Design Document for Simple Shell

Table of Content

Special Data Structure

In this program, I create two data structures for the convenience.

- List
- ExecNode

List

The first one is the `List`, which is the abstract data type represent the common incrementable List to hold the generic pointers. Basically, this is a simple `vector<void *>` in C++ or `ArrayList<Object>` in Java.

```
typedef struct list {
    int size;
    void** content;
    int limit;
} * List;
```

The ADT has such fields:

Fields	Type	Usage

size	int	to maintain valid content size
content	void**	an array holding the actual pointers
limit	int	the actual physical size of the content array

This ADT has minimal useful functions:

Function	Signiture	Usage
lsnew	(int) -> List	Create a new list with specific size
lspush	(List, void*) -> void	Push a new pointer into the list
lsdup	(List) -> void**	Create a copy array of the content, ending with NULL
lsclear	(List) -> void	Clear the list content
lssize	(List) -> size	Get the size of list

ExecNode

The data structure representing a single Execution.

```
typedef struct exec_node {
    char** argv;
    int argc;
    char* infile;
    char* outfile;
    char forward;
    char background;
} ExecNode;
```

This ADT maintains the necessary information to execute a program:

Fields	Type	Usage
argv	char**	The arguments array of this execution, ending with NULL
argc	int	The number of arguments
infile	char*	The file will be redirected to STDIN. NULL for no redirection.
outfile	char*	The file will be redirected from STDOUT. NULL for no redirection.

forward	char	A flag representing if this execution will be forwarded. 1 for true, 0 for false.
background	int	UNUSED. A flag representing if this execution is in background.

There is only two function for this ADT:

Function	Signiture	Usage
ennew	() -> ExecNode*	Create a new empty execution node
enfree	(ExecNode*) -> void	Free all memory used by this execution

Utility Functions

The program contains some utility functions to help the implementation easier.

Function	Signiture	Usage
open_f	(char*m, int) -> int	A wrapper function for syscall `open` . Report error and exit if failed.
redirect	(int, int) -> void	A wrapper function for syscall `dup2` . Report error and exit if failed.

General Flow

The general flow of this program is simple.

```
Update the current work directory.
Create list `node` for all execution nodes.
Create list `args` to cache the valid arguments for current execution node.
In endless loop:
    Update the prompt.
    Get the arguments array.

    If it's built-in argument, handle it and wait next arguments array.
    Else loop for each argument string in this argument:
        if the string arugment is special character
            then handle the special effect
        else
            setup current node and push this string argument to args array.
        Invalidate current node if it exist
        Execute the current sequence of execution nodes
        Free all the nodes just used
```

The details of the flow could break into:

- Handle Built-in Command
- Handle Special Character
- Execution Execution Nodes

Handle Built-in Command

```
If the arg is "exit", then just exit the whole program
If the arg is "cd", handle the basic change folder, and update current working direc
```

Handle Special Character

Here I check the first character for the first argument, since the special argument must have only 1 length for our requirement.

```
If the char is |
    If the next argument exist and is non-special, set current execution node forwar

If the char is ;
    Invalidate the current execution node and flush recorded arguments array to exec

If the char is <
    if next argument exist and is valid, set current infile to next argument.

If the char is >
    If next argument exist and is valid, set current outfile to next argument.
```

Execution Execution Nodes

This is the tricky part in the program. Bascially, it loops for each execution node and execute node by its content.

```
For each node n in the list:
    If this node need forward, then create the pipe.

    Fork the program:
        If fail, report and exit.
        If it's in child:
            If the node is assigned infile, redirect infile to stdin.
            Else if the node assigned with pipe from previous process, redirect prev

            If the node is assigned outfile, redirect stdout to outfile.
            Else if the node need to pipe, assgin stdout to the pipe created.

            Replace the process content by execvp.
        If it's in parent:
            Wait for child.
            Close the write pipe in parent if we need to forward.
            Maintain the pipe's read port to the next iteration
```

The trick part is that we maintain an `int` variable out of the loop to hold the pipe read port from previous execution node. If previous node doesn't pipe, it's just STDIN.