What is a shell?

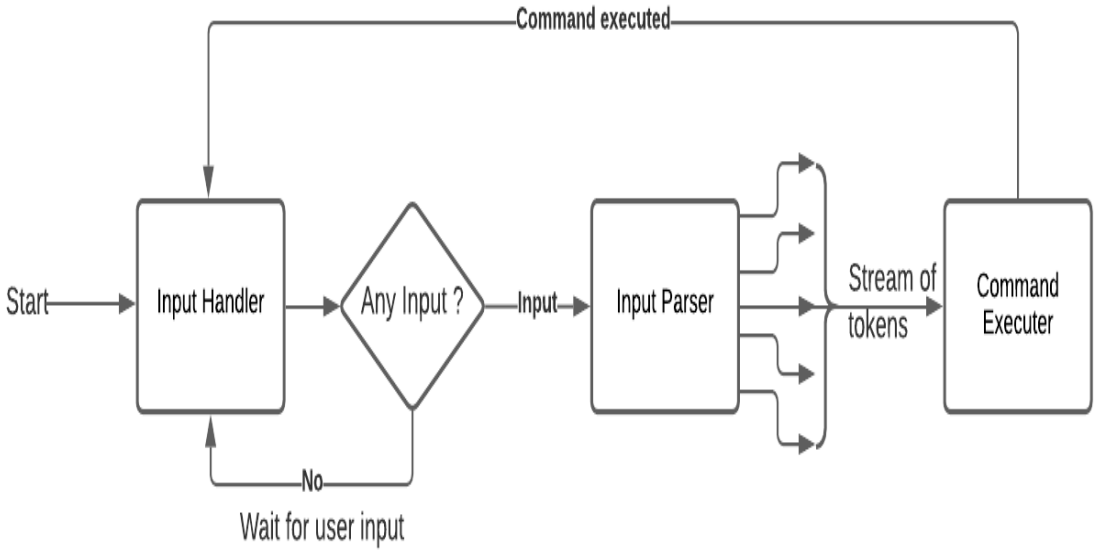
A Shell is a program that takes the command inputs written from the user's keyboard and passes them to the machine to execute them through the kernel. It also verifies if the command inputs from the user are correct.

Shell lifetime

- Initialize
- Interpret: Next, the shell reads commands from stdin (which could be interactive, or a file) and executes them.
- Terminate: After its commands are executed, the shell executes any shutdown commands, frees up any memory, and terminates.

Shell Architecture

- Input Handler: Read the command from standard input, it should always be ready to take any command and accept user input smoothly.
- Input Parser: acts as command interpreter that tell the command executer what to do, it Separate the command string into a program and arguments.
- Command Executer: Run the parsed command.



Input Handler

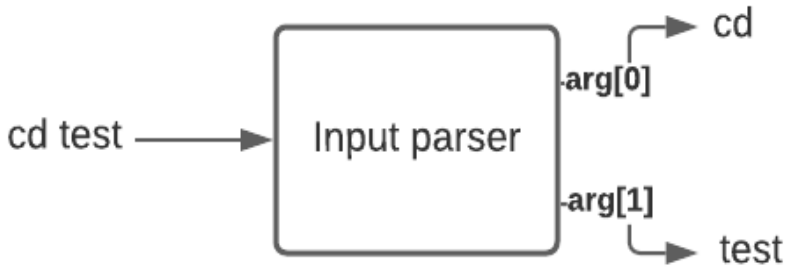
The problem when reading a line from stdin is that we don't know how much text a user will enter into their shell. We can't simply allocate a block and hope they don't exceed it. instead, we used readline library, readline read a line from the terminal and return it, it accepts one argument as a prompt printed when seeking user input, also readline offers editing capabilities while the user is entering the line one of them is a csh-like history expansion we used in our code.

A typical input handler can be written as following:

```
1.  char * getInput() {
2.
3.    char * line; //string used to store user input
4.    line = readline(prompt); //read the user input
5.    if (strlen(line) != 0) {
6.      add_history(line); //store non empty commands
7.    }
8.
9.    return line;
10. }
```

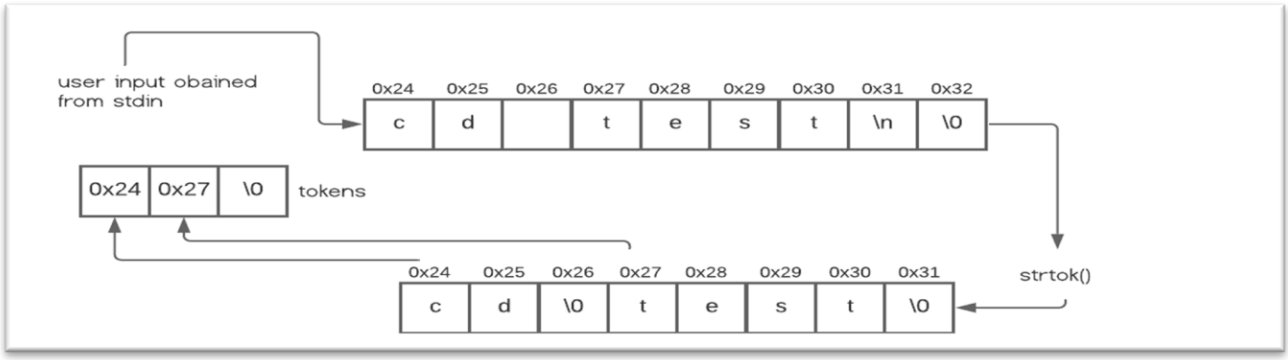
Input Parser

After we get the line of input, we need to parse that line into a list of arguments, we will simply use whitespace to separate arguments from each other so, all we need to do is "tokenize" the string using whitespace as delimiters, for this purpose we used strtok, a function which breaks string into a series of tokens using a delimiter we specify.



How strtok() work

It breaks string by sets a null character in each delimiter indicated, then we obtain tokens by creating an array of pointers, each pointer points to the address of the first character right to NULL.



We do not get the token by using only one Strtok() call as it returns only a pointer to the first token found in the string , so we use consecutive calls:
On a first call, the function expects as argument a C string whose first character is used as the starting location to scan for tokens.
In subsequent calls, the function expects a null pointer and uses the position right after the end of the last token as the new starting location for scanning.

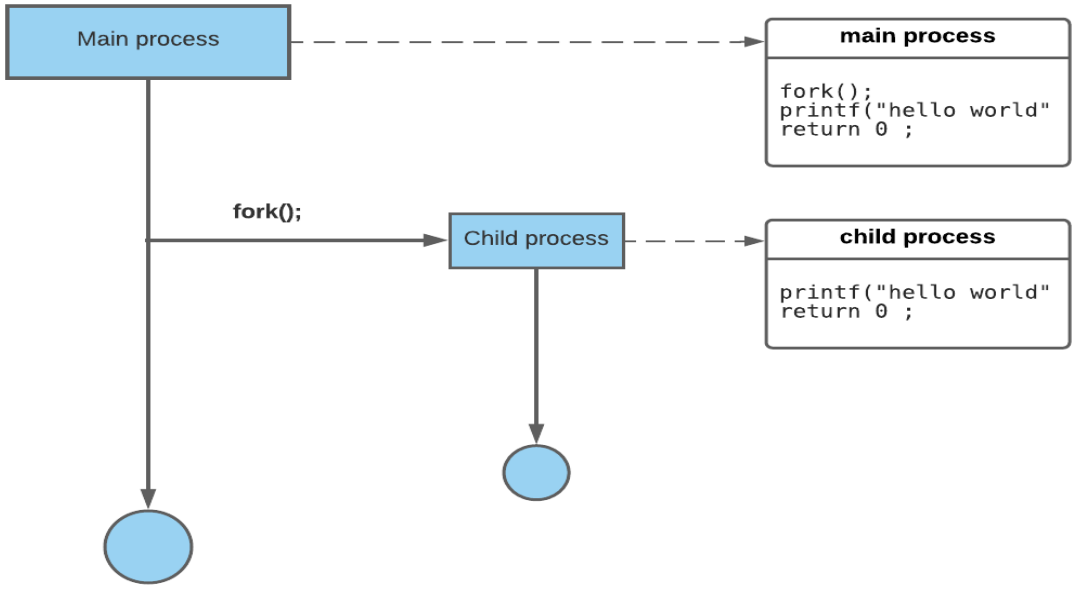
Our parser implementation:

```
1.  #define TOK_BUFSIZE 10
2.  #define TOK_DELIM " \t\r\n\a"
3.
4.  char ** parser(char * line) {
5.
6.    size_t bufsize = TOK_BUFSIZE; //max tokens one command may have
7.    int position = 0;
8.    char ** tokens = malloc(bufsize * sizeof(char * ));
9.    char * first_token;
10.
11.   first_token = strtok(line, TOK_DELIM);  //first call
12.   while (token != NULL) {
13.     tokens[position] = token;
14.     position++;
15.   //check wheter the command tokens exceed the memory we allocated
16.     if (position >= bufsize) {
17.       bufsize += TOK_BUFSIZE; //increment size by 10
18.       tokens = realloc(tokens, bufsize * sizeof(char * )); //reallocation
19.     }
20.
21.     token = strtok(NULL, TOK_DELIM);//subsequent calls
22.   }
23.   //assign the last position token with null indicating the end of the command,
24.   //that tell the command executer to start execute the next command
25.   tokens[position] = NULL;
26.   return tokens;
27. }
```

Command executer

Fork()

The only way for processes to get started is by calling the fork() system call. When this function is called, the operating system makes a duplicate of the process and starts them both running. The original process is called the “parent”, and the new one is called the “child”. fork() returns 0 to the child process, and it returns to the parent the process ID number (PID) of its child. In essence, this means that the only way for new processes is to start is by an existing one duplicating itself.



Exec()

Typically, when we want to run a new process, we don’t just want another copy of the same program – you want to run a different program. That’s what the exec() system call is all about. It replaces the current running program with an entirely new one. This means that when you call exec, the operating system stops your process, loads up the new program, and starts that one in its place. A process never returns from an exec() call unless there’s an error.

Putting together fork() & exec()

With these two system calls, we have the building blocks for how most programs are run on Unix. First, an existing process forks itself into two separate ones. Then, the child uses exec() to replace itself with a new program. The parent process can continue doing other things but, in our case, we make it wait until the child executed using the system call wait().

Typical implementation for process handler

```c
void processExecuter(char **args)
{
  pid_t pid, wpid;
  int status;

  pid = fork();
  if (pid == 0) {
    // Child process
    execvp(args[0], args);
    }
  else {
    // Parent process
    //wait until child process executed
    do {
      wpid = waitpid(pid, &status, WUNTRACED);
    } while (!WIFEXITED(status));
  }
}
```

Pipes

Pipe is a connection between two processes, such that the standard output from one process becomes the standard input of the other process.

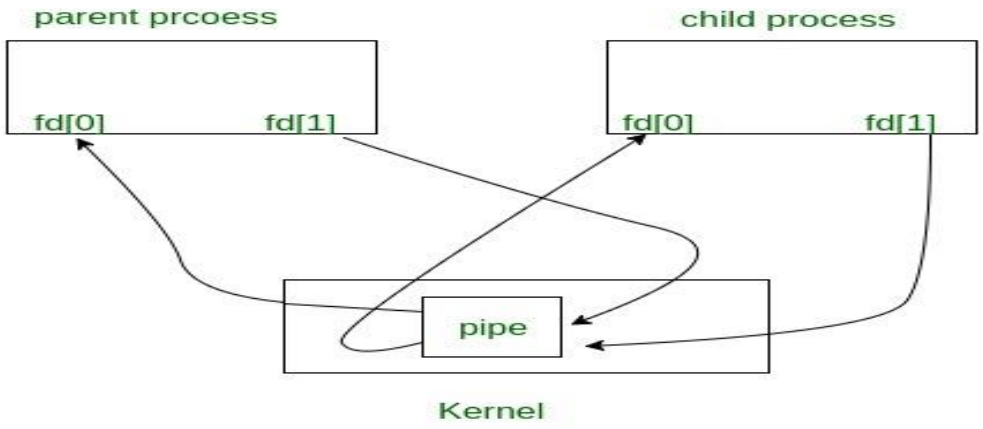
A typical syntax of UNIX pipe : command_1 | command_2 | command_3 | .... | command_N

Pipe Handling

-First step when handling pipes is to recognize pipes and separate its parts “ls | wc”→ ls , wc this is done using special parser for pipes.

-After parsing each part, call both parts in two separate new children, using execvp.

When we use fork in any process, file descriptors remain open across child process and also parent process. If we call fork after creating a pipe, then the parent and child can communicate via the pipe.



A typical pipe handling procedure

1. Declare an integer array of size 2 for storing file descriptors. File descriptor 0 is for reading and 1 is for writing.
2. Open a pipe using the pipe() function.
3. Create two children.
   The output of Child1 must be taken into the pipe so that Child2 can accept it as an input.

   In Child1:
   1. Copy file descriptor 1 to stdout.
   2. Close file descriptor 0.
   3. Execute the first command using execvp()

   In Child2:
   1. Copy file descriptor 0 to stdin.
   2. Close file descriptor 1.
   3. Execute the second command using execvp().
4. Wait for the two children to finish in the parent.

Built-ins

The builtins are functions included in the shell itself and don't need any other program to run them.

If we want to change directory, we need to use the function chdir(). The thing is that the current directory is a property of a process. So, if we wrote a program called cd that changed directory, it would just change its own current directory, and then terminate. Its parent process's current directory would be unchanged. Instead, the shell process itself needs to execute chdir(), so that its own current directory is updated. Then, when it launches child processes, they will inherit that directory too. Similarly, if there was a program named exit, it wouldn't be able to exit the shell that called it, that's why we need to add some commands to the shell itself.

Signals

A signal is an event which is generated to notify a process or thread that some important situation has arrived. When a process or thread has received a signal, the process or thread will stop what its doing and take some action, Each signal has a default action, some action may terminate the process others may ignore the signal

Interrupt handling: When the user types the INTR character (normally Ctrl + C) the SIGINT signal is sent this Interrupt the process and cause it to be terminated, instead of using the default interrupt handler we will specify a custom handler to print "interrupted" informing the user that the current process stopped.

SPAG_SHELL lifetime

1. Initialize the shell: clear the screen then print welcome sentences.
2. Declare signal handler for interrupt signal.
3. Print prompt and start accepting user input.
4. After a command is entered, the following things are done:
    1. Command is entered and if length is non-null, keep it in history.
    2. Parse input pipes then return the pipe elements, if the command didn't include a pipe char the parser will return the command itself.
    3. Parsing the commands returned from pip parser.
    4. Checking if built-in commands are asked for, if so, send command to built-ins handler
    5. Else executing system commands and libraries by forking a child and calling execvp.
    6. asking for next input.

We can type a pseudo code for the previous procedure as

```
1.  initiateShell(); //initating the shell
2.  signal(interrupt signal, interrupt signal Handler);//declare interrupt signal handler
3.    //looping
4.    while (1) {
5.
6.      input=getInput()
7.      pipe_elements = pipeParser(input);
8.      //parsing all pipe commands
9.      for (int currentCommand = 0; currentCommand < totalNoOfCommands; currentCommand++)
    {
10.       commands[currentCommand] = commandParser(pipes[currentCommand]);
11.     }
12.     //if the command is a builtin command pass it to builtins handler
13.     if (isBuiltin()) builtinExecuter(commands);
14.     else commandHandler(commands);
15.   }
```

The source code includes heavily commented Implementation and explanation for `initiateShell()`, `interruptHandler()`, `getInput()`, `pipeParser()`, `commandParser()`, `builtinExecuter()` and `commandHandler()`