시스템 프로그래밍 – 쉘 구현

[redacted]

1. 구현한 기능

-foreground and background execution (&)

-multiple commands separated by semicolons

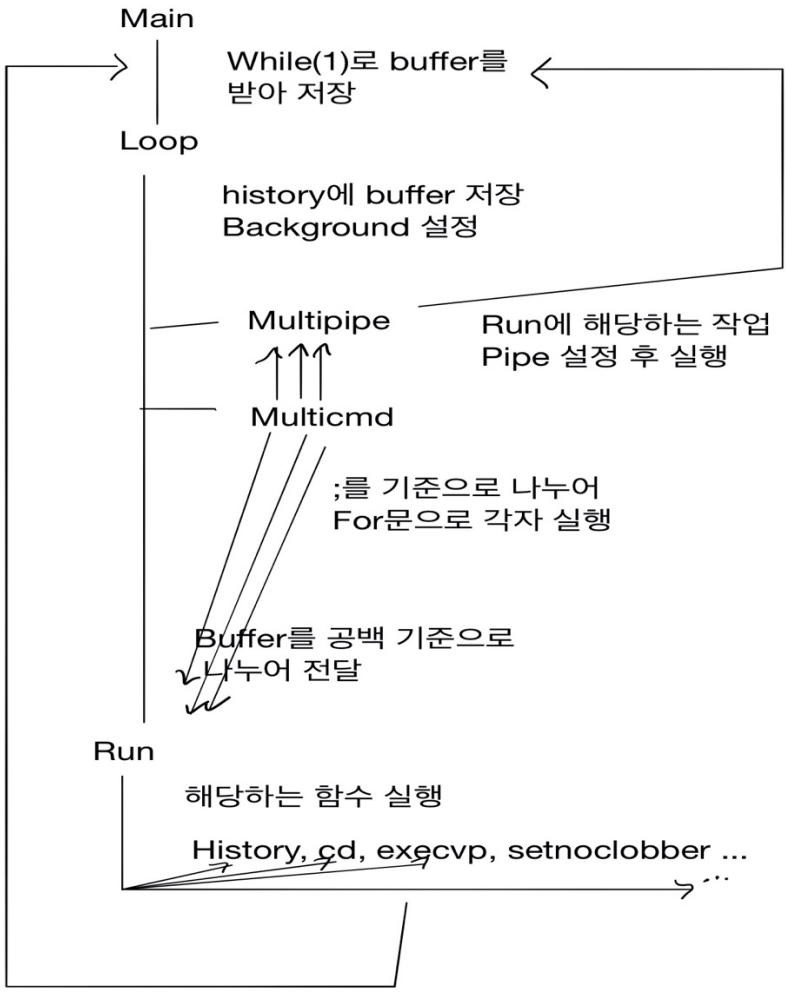
-history command

-shell redirection (>,>>,>|,<)

-multiple pipe

-cd command

2. 프로그램 기본 실행 흐름



3. 함수 설명

```
void fatal(const char *str, int errcode)
```

에러 처리 함수

str을 perror로 STRERR에 출력하고, exit(errcode).

```
int parsing(char buffer[], char* delimiter, char* arg[], int argn)
```

문자열을 나누어, 행렬에 저장하는 함수

문자열 buffer를 delimiter에 포함된 char를 기준으로 나누어준다.

나누어진 buffer를 arg에 char* 행렬로 넣어주고, 그 행렬의 크기 argn을 리턴한다.

```
int main(void)
```

공통으로 쓰일, 배열과 여타 자료형을 선언해준다.

무한루프를 돌며, 키보드로부터 버퍼를 받고, 공백이면 다시 continue

그렇지 않으면, loop함수에 넘겨준다.

```
void loop(char buffer[], char* history[], int* historyCnt, int* noClobber)
```

히스토리를 관리하고, 멀티파이프, 멀티커맨드, 그 외를 지정해주는 함수.

buffer를 history배열에 순서대로 쌓는다.

멀티파이프, 멀티커맨드만 따로 처리를 해주고,

나머지는 공백을 기준으로 parsing해, run 함수에 넘겨준다.

```
int isBackground(char* buffer)
```

background 여부를 확인하는 함수.

buffer가 &를 포함해, background execution의 실행여부를 결정해준다.

&를 포함했으면 1을, 그렇지 않으면 0을 리턴한다.

```
void cdCommand(char* arg[], int argn)
```

cd 명령어에 대한 수행을 해주는 함수.

arg[0]은 항상 문자열 “cd” 이다. Argn은 arg의 크기.

If, argn이 1이면 (Cd만 타이핑되면) , home으로 chdir,

Else if, argn이 2면, arg[1]에 해당하는 디렉토리가 있으면 chdir, 그렇지 않으면 에러 메세지 출력

Else, 에러 메세지 출력

```
void historyCmd(char* arg[], int argn, char* history[])
```

history 명령어에 대한 수행을 해주는 함수.

Arg[0]은 항상 문자열 "history" 이다. Argn은 arg의 크기.

Argn이 1이면, shell에 처리된 buffer들을 순서대로 저장해둔 history 배열을 순서대로 출력해준다.

```
void historyNum(char* arg[], int argn, char* history[], int* historyCnt, int*
noClobber)
```

!숫자 로 된, history에 저장된 buffer를 다시 실행시키는 함수

Arg[0]의 가장 첫번째 글자는 항상 '!'이다.

Argn이 1이면, arg[0]의 포인터를 하나 뒤로 옮겨준다. (! 제거)

Arg[0]이 숫자면, 저장된 history number에 그 숫자가 있는지 확인 (historyCnt와 비교)

있으면, 그 숫자에 해당하는 buffer를 loop함수의 파라미터로 전해줌.

없거나, 숫자가 아니면, 에러메세지 출력

```
int isRedirect(char* arg[], int *argn, int noClobber)
```

리다이렉션을 처리하는 함수.

arg에 >,>>,<,2>,2>,>| 가 있는지, 확인한고,

그에 따라, 파일을 열고, dup2를 사용해, 파일을 입/출력으로 재지정해준다.

1) >

arg배열에서 '>' 다음 문자에 해당하는 파일이 이미 존재하고, noClobber가 set 돼있으면, 에러메세지

그렇지 않으면, 해당 문자에 대한 파일을 열어주고 (없으면 생성) 표준 출력으로 지정해준다.

Arg배열에서 '>'를 Null로 설정해주어, 배열에서 '>' 와 파일이름에 해당하는 값을 제거해준다.

Argn도 2줄여준다. ('>'와 파일이름 두 개의 사이즈가 줄어들었기 때문)

2) >>

Arg배열에서 '>>' 다음 문자에 해당하는 파일을 append로 열어주고 (없으면 생성),

파일을 표준 출력으로 지정해준다.

Arg배열에서 '>>'를 Null로 설정해주어, 배열에서 '>>' 와 파일이름에 해당하는 값을 제거해준다.

Argn도 2줄여준다. (‘>’와 파일이름 두 개의 사이즈가 줄어들었기 때문)

3) <

Arg배열에서 ‘<’ 다음 문자에 해당하는 파일을 열어주고 (없으면 에러 메세지),

파일을 표준 입력으로 지정해준다.

Arg배열에서 ‘<’를 Null로 설정해주어, 배열에서 ‘<’ 와 파일이름에 해당하는 값을 제거해준다.

Argn도 2줄여준다. (‘<’와 파일이름 두 개의 사이즈가 줄어들었기 때문)

4) 2>

Arg배열에서 ‘2>’ 다음 문자에 해당하는 파일을 열어주고 (없으면 생성) 표준 에러로 지정해준다.,

Arg배열에서 ‘2>’를 Null로 설정해주어, 배열에서 ‘2>’ 와 파일이름에 해당하는 값을 제거해준다.

Argn도 2줄여준다. (‘>’와 파일이름 두 개의 사이즈가 줄어들었기 때문)

5) >|

noClobber flag와 상관없이, >의 파일 표준 출력 지정 동작을 수행한다.

```
void setNoClobber(char* arg[], int argn, int* noClobber)
```

noclobber를 설정하는 함수

arg[0]은 항상 set 이다.

Argn이 2면, arg[1]이 “+C” 일 때는 noClobber를 1로 설정

“-C”일 때는 0으로 설정.

Argn이 3이면, arg[2]가 noclobber이고, arg[1]이 “+o”면 noClobber를 1로 설정

arg[2]가 noclobber이고, arg[1]이 “-o”면 noClobber를 0으로 설정

그 외에는, 사용법 출력

```
int run(char* arg[], int background, int argn, char* history[],  int* historyCnt,
int* noClobber)
```

명령어에 따라, 구체적인 일을 수행하게 해주는 함수.

Arg[0]을 판독해, 어떤 동작을 수행해야 하는지 나누어준다.

Fork를 해, 자식 프로세스에서 해당 동작을 수행하는데, background가 1이면 부모 프로세스는 자식

프로세스를 기다리지 않고 (WNOHANG), background가 0이면 부모 프로세스는 자식 프로세스를 기다려준다.

```
void multiCmd(char buffer[], int background, char* history[], int* historyCnt, int* noClobber)
```

멀티커맨드를 처리하는 함수.

buffer[]에는 ;로 나뉘어 질, 명령어가 들어가 있다. Ex) cd ..; ls -la;

Parsing함수를 사용해, ;를 기준으로 나누어준다. newarg Ex) [cd ..,ls -la]

나뉘어진 명령어를 각각 다시 공백을 기준으로 나누어준다. arg Ex) [cd,..] , [ls,-la]

For문을 통해, arg를 각각의 실행흐름에 맞춰 실행시켜준다.

파이프가 쓰일 때만, 실행흐름이 다르기 때문에, multipipe로 옮겨주고

나머지는 run 함수를 실행시켜준다.

```
void multiPipe(char buffer[], int background, char* history[], int* historyCnt, int* noClobber)
```

파이프(멀티파이프 포함) 처리하는 함수.

buffer는 '|'를 포함하고 있는 문자열이다.

파이프의 입력 부분을 유지해줄 int in을 설정해주고, 파이프의 첫번째는 표준입력을 건드리지 않기 때문에 초깃값을 0으로 설정한다.

Buffer를 |를 기준으로 parsing 해준다. Newarg

그 후, 루프문에서 fork를 하고, 각각 공백을 기준으로 나누어준다. Arg

자식 프로세스에서는 출력 지정과 명령어 실행을 해주고,

부모 프로세스는 자식이 끝나길 기다린 후, 파이프의 입력 부분을 저장해주고 다음 루프로 넘어간다.