存する。まず前にある別名と比較して、もし同じならキーワードの言葉が追加しないまま新しいコマンドの意味に変更して、実行終了する。
   (f) 前にある別名と比較すると、同じ別名でないなら新しい言葉を作る。
2. Unalias コマンド
   (a) 入力されたのは [unalias] のコマンドである場合、alias_command に入って、実行はじめる。
   (b) 引数が２つではないとき、args[1] が NULL の場合、「Insufficient data for command [unalias]」を出力して、args[2]!=NULL の場合は「Too much data for command [unalias]」を出力して、実行を終了する。
   (c) しかし、引数が２の場合、ではその alias_word[i][0] の別名の配列を登録した言葉に探す。探し終わったらその場所を削除して、数が大きいの方がカバーする。例えば、alias_word[i][0] が削除される場合、alias_word[i+1][0] が alias_word[i][0] になり、alias_word[i+2][0] が alias_word[i+1][0] に移動されて、こういう形で、alias_word[last_alias][0] が alias_word[last_alias-1][0] に移動される。
   (d) 見つけない場合、「Alias [別名] not found」を出力して、実行を終了する。

## 9.3 実装方法

1. Alias コマンド

(a) args[1] が NULL の場合、別名が入力されない意味で、一覧に表示する。エイリアスのペアが alias_word[i][0] と alias[i][1] という形に保存されるので、i は 0 から last_alias という alias の個数の前ま 1 行 2 言葉で出力する。

(b) しかし、args[2] が NULL の場合、別名があって、元のコマンドがないので「Insufficient data for command [alias]」を出力して、実行終了する。args[3] が NULL ではない場合には引数が 4 個以上で、あり過ぎるなので「Too much data for command [alias]」を出力して、実行終了する。

(c) last_alias は登録した alias の個数を数える変数で、もし last_alias はもう MAXALIAS と同じ場合、ストレージがもういっぱいなので「The alias storage is full: delete some aliases first」を出力して、実行終了する。

(d) 以上の問題がなければ新しい alias 名を保存する。まず last_alias 個数の alias_command[0][i] にある別名と strcpy を使って比較して、もし同じ別名があればなら元のコマンドだけ変更して、実行終了する。

(e) 前にある別名と比較すると、同じ別名でないなら新しい言葉を作る。やりかたとしては新しい単語とコマンドの意味を malloc でメモリを確保する。そして alias_word[last_alias][0] が別名に指定されて、alias_word[last_alias][1] が意味を指定する。alias の個数が増やすので last_alias に 1 を足す。

2. Unalias コマンド

(a) 入力されたのは [unalias] のコマンドである場合、alias_command に入って、実行はじめる。

(b) 引数が 2 個ではない場合、一個より以下は「Insufficient data for command [unalias]」を出力して一個以上は「Too much data for command [unalias]」を出力して、実行を終了する。

(c) しかし、引数が 2 の場合、ではその別名を登録した言葉に探す。探し終わったらその場所を削除して、他の場所が空いた場所に入れ込む。

(d) 見つけない場合、「Alias [別名] not found」を出力して、実行を終了する。

## 9.4 テスト

### 9.4.1 テスト方法

Alias 機能の確認として alias と unalias という二つのコマンドを確認する。
**Alias コマンド：** コマンドの別名が設定できるようにするコマンドである。しかし、設定する言葉がない場合、現在登録された alias の言葉が一覧に表示する。
**Unalias コマンド：** 別名設定された言葉が解除するコマンドである。

| 条件 | テストケース |
|---|---|
| Alias は引数が足りない場合、エラーを出してプログラム終了 | alias cat |
| 別名とコマンドが alias に追加できる | alias rekishi history |
| 既に別名である言葉がまた定義されても新しく変更できる | alias current_dir pwd<br>alias current_dir<br>cd |
| Unalias は存在した別名の定義を消せる | unalias current_dir |
| 追加した alias が表せる | alias |
| 追加した別名が実行できる | rekishi |
| Unalias は他の引数がない場合、エラーを出て実行終了。 | unalias |
| Unalias で消したい別名存在しない場合知らせる | unalias new_command |

### 9.4.2 テスト結果

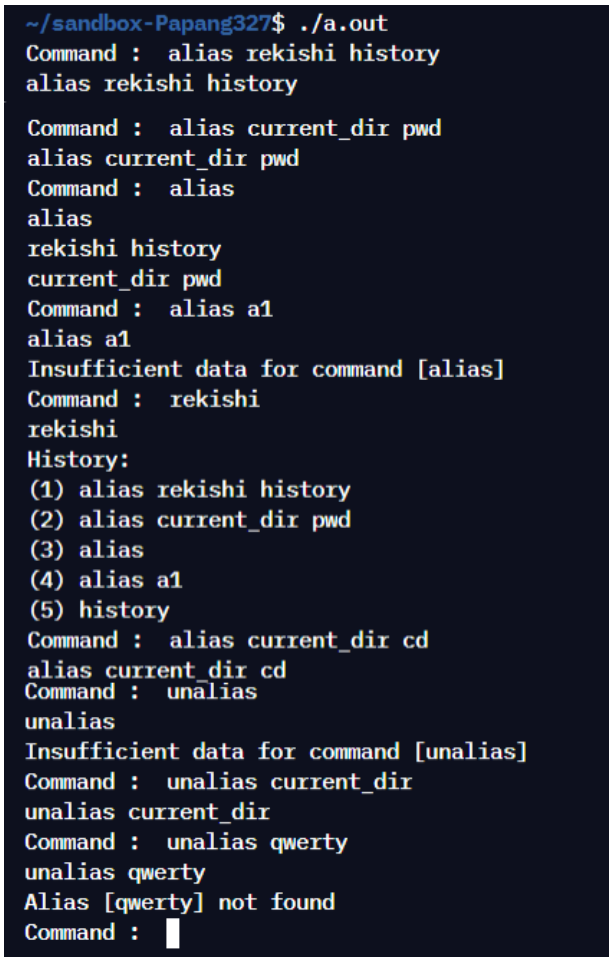


図 9: エイリアス機能をテストする結果

図 9 で表す通り、alias が別名と定義と一緒に記入されると新しく alias 欄に入れかわる。しかし、前に定義された別名がもう一度定義される場合、新しい意味定義される。alias だけであって他の引数がない場合、全て保存される別名と定義が表示される。unalias は別名を消すことができる。しかし、別名が alias 欄にない場合は知らせる。alias と unalias の引数や足りない場合でも、エラーをだして次のコマンドに尋ねる。

# 10　自分で考えた機能

## 10.1　仕様

ディレクトリの管理機として、pwd と rm を作成した。
**Rm コマンド：**　rm は remove を略で、指定した無限数のディレクトリにあるファイルを削除する。しかし、フォルダーを削除できるようにコマンドの 2 個目に「-r」をオプションとして入力されないといけない。
**pwd コマンド：**　pwd コマンドではそのまま書いて、カレントディレクトリの絶対パスを出力するコマンドである。

## 10.2　処理の流れ（処理概要）

1. Rm コマンド：

   (a) 入力されるコマンドが「rm」の場合、rm_command に実行される。

   (b) 入力されるのは「rm」だでか「rm -r」である場合、「Insufficient data for command [rm]」を出力して実行終了する。

   (c) もし二つ目の引数は「-r」である場合、ファイル名が start = 2 から始まる。しかし、「-r」がない場合、ファイル名が start = 1 から始まる。args[start] から最後の引数までファイルが存在かどうか確認して、消す。しかし、消す前にこのデータはファイルかフォルダーか確認して、start = 2 ではない場合はフォルダーを消すことができない。

## 10.3　実装方法

1. Rm コマンド

   (a) 入力されるコマンドが「rm」の場合、rm_command に実行される。

   (b) 引数が 1 個しかないや 2 個で、2 個目は「ー r」である場合、「Insufficient data for command [rm]」を出力して実行終了する。

   (c)　もし二つ目の引数は「-r」である場合、ファイル名が start = 2 個目の引数から始まる。しかし、「-r」がない場合、ファイル名が start = 1 個目の引数から始まる。args[start] からこのデータがファイルかディレクトリか確認して、ディレクトリの場合は start=2 だけ消すことが行う。remove 関数を使ってファイルが成功に消せたら 0 をだして、「[ファイル名] is deleted successfully」を出力して、次のファイルに移動する。しかし、出力が 0 ではない場合、「[%s] is not deleted」を出力する。しかし、start=1 でディレクトリの場合、「Directory [ディレクトリ名] cannot be deleted」を出力して次のファイルに移動する。

## 10.4　テスト

### 10.4.1　テスト方法

自分で考えた機能とは pwd コマンドと rm コマンドである。pwd コマンドとはカレントディレクトリを表示することができるコマンドで、rm コマンドとはオプションによって、無限のデータで消すことができる。二つのコマンドが実行できるのを確認する。

| 条件 | テストケース |
|---|---|
| pwd は実行して、正しい結果がでる | pwd |
| rm は引数がない場合、エラーを出る | rm |
| Rm -r ではがない場合、エラーを出る | rm -r |
| rm は一つのファイル消せる | rm file1.txt |
| rm は複数のファイルを消せる | rm file* |
| Rm でフォルダーに消せない | rm folder |
| Rm -r ではフォルダーを消せる | rm -r folder |

### 10.4.2 テスト結果

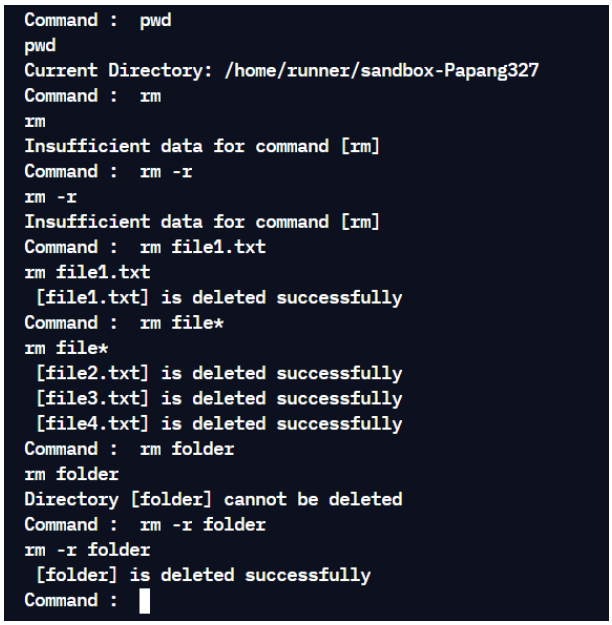


図 10: 自分で考えた機能をテストする結果

図 10 で見れる通り、pwd はカレントディレクトリを出力することができる。rm の場合、は-r が付けないと、ファイルのみ消せる。-r が付けるとファイルでもフォルダでも消せるようになる。これは一つのファイルで複数のファイルでも対応できる。しかし、引数でファイル名がないとエラーをでてプログラム終了にする。

## 11 工夫点

1. ヒストリとワイルドカードの基本機能を実現するだけでなく、ヒストリ機能の拡張やワイルドカードの拡張もやり終わった。

2. ヒストリ機能で、history のディレクトリを保存する方法は前のプログラムでは普通に history[0] から histroy[31] に入れて、ディレクトリの 33 個目が保存できるようにすべてのディレクトリが前の場所に移動して、history[n] が history[n-1] に移動してから新しいディレクトリが history[31] に入ったが、気づいたのは毎回の実行で 32 行の計算が増やしてしまったので、代わりに数学を使って history[(n-1)%32] に入れると保存するのは計算一回しかないで、これは自慢の工夫点である。

# 12　考察

図 11 では外部コマンドを実行するとき、何か起こるのかを表す。以下の考察では fork 関数とフォアグラウンドとバックグラウンドを説明する。

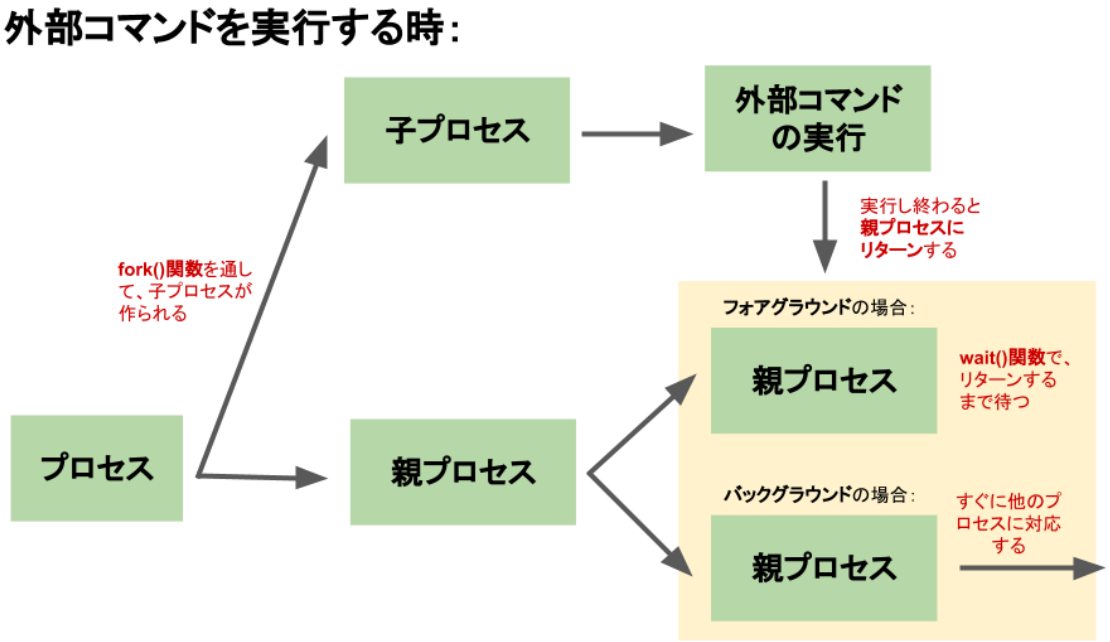


図 11: 外部コマンドで実行するプロセス

**fork() 関数**

Fork 関数は新しいプロセスを作るための関数であり、前のプロセスは「親プロセス」と呼ばれ、新しく作ったプロセスは「子プロセス」である。外部コマンドを実行するときは子プロセスを使って実行する。fork 関数の入力がないで、出力はプロセスによって違う。以下はそれぞれ出力の意味である。

0:　このプロセスは子プロセスである。

正の整数：　親プロセスであるという意味で、この数字は子プロセスのプロセス ID である。

不の整数：　子プロセスの作成は失敗である。

**フォアグラウンドとバックグラウンド**

LINUX にはコマンドの後ろに「&」が付く場合、バックグラウンドで実行するの意味である。フォアグラウンドで実行する場合は「&」をついてない。このプログラムではバックグラウンドで実行するコマンドは command_status で 1 を表して、フォアグラウンドに実行するコマンドは command_status が 0 である。バックグラウンドとフォアグラウンドの違いは何かというと、フォアグラウンドのコマンドが実行し終わるまで親プロセスが待つが、バックグラウンドのコマンドはコマンドを実行途中他のコマンドを実行することができる。

## 13　感想

今学期の課題量が凄く重いでした。単位が少ないで、今学期は軽い方ではないかなと思って、ProC ではなく、実験 A、PBL などの課題量が重ねて、つらかったです。日本語でレポートを書くのも話したいことがうまく伝われなく、適当な言葉が思い浮かべなくて、悔しかったです。
しかし、毎回プログラミングするとき、それぞれの関数が実行成功するとき、やっと勉強するものが理解できるときは勝つ気持ち一杯で、嬉しかったです。この気持ちが忘れないで、次の学期でも、未来でも頑張りつつができたらいいなと思います。
今年の授業、ありがとうございました。色々な学ぶことができて嬉しかったです。来学期でプログラミングも日本語ももっとうまくできるように頑張りたいと思います。

## 14　参考文献

Difference Between Process, Parent Process, and Child Process - GeeksforGeeks. (2021). Retrieved 15 July 2021, from https://www.geeksforgeeks.org/difference-between-process-parent-process-and-child-process/

fork() in C - GeeksforGeeks. (2021). Retrieved 15 July 2021, from https://www.geeksforgeeks.org/fork-system-call/

# 15 添付

## 15.1 プログラムリスト

```
/*-------------------------------------------------------------------------
*  簡易版シェル
*-------------------------------------------------------------------------*/

/*
*  インクルードファイル
*/

#include <stdio.h>
#include <stdlib.h>
#include <sys/types.h>
#include <unistd.h>
#include <sys/wait.h>
#include <string.h>
#include <dirent.h>
#include <sys/stat.h>
#include <stdbool.h>
/*
*  定数の定義
*/
#define MAXSTACK 1000
#define MAXALIAS 1000
#define BUFLEN    1024     /* コマンド用のバッファの大きさ */
#define MAXARGNUM  256     /* 最大の引数の数 */
char *alias_word[MAXALIAS][2];
char *stack_dir[MAXALIAS];
char *history[32];
int historyend=0;
int last_alias  = 0;
int last_stack = 0;
char word_command[256] = "Command :";
/*
*  ローカルプロトタイプ宣言
*/
bool is_file(const char* path);
bool is_dir(const char* path);
int string_to_num(char* string);
int history_position(char arg[]);
int parse(char [], char *[]);
char* connect_args(char* args[],int command_status);
void save_history(char *args[],int command_status);
void execute_command(char *[], int);
```

```
void cd_command(char *args[]);
void pushd_command();
void dirs_command();
void popd_command();
void history_command(char *args[]);
void exc2_command(char *args[]);
void alias_command(char *args[]);
void unalias_command(char *args[]);
void prompt_command(char *args[]);
void pwd_command(char *args[]);
char* wildcard_command(char* args);
void script_command(char *args[]);
void rm_command(char *args[]);
void external_command(char *args[],int command_status);/*------------------------------
*
*  関数名    : main
*
*  作業内容 : シェルのプロンプトを実現する
*
*  引数      :
*
*  返り値    :
*
*  注意      :
*
*-------------------------------------------------------------------------*/
bool is_file(const char* path) {
   struct stat buf;
   stat(path, &buf);
   return S_ISREG(buf.st_mode);
}

bool is_dir(const char* path) {
   struct stat buf;
   stat(path, &buf);
   return S_ISDIR(buf.st_mode);
}
int main(int argc, char *argv[])
{
   char command_buffer[BUFLEN]; /* コマンド用のバッファ */
   char *args[MAXARGNUM];       /* 引数へのポインタの配列 */
   int command_status;          /* コマンドの状態を表す
                                   command_status = 0 : フォアグラウンドで実行
                                   command_status = 1 : バックグラウンドで実行
                                   command_status = 2 : シェルの終了
                                   command_status = 3 : 何もしない */
```

```
alias_word[0][0] = NULL;
alias_word[0][1] = NULL;
/*
 *  無限にループする
 */

for(;;) {

    /*
     *  プロンプトを表示する
     */
    printf("%s  ",word_command);
    /*
     *  標準入力から１行を command_buffer へ読み込む
     *  入力が何もなければ改行を出力してプロンプト表示へ戻る
     */
    char c,command_len=0;
    if(c==EOF) break;
    while((c=getchar())!=EOF){
      if(c=='\n' || c=='\0') {
        command_buffer[command_len]=' ';
        command_len++;
        command_buffer[command_len]='\0';
        command_len=0;
        break;
      }
      else{
      command_buffer[command_len]=c;
      command_len++;
      command_buffer[command_len]='\0';
      }
    }
    if(c==EOF) {
        command_buffer[command_len]=' ';
        command_len++;
        command_buffer[command_len]='\0';
        command_len=0;
      }
    printf("%s\n",command_buffer);
    /*
     *  入力されたバッファ内のコマンドを解析する
     *
     *  返り値はコマンドの状態
     */
    command_status = parse(command_buffer, args);
    /*
```

```
         *  終了コマンドならばプログラムを終了
         *  引数が何もなければプロンプト表示へ戻る
         */

        if(command_status == 2) {
            printf("done.\n");
            exit(EXIT_SUCCESS);
        } else if(command_status == 3) {
            continue;
        }
        /*
         *  コマンドを実行する
         */
        execute_command(args, command_status);
    }

    return 0;
}

/*----------------------------------------------------------------------------
*
*  関数名    : parse
*
*  作業内容 : バッファ内のコマンドと引数を解析する
*
*  引数      :
*
*  返り値    : コマンドの状態を表す :
*                0 : フォアグラウンドで実行
*                1 : バックグラウンドで実行
*                2 : シェルの終了
*                3 : 何もしない
*
*  注意      :
*
*----------------------------------------------------------------------------*/

int parse(char buffer[],         /* バッファ */
          char *args[])          /* 引数へのポインタ配列 */
{
    int arg_index;    /* 引数用のインデックス */
    int status;    /* コマンドの状態を表す */

    /*
     *  変数の初期化
     */
```

```
arg_index = 0;
status = 0;

/*
 *  バッファ内の最後にある改行をヌル文字へ変更
 */

*(buffer + (strlen(buffer) - 1)) = '\0';

/*
 *  バッファが終了を表すコマンド（"exit"）ならば
 *  コマンドの状態を表す返り値を 2 に設定してリターンする
 */

if(strcmp(buffer, "exit") == 0) {

    status = 2;
    return status;
}

/*
 *  バッファ内の文字がなくなるまで繰り返す
 *  （ヌル文字が出てくるまで繰り返す）
 */

while(*buffer != '\0') {

    /*
     *  空白類（空白とタブ）をヌル文字に置き換える
     *  これによってバッファ内の各引数が分割される
     */
    while(*buffer == ' ' || *buffer == '\t') {
        *(buffer++) = '\0';
    }

    /*
     * 空白の後が終端文字であればループを抜ける
     */

    if(*buffer == '\0') {
        break;
    }

    /*
     *  空白部分は読み飛ばされたはず
```

```
         *  buffer は現在は arg_index + 1 個めの引数の先頭を指している
         *
         *  引数の先頭へのポインタを引数へのポインタ配列に格納する
         */

        args[arg_index] = buffer;
        ++arg_index;

        /*
         *  引数部分を読み飛ばす
         *  （ヌル文字でも空白類でもない場合に読み進める）
         */

        while((*buffer != '\0') && (*buffer != ' ') && (*buffer != '\t')) {
            ++buffer;
        }
    }

    /*
     *  最後の引数の次にはヌルへのポインタを格納する
     */

    args[arg_index] = NULL;

    /*
     *  最後の引数をチェックして "&" ならば
     *
     *  "&" を引数から削る
     *  コマンドの状態を表す status に 1 を設定する
     *
     *  そうでなければ status に 0 を設定する
     */

    if(arg_index > 0 && strcmp(args[arg_index - 1], "&") == 0) {

        --arg_index;
        args[arg_index] = '\0';
        status = 1;

    } else {
        status = 0;
    }

    /*
     *  引数が何もなかった場合
     */
```

```
    if(arg_index == 0) {
        status = 3;
    }

    /*
     *  コマンドの状態を返す
     */

    return status;
}

/*-------------------------------------------------------------------------
*
*  関数名    : execute_command
*
*  作業内容 : 引数として与えられたコマンドを実行する
*              コマンドの状態がフォアグラウンドならば、コマンドを
*              実行している子プロセスの終了を待つ
*              バックグラウンドならば子プロセスの終了を待たずに
*              main 関数に返る（プロンプト表示に戻る）
*
*  引数      :
*
*  返り値    :
*
*  注意      :
*
*-------------------------------------------------------------------------*/

void execute_command(char *args[],     /* 引数の配列 */ int command_status)     /* コ
マンドの状態 */{
    int status;    /* 子プロセスの終了ステータス */
    char *command[]  = { "cd", "pushd", "dirs", "popd", "history","!!","alias","unalias",
      /*
      * check for command wildcard
      */
      for(int i=0;args[i]!=NULL;i++){
        if(strstr(args[i],"*")!=NULL){
          strcpy(args[i],wildcard_command(args[i]));
        }
      }
      char *command_buffer = connect_args(args,command_status);
      command_status = parse(command_buffer, args);
      /*
      *   check for command !!
```

```
  */
  if(strcmp(args[0],command[5])==0){
    char command_buffer[BUFLEN];
    strcpy(command_buffer,history[(historyend-1)%32]);
    command_status = parse(command_buffer, args);
  }
  /*
  *   check for command !string !n !-n
  */
  else if(*args[0]=='!'){
    char command_buffer[BUFLEN];
    int his_po = history_position(args[0]);
    if(his_po<0){
      printf("Past command according to the condition [%s] cannot be found\n",args[0]
      return;
    }
    strcpy(command_buffer,history[his_po]);
    command_status = parse(command_buffer, args);
  }
  /*
  convert word from alias to be in command form
  */
  if(strcmp(args[0],"alias")!=0){
    for(int i=0;alias_word[i][0]!=NULL;i++){
      if(strcmp(alias_word[i][0],args[0])==0){ //if alias[0][i]=args[0]
          strcpy(args[0],alias_word[i][1]);
          break;
      }
    }
  }
  /*
  save command in history
  */
  save_history(args,command_status);
/*char *command[] = { "cd", "pushd", "dirs", "popd", "history","!!","alias","unalias"
int command_no = -1;
for(int i=0;command[i]!=NULL;i++){
  if(strcmp(args[0],command[i])==0) command_no= i;
}
switch(command_no){
  case 0: cd_command(args);break;
  case 1: pushd_command(args);break;
  case 2: dirs_command(args);break;
  case 3: popd_command(args);break;
  case 4: history_command(args);break;
  case 5: break;
```

```
      case 6: alias_command(args);break;
      case 7: unalias_command(args);break;
      case 8: prompt_command(args);break;
      case 9: pwd_command(args);break;
      case 10: rm_command(args);break;
      default: external_command(args,command_status);break;
    }
}

void external_command(char* args[],int command_status){
     /*
     *  子プロセスを生成する
     *  生成できたかを確認し、失敗ならばプログラムを終了する
     */
    /******** Your Program ********/
    /* プロセスＩＤ */
    int pid= fork(); //pid=0(child) >0(parent)
    if(pid < 0 ){
      fprintf(stderr,"ERROR: Failed to fork child process \n");
      _exit(0);
    }
    /*
     *  子プロセスの場合には引数として与えられたものを実行する
     *  引数の配列は以下を仮定している
     *  ・第１引数には実行されるプログラムを示す文字列が格納されている
     *  ・引数の配列はヌルポインタで終了している
     */
    /******** Your Program ********/
    if(pid==0) {// child process
        if(execvp(args[0],args)<0){
          fprintf(stderr,"[%s] command doesnt exist\n",args[0]);
        }
        _exit(0);
        return;
    }
    /*
     *  コマンドの状態がバックグラウンドなら関数を抜ける
     */
    /******** Your Program ********/
    if ( pid>0 && command_status == 1) { // background
      printf("background command &\n");
      return; //
    }
    /*
     *  ここにくるのはコマンドの状態がフォアグラウンドの場合
     *
```

```
     *  親プロセスの場合に子プロセスの終了を待つ
     */
    /******** Your Program ********/
    else if(pid>0 && command_status==0){ //now in parent process
    //wait() は、状況情報を取得した子の プロセス ID (PID) になっている値を戻
        if(wait(NULL)==-1){ //error in
          fprintf(stderr,"ERROR\n");
          return;
      }
    }
}

int history_position(char arg[]){
 int current_position = (historyend-1+32)%32;
 char *string = arg;
 string++;
 /* Position for !string*/
 for(int i=0;i<32 && i<historyend;i++){
   if(strstr(history[current_position],string)!=NULL && (strstr(history[current_position
     return current_position;
   }
   else{
     current_position-=1;
     if(current_position<0) current_position+=32;
   }
 }
 /* Position for !-n*/
 int num;
 if(*string=='-'){
   string++;
   num = string_to_num(string);
   if (num<0|| num>32) return -1;
   current_position = (historyend-num);
   if(current_position<0) return -1;
   return current_position%32;
 }
 /* Position for !n*/
 else {
   num = string_to_num(string);
   if(num>32 || num>historyend) return -1;
   if(historyend<32 && num<=historyend) return num-1;
   return current_position = (historyend+num-1)%32;
 }
 return -1;
}
```

```
int string_to_num(char* string){
 int num =0;
 while(*string!='\0'){
     if(*string>='0' && *string<='9'){
       num*=10;
       num+=(*string-'0');
       string++;
     }
     else{
       return -1;
     }
 }
 return num;
}

char* connect_args(char* args[],int command_status){//
   char *combine_args = (char *)malloc(sizeof(char)*256);
   strcpy(combine_args,args[0]);
   for(int i=1;args[i]!=NULL;i++){
       strcat(combine_args," ");
       strcat(combine_args,args[i]);
   }
   if(command_status==1){
     strcat(combine_args," ");
       strcat(combine_args,"&");
   }
   strcat(combine_args," ");
   return combine_args;
}
void save_history(char *args[],int command_status){
   char *buf_history = connect_args(args,command_status);
   /* this line need confirmation : how to use free correctly */
   //free(history[historyend%32]);
   history[historyend%32] = buf_history;
   historyend++;
}

void cd_command(char *args[]){
 char cwd[256];
 if(args[2]!=NULL && args[1]!=NULL){
   printf("Too much data for command [cd]\n");
   return;
 }
 else{
   if(args[1]==NULL){
       if(getenv("HOME")==NULL || chdir(getenv("HOME"))==-1){
```

```
        printf("The location is not valid\n");
        return;
      }
   }
   else if(chdir(args[1])==-1){
     printf("The location is not valid\n");
     return;
   }
   if (getcwd(cwd, sizeof(cwd)) != NULL) {
       printf("Current Directory: %s\n", cwd);
   } else {
       perror("getcwd() error");
   }
 }
}

void pushd_command(char *args[]){
 char cwd[256];
 if(args[1]!=NULL){
   printf("Too much data for command [pushd]\n");
   return;
 }
 if(last_stack==MAXSTACK){
   printf("The stack is full\n");
 }
 else{
     char *stack_word1 = (char*)malloc(sizeof(char)*(256));
     strcpy(stack_word1, getcwd(NULL, 0));
     stack_dir[last_stack] = stack_word1;
     last_stack++;
   }
}

void dirs_command(char *args[]){
 if(args[1]!=NULL){
   printf("Too much data for command [dirs]\n");
 }
 printf("Current Stack(from newest): \n");
       for(int i=0;i<last_stack;i++){
         printf("\t%s\n",stack_dir[last_stack-1-i]);
       }
}

void popd_command(char *args[]){
 char cwd[256];
 if(args[1]!=NULL){
```

```
   printf("Too much data for command [popd]\n");
   return;
 }
 if(last_stack<1){
   printf("No directory in stack\n");
   return;
 }
 last_stack--;
 if(chdir(stack_dir[last_stack])==-1){
     printf("The uppermost directory in slack is not valid\n");
 }
 /* this line need confirmation : how to use free correctly */
 free(stack_dir[last_stack]);
 if (getcwd(cwd, sizeof(cwd)) != NULL) {
       printf("Current Directory: %s\n", cwd);
 }
 else {
       perror("getcwd() error\n");
 }
 return;
}
void history_command(char *args[]){

 int start=0;
 if(args[1]!=NULL){
     printf("Too much data for command [history] \n");
     return;
 }
 if(historyend>=32) start = historyend%32; //historyend=32(0-31) 33(1-0)
 printf("History:\n");
 for(int i=0;start+i<historyend && i<32;i++){
     printf("(%d) %s\n",i+1,history[(start+i)%32]);
 }
 return;
}

void alias_command(char *args[]){
   if(args[1]==NULL){
     for(int i=0;i<last_alias;i++)
     {
       printf("%s %s\n",alias_word[i][0],alias_word[i][1]);
     }
     return;
   }
   else if(args[2]==NULL){
     printf("Insufficient data for command [alias] \n");
```

```
    return;
  }
  else if(args[3]!=NULL){
    printf("Too much data for command [alias] \n");
    return;
  }
  else{
    if(last_alias==MAXALIAS){
      printf("The alias storage is full: delete some aliases first\n");
      return;
    }
    else{
    for(int i=0;i<last_alias;i++)
    {
      if(strcmp(alias_word[i][0],args[1])==0){
        strcpy(alias_word[i][1],args[2]);
        return;
      }
    }
    char *alias_word1 = (char*)malloc(sizeof(char)*(strlen(args[1])));
    char *alias_word2 = (char*)malloc(sizeof(char)*(strlen(args[2])));
    strcpy(alias_word1, args[1]);
    strcpy(alias_word2, args[2]);
    alias_word[last_alias][0]=alias_word1;
    alias_word[last_alias][1]=alias_word2;
    last_alias++;
    alias_word[last_alias][0]=NULL;
    alias_word[last_alias][1]=NULL;
    return;
    }
  }
  return;
}
void unalias_command(char *args[]){
 if(args[1]==NULL){
    printf("Insufficient data for command [unalias] \n");
    return;
  }
  else if(args[2]!=NULL){
    printf("Too much data for command [unalias] \n");
    return;
  }
  else{
    int i=0;
    for(1;i<last_alias && strcmp(args[1],alias_word[i][0])!=0;i++);
    if(i<last_alias){
```

```
       /* this line need confirmation : how to use free correctly */
       free(alias_word[i][0]);
       while(i<last_alias){
         alias_word[i][0]=alias_word[i+1][0];
         alias_word[i][1]=alias_word[i+1][1];
         i++;
       }
      last_alias--;
     }
    else{
      printf("Alias [%s] not found\n",args[1]);
    }
   }
}
void prompt_command(char *args[]){
 if(args[1]!=NULL && args[2]!=NULL){
     printf("Too much data for command [prompt]\n");
     return;
   }
 else if(args[1]==NULL){
     strcpy(word_command,"Command :");
     return;
 }
 strcpy(word_command,args[1]);
}

void pwd_command(char *args[]){
 char cwd[256];
 /*if(args[1]!=NULL){
   printf("Too much data for command [pwd]\n");
   return;
 }*/
 if (getcwd(cwd, sizeof(cwd)) != NULL) {
       printf("Current Directory: %s\n", cwd);
 }
}

char* wildcard_command(char *args){
 char* new_args= (char *)malloc(sizeof(char)*256);
 int position = -1; // 0 = * 1= *string 2 = string*
   if(strcmp(args,"*")==0){
           position = 0;
         }
   else if(strstr(args,"*")-args==0){
     position = 1;
     args++;
```

```
    }
    else if(strstr(args,"*")!=NULL && strstr(args,"*")-args!=0){
      position = 2;
      int i=0;
      for(i = 0;*(args+i)!='\0';i++);
      i--;
      args[i]='\0';
    }
    else{
      return " ";
    }
    struct stat    filestat;
    struct dirent *directory;
    DIR           *dp;
    dp = opendir(".");
    /*** [ディレクトリエントリの取得] ***/
    while((directory = readdir(dp))!=NULL){
      if(!strcmp(directory->d_name, ".") ||
        !strcmp(directory->d_name, ".."))
        continue;
    /*** [i-node 情報(ファイル情報)の取得] ***/
      if(stat(directory->d_name,&filestat)<0){
        perror("main");
        exit(1);
      }else{
        /* 表示 */
        if(strstr(directory->d_name,".")-directory->d_name==0) continue;
        if(position==0){
            strcat(new_args,directory->d_name);
            strcat(new_args," ");
        }else if(position==1 && strstr(directory->d_name,args)-directory->d_name+strlen(a
          strcat(new_args,directory->d_name);
          strcat(new_args," ");
        } else if(position==2 && strstr(directory->d_name,args)-directory->d_name==0){
          strcat(new_args,directory->d_name);
          strcat(new_args," ");
        }
      }
    }
    return new_args;
}
void rm_command(char *args[]){
 if(args[1]==NULL || (args[1][0]=='-' && args[1][1]=='r' && args[1][2]=='\0' && args[2]=
    printf("Insufficient data for command [rm] \n");
    return;
 }
```

```
 int start;
 if(args[1][0]=='-' && args[1][1]=='r' && args[1][2]=='\0') start = 2;
 else{
   start =1;
 }
 for(int i=start;args[i]!=NULL;i++){
   if(is_file(args[i])==true || (is_dir(args[i])==true && start==2)){
     if (remove(args[i]) == 0) {
         printf(" [%s] is deleted successfully\n",args[i]);
     } else {
         printf("[%s] is not deleted\n",args[i]);
     }
   }
   else if(is_dir(args[i])==true && start==1){
     printf("Directory [%s] cannot be deleted\n",args[i]);
   }
 }
 return;
}
```