**EECE7376: Operating Systems: Interface and Implementation**

**Spring 2023 ‐ Project**

Yinan Hu, NUid: 002108083

Xuebao Zhao, NUid: 002108354

**Part1 - Shell**

1. **Approach**

This problem is to implement a shell that reads and executes multiple commands with sub-commands, input/output redirection, and background execution support. And the string manipulation routines is implemented based on Homework 2 - Problem 3.

* 1. **Parse Command**

This part is included in command.h and this header file contains functions to parse commands with sub-commands, whether need input/output redirection, and background execution support. It consists of several functions that aid in parsing the commands and populating the Command and SubCommand structures for preparation of execution in shell.c file. The main functions in command.h are:

1. ReadArgs: Extracts arguments from the input string and stores them in argv.
2. ReadCommand: Populates the Command data structure from the input string by tokenizing the input string based on the '|' character, storing the sub-commands in the Command structure, and extracting arguments for each sub-command.
3. RemoveElements: Removes certain elements, such as input/output redirection and background symbols, from argv.
4. RemoveCommandElements: Removes the mentioned elements for all sub-commands in the Command structure.
5. ReadRedirectsAndBackground: Populates the stdin\_redirect, stdout\_redirect, and background fields of the Command structure.
6. PrintArgs: prints all arguments for the given sub-command
7. PrintCommand: prints all arguments for each sub-command of the command calling PrintArgs.
   1. **Execute sub-command**

The “execute\_sub\_command” function accepts the sub-command array from one command, input/output file descriptors, and a background flag as its arguments. It creates child process to handle the execution of the sub-command and it follows these steps:

1. Fork the current process, creating a child process to execute the sub-command. And the parent process wait or print pid.
2. In the child process, redirect input and output if necessary, by duplicating the file descriptors using dup2(). The process can get the input from the last command or standard input, output the result to the next command or standard output by indicating corresponding file descriptors.
3. Execute the sub-command using execvp().
4. If the sub-command is not found, print a proper error message and exit, immediately followed by the next prompt.
5. In the parent process,we wait for the child process to complete if the command is running in the foreground. Because when a command is launched in the foreground, the shell should wait until the last of its subcommands finishes before it prints the prompt again. When running in background, the parent process prints the pid if it is the last command and continue. Because when a command is launched in the background using suffix &, the shell should print the pid of the process corresponding to the last sub-command. It should then immediately display the prompt and accept new commands, even if any  
   of the child processes are still running.
   1. **Execute the whole command line**

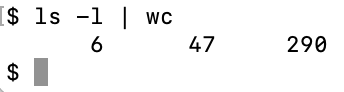
The “execute\_command function accepts a command structure as its argument and is responsible for handling input/output redirection and executing sub-commands. It uses a pipe to implement direct connection between commands and permits data to be transferred between them continuously. The function follows these steps:

1. Initialize file descriptors for input and output. The input redirection operator < redirects the standard input of the first sub-command while the output redirection operator > redirects the standard output of the last sub-command. The open() function is used to handle their redirections. Check if there is error related to files. If the input file is not found the output file cannot be created, a proper message should be displayed.
2. Iterate each sub commands by a loop, setting up pipes between them and calling execute\_sub\_command() for each sub-command. We create a pipe for the communication of two neighbor processes for each sub-command over an array where fd[0] is used for reading and fd[1] is used for writing. The pipe acts as a container which takes the output of the sub-command in the current process and gives it to the sub-commend in the next child process.
3. We use execute\_sub\_command() to create a child process for the execution and redirection of every sub-command. Before calling execute\_sub\_command(), we set the input/output file descriptors according to the command’s fileld: “stdout\_redirect” and “stdin\_redirect” and pass them to execute\_sub\_command().
4. In the process for output, the output of the command will go to file descriptor of input, fd[0] for the next sub-command's input. And similarly, the file descriptor of output is set as fd[1].
5. Close the file descriptors and repare input for the next sub-command.
   1. **Main function**

The main function serves as the entry point for the program and is responsible for reading commands from the user, parsing them, and executing them. It operates in an infinite loop and performs the following steps:

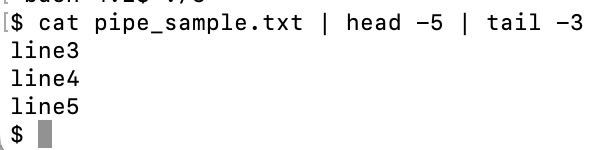
1. Print a $ prompt symbol when it is ready to receive a new command from the user. Then read a command from the user.
2. Remove the newline character at the end of the command, if any.
3. Parse the command using the ReadCommand(), ReadRedirectsAndBackground() and RemoveCommandElements() functions.
4. Execute the command.
5. Continue the loop and receive next command.
6. **Result**
   1. **Command with piping launched in the foreground**

The shell takes the output of “ls -l” and gives it to the sub-command “wc” as an input. When a command is launched in the foreground, the shell should wait until the last of its subcommands finishes before it prints the prompt again.



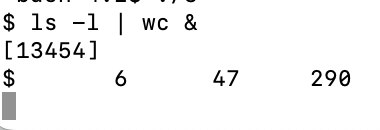
* 1. **Command with piping for twice in the foreground**

This command select first 5 lines through (head -5) command and that will be input to (tail -3) command which will finally print last 3 lines from that 5 lines.



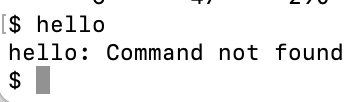
* 1. **Command with piping launched in the background**

When a command is launched in the background. The shell should print the pid of the process corresponding to the last sub-command. It should then immediately display the prompt and accept new commands, even if any of the child processes are still running. After printing the prompt, the child process output its result.



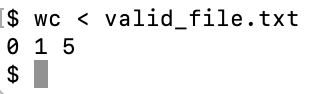
* 1. **Command not found**

When a sub-command is not found, a proper error message should be displayed, immediately followed by the next prompt.



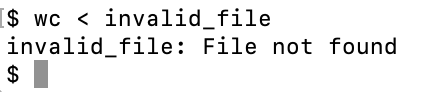
* 1. **Input redirection - successful**

The input redirection operator < should redirect the standard input of the first sub-command of a command.



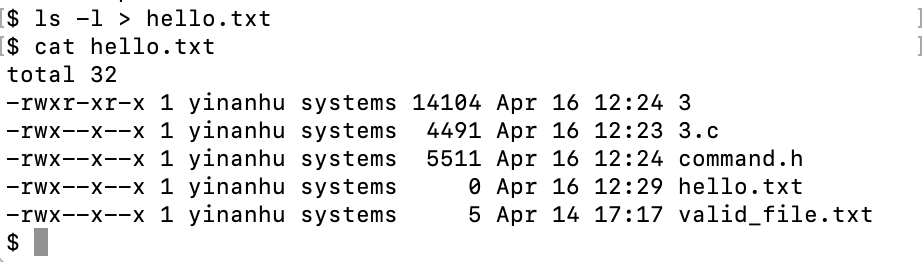
* 1. **Input redirection – file not found**

If the input file is not found, a proper message should be displayed.



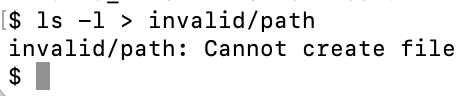
* 1. **Output redirection – successful**

The output redirection operator > should redirect the standard output of the last sub-command  
of a command.



* 1. **Output redirection – invalid path**

If the output file cannot be created, a proper message should be displayed.



**Part2 – xv6 Extension**

1. **Approach**
   1. **Modify the scheduler**

* Open the “proc.h” file and add a “priority” and a time slice counter(“timeslice”) attributes to the struct proc.
* Open the “proc.c” file and initialize the time slice counter attribute of a new process in the “allocproc()” function. Here we set default priority to 1 and the initial time slice counter to 0.
* Open the “proc.c” file, define the “QUANTUM” constant at the beginning of the “proc.c” file and modify the “scheduler()” function to implement the new time slice-based priority adjustment feature. In the “scheduler()” function, the first for loop iterates over priority levels (0, 1, and 2). The second for loop iterates over all processes in the process table. If a process is not in the RUNNABLE state or its priority doesn't match the current priority level, the scheduler skips it and proceeds to the next process. When a RUNNABