Anubis UNIX Shell

Jack Kilrain (u6940136)

03 September 2023

Contents

1 Statment Of Originality

I declare that everything I have submitted in this assignment is entirely my own work, with the following exceptions:

- Inspiration for command structure and executor structure from *Introduction to Systems Programming: a Hand-on Approach (v2015-2-25)*: https://www.cs.purdue.edu/homes/grr/SystemsProgrammingBook/Book/Chapter5-WritingYourOwnShell.pdf
- Reference for behaviour of self-pipe: https://lkml.org/lkml/2006/7/10/300

2 Report

Note: All tests pass locally with `./test-anubis -t <1-30>` and `./test-anubis`, however the CI pipeline shows failures. I've included a `Dockerfile` with the exact environment used to develop and run the tests locally for reproducability. It can be built with `docker build -t comp3300 .` and then run with a mount for the assignment with `docker run -v "$(pwd):/comp3300-2023-assignment1" -it comp3300:latest`

2.1 Architecture

Three sections compose the Anubis shell in a coupled architecture. These are designed to allow for both flexibility, maintainability and efficiency. These sections are as follows:

- Structure
- Command Interpertation
- Execution
- Built-In Commands
- Resource Lifetimes

In terms of design philosphy the intention is to create a defined structure for commands that allows for flexible defintions, ease of parsing and adaptability for execution contexts.

2.2 Structure

Before delving into the core behaviour of the shell, first we look at the structure of the commands. These are broken up into distinct structs of a nested nature. Each nesting considers an atomic component of the commands.

At the high level We have the `CommandTable` which is essentially an array of one or more `CommandLine`. This creates a general form to store a series of commands in that generalises for both interactive and batch execution.

```
typedef Command** PipeList;

typedef struct CommandTable {
    size_t lineCount;
    CommandLine** lines;
} CommandTable;
```

Next, the `CommandLine` is composed of one or more piped commands, any enabled `IoModifiers` and whether it is to be run in the foreground or background. The `PipeList` entry contains instances of `Command`, note that a single `Command` entry in the `PipeList` is transparent as a single `Command`. Thereby requiring no additional metadata to determine the nature of single command execution vs piped execution. Note that this structure is packed via a common compiler attribute to ensure that it minimises the memory usage needed to represent the structure.

```
typedef struct __attribute__((__packed__)) CommandLine {
    size_t pipeCount;
    PipeList pipes;
    IoModifiers* ioModifiers;
    BackgroundOp bgOp;
} CommandLine;
```

The `IoModifiers` structure only contains a single entry for a truncated output redirection target, however, this is extendible to other types of IO such as appended output redirection, multi-descriptor targets, etc.

```c
typedef struct IoModifiers {
    char* out;
} IoModifiers;
```

Each instance of `Command` in the `PipeList` contains a target command name as well as an array of arguments (`Args`) to be passed to the resolved command.

```c
typedef char** Args;

typedef struct __attribute__((__packed__)) Command {
    size_t argCount;
    char* command;
    Args args;
} Command;
```

2.3 Interpreting

In order to construct the aforementioned command structure, we need to be able to interpret user input in a structural way. To do this, we employ the assistance of lexical analysis in tandem with parsing of lexums to generate and verify the structure.

2.3.1 Lexer

Lexical analysis for user input is exceedingly simple for the case of commands. This is mostly due to the lack of nested contexts or reference chaining that would otherwise be present in standard programming languages. However, we still have a series of variable length elements that are needed to be recongised. In out lexum alphabet, we consider the following symbols to be understood (loose regex for indication only):

- `AMPERSAND`: `&`
- `PIPE`: `|`
- `GREATER`: `>`
- `STRING`: `((?<=[&\|>\n\t\s])(.*)(?=[&\|>\n\t\s])) | ((?<=")(.*)(?=")) | ((?<=')(.*)(?='))` (Surrounding in special characters or quotations)
- `WHITESPACE`: `[\n\t\s]`
- `EOF/EOI`: `\0`

Note that for all of our purposes, anything that isn't a lexum with special meaning, is considering a string. At that, strings are bounded by these special lexums. This makes parsing significantly easier as we can simply rely on next-token intrinsicts with flags. Deduplication is handled dynamically for the `AMPERSAND` as a lexer construct, avoiding the need to reap multiple during parsing.

When parsing strings we should also consider the case of escape sequences, thus we can also handle these with flags and next-token instrinsics. Lastly, the case of quoted strings is to be considered, these are always in pairs and can be once again handled with flags, however it is worth noting that the lexer has no state of depth greater than 1. Thus we delegate validation of input to the parser once lexums have been created. Note that it does however handle string termination around the basis of quotations with flags which is a slight extension of its capabilities to ease usage. The lexer employs a greedy minimal match methodology, whereby it will attempt to consume as many tokens as necessary for the minimum match to occur.

2.3.2 Parser

In terms of a formal grammar definition for our simplified shell syntax, we have the following:

```
GOAL: CommandList;

Args: <STRING>*;

Command: <STRING> Args;

PipeList:
    | <PIPE> Command PipeList
    | Command;

IoModifier:
    | <GREATER> <STRING>;

IoModifiers: IoModifier?;

BackgroundOp: <AMPERSAND>?;

CommandLine: PipeList IoModifiers BackgroundOp;

CommandList: CommandLine*;
```

This grammar has been defined in a manner that formalises the parser as LR(1) (see Donald Knuth's paper https://doi.org/10.1016/S0019-9958(65)90426-2), which means Left-to-right input traversal, Right-most derivation (recursive descent into nested expansions) and single-token (or `k=1`) lookahead to determine validity and parse movement. The implementation of this parser follows a standard form of recursive descent from the top most token marked as `GOAL` downwards until an `EOF/EOI` is reached or an erronous input is detected. Error handling is done progressively for each expected sequence (for example `<GREATER> <STRING>` for IO modifiers).

2.4 Execution

Once the input has been parsed and validated, the structural representation is ready for execution. This is forwarded to the executor to handle.

2.4.1 Executor

In essence, the executor transforms the structural representation of the commands into sequences of syscalls based on the required actions. The process followed by the executor for each `CommandLine` in a `CommandTable` is defined via the follow seqeuence:

1. Save `STDIN/STDOUT` file descriptors
2. Configure input overrides for initial input IO modifiers (not implemented in the structure but supported in operation)
3. Iterate over the piped commands and perform the following
    1. Redirect input from the previous or inital IO context
    2. Configure the output overrides for output IO modifiers
    3. Redirect output to next IO context
    4. Check if command is a built-in
        1. If it is then invoke and continue to next piped command
        2. Otherwise continue
    5. Construct and initialise a self-pipe for error communication between parent and child
    6. Execute command as a child process
    7. Parent waits for self-pipe closure or error byte from child

        1. If an error byte is found, handle it
        2. Otherwise continue
    8. Close and clean up the self-pip
    9. Continue to next piped command, letting previous run asynchronously with blocking IO semantics to ensure ordering
4. Restore the saved `STDIN/STDOUT` file descriptors
5. If command is marked to be run in foreground
    1. Wait for child processes
    2. Otherwise continue and let child run asynchronously
6. Return from execution.

2.4.1.1 IO Redirection Initially we save the `STDIN/STDOUT` file descriptors via the `dup(int)` syscall to allow for indempotency between execution of commands. Before execution of each `Command` in the `PipeList` we initally configure IO to ensure that there are no special conditions needed for the first command in the `PipeList`. This sets up any input file rediction if it were present or otherwise defaults to a `dup(int)` of `STDIN`.

With the context of each `Command` execution, we bind the previous output IO (this could be `STDIN`, or a pipe from a previous process), to the `STDIN` for this command. Then a new pipe is created to allow for the `STDOUT` to be forwarded to the next command or to a file output.

2.4.1.2 Builtin Invocation Before attempting path resolution and execution of a potential executable denoted under the current `Command` instance, we lookup any built-in that may match first. This allows built-ins to perform the same opeations over IO as standard executables if need be (given the they are executed with IO already configured). In a later section (Builins), we will discuss how built-ins are handled in more detail.

2.4.1.3 Self Pipe Execution of a child process after a `fork()` can be tricky to handle when it comes to errors. Particulr ensuring that errors bubble up to the parent in the failure case but in a successful case execution is not impeded. A technique called self-piping has been used since its inception in ~2006 (circa this mailing list entry: https://lkml.org/lkml/2006/7/10/300) which allows exactly this.

It works by creating a pipe, where the write port is held by the child and the read port is held by the parent. During its creation, `fcntl(...)` is used to configure the pipe with the `O_CLOEXEC` flag ammended to the current flags of the pipe.

```
#include <fctnl.h>
#define WRITE_PORT 1
#define READ_PORT 0
fcntl(
    selfPipe[WRITE_PORT], F_SETFD,
    fcntl(selfPipe[WRITE_PORT], F_GETFD) | FD_CLOEXEC
)
```

Using this flag allows the pipe to be closed on two conditions:

1. If an `exec(...)` call is performed successfully
2. A byte is written into the pipe and closed

The first case allows for the parent to proceed without waiting for the execution to finish and the second allows it to handle any error byte written to the pipe. To do this the parent polls on the pipe until either an error byte is found or an `EOI` (-1) is ecountered.

```
int count = -1;
close(selfPipe[WRITE_PORT]);
```

```
while ((count = read(selfPipe[READ_PORT], error, sizeof(errno))) == -1) {
    if (errno != EAGAIN && errno != EINTR) {
        break;
    }
}
return count;
```

As for why it is called as self pipe, that is because the child is essentially a copy of the parent (in terms of process semantics) and thus the pipe is essentially between the same process giving rise to the name self-pipe.

2.4.1.4 Background Execution At the end of the exeuction of a `CommandLine`, we conditionally invoke the `wait(NULL)` syscall to block on all child processes. This is only used when the `bgOp` flag is not set, implying foreground execution. Otherwise the parent process proceeds to the next `CommandLine`. Note that is possible since, between `CommandLine` instances there is no structural dependency, though execution dependency may exist for the programs being executed, that is in the realm of the programmer to handle, not the shell.

2.4.2 Path Resolution

2.4.2.1 Path Variable For the purposes of this shell, the internal path variable uses the same structure and semantics as the standard `PATH` environment variable on UNIX systems. It is essentially a string of `:` (colon) delimited paths that can be searched when determining the presence of an executable.

Adding a new path is a simple operation consisting of appending `:<file/directory>` to the end of the path variable. There are no restrictions on whether these are absolute or relative paths, similarly no distinction is made to de-duplicate entries in this list. It is left up to the programmer to ensure a consistent state as earlier entries will match before later entries regardless of duplication anyway.

Clearing the path is simply an operation to reduce the path variable to an empty string.

2.4.2.2 Target Resolution Given an input command as a target, we first determine if it has any unescaped forward slashes in the target. If so, it is treated as a path to a specific file system location and will not be resolved, thus being returned as is. If this is not the case, resolution continues.

Firstly, the path variable is iterated over by tokenising on the `:` (colon) delimiter. Each iteration invokes the `stat(...)` syscall on the path entry and determines if it exists, skipping if not. Next if the path is a directory we invoke directory resolution to any files within. The directory access is determined via the `access(...)` syscall against the `F_OK | X_OK` (file exists & executable) flags. If this fails, we continue.

If the path refers to a file the same access check if perform, returning that path directly if successfull. Otherwise we continue to the next delimited iteration.

2.4.2.2.1 Directory Introspection In the case that we have an accessible directory to inspect, we scan though the directory via the `scandir(...)` syscall matching only entries that have `DT_REG` (regular file), `DT_LNK` (symbolic links), `DT_CHR` (character device like a tty) or `DT_UNKNOWN` (unknown type) set. In the first case, if the file name matches then we can return the directory entry as a match. In the second case, if the resolved symlink points to a regular file or character device then we have a match and in the last case, if invoking `stat(...)` on the unknown entry yields a regular file or character device type under `mode`, then we have a match. All other cases yield no matches.

```
#define NON_EXEC_TARGET(entry) (\
    (entry) != DT_REG\
    && (entry) != DT_LNK\
```

```
    && (entry) != DT_CHR\
    && (entry) != DT_UNKNOWN\
)

LINKAGE_PRIVATE int directory_filter(const struct dirent* dir) {
    if (!dir || NON_EXEC_TARGET(dir->d_type) || strcmp(dir->d_name, filename)) {
        return 0;
    }
    if (dir->d_type == DT_LNK || dir->d_type == DT_UNKNOWN) {
        struct stat sstat;
        return stat(filename, &sstat)
            && (S_ISREG(sstat.st_mode) || S_ISCHR(sstat.st_mode));
    }
    return 1;
}
```

2.5 Builtins

Built-in commands are designed to be easy to, write, invoke and have the same semantics as regular commands due when executed as processes. Esentially this means the following:

1. Registration should be simple
2. A consistent minimal SPI
3. Ivocation is low overhead
4. Same IO and redirection semantics as regular child processes.

In order to meet the second requirement, we have standard type that a built-in must implement, this allows for consistency and easy of use:

```
typedef int (*BuiltinCmd)(char** args, size_t argCount);
```

This ensures only the required information, that is the arguments, are needed. Everything else is resolvable the same way a child process would. For instance in the case of IO, using the `STDIN_FILENO/STDOUT_FILENO` file descriptors is valid as they have been preconfigured as if it were a child process. In addition they are cleaned up and forwarded in the same manner as well automatically without needing any special handling in the bultin.

Ensuring lower invocation overhead and registration simplicity is done by employing a constraint on how builtins are registered, more specifically on the ordering in the `BuiltIn built_in_command[]` extern as alphabetical over the names. This allows for a binary search to be performed over the names, significantly reducing the time to resolve a command.

```
LINKAGE_PRIVATE BuiltIn* builtin_binary_search(char* command) {
    int lower = 0;
    int mid;
    int upper = built_in_commands_size - 1;
    int ret;
    while (lower <= upper) {
        mid = (lower + upper) / 2;
        BuiltIn* builtIn = &built_in_commands[mid];
        ret = strcmp(builtIn->name, command);
        if (ret == 0) {
            return builtIn;
        } else if (ret > 0) {
            upper = mid - 1;
```

```
        } else {
            lower = mid + 1;
        }
    }
    return NULL;
}
```

Since `strcmp(...)` is implemented in a fail-fast manner, we can be sure it will check length first, then progressive character match. This fits perfectly into the binary search algorithm and improves efficiency from $\mathcal{O}(n)$ to $\mathcal{O}(\log(n))$.

2.5.1 `cd`

The implementation of `cd` is essentially a checked wrapper around the `chdir(char*)` syscall. We check that there is only one path argument first, before invoking the sycall, then the result is either `errno` or `0` depending on failure or success.

2.5.2 `exit`

Similar to `cd`, we wrap the `exit(0)` syscall with a check to ensure that there are no parameters povided.

2.5.3 `path`

Lastly, the `path` builtin is a wrapper around the path operations described above (clear & add). We check the argument count first and invoke a clear if no args are present, otherwise adding args as path entries. Depending on the result, either `errno` or `0` is returned depending on failure or success.

2.6 Resource Lifetimes

All heap-allocated resources inside the shell have their lifetimes properly managed to ensure that there are no memory leaks. In order to perform cleanup properly without requiring hard-to-maintain cleanup hook invocations from `exit(0)` points, we use the `atexit(func)` clib to create a binding to a function that performs cleanup on exit. This function essentially acts as a resource deallocator that works irrespective of program state, ensuring it cannot fail.

```
static bool initialised = false;
static Parser parser;
static CommandTable* table = NULL;
static Lexer* lexer = NULL;
static char* line = NULL;

static void exit_handler(void) {
    command_table_free(table);
    lexer_free(lexer);
    path_free();
    checked_free(line);
    while (wait(NULL) >= 0);
}

// ... snip ...

int (int argc, char** argv) {
    int ret;
    transparent_return(atexit(exit_handler));
```

```
    // ... snip ...
}
```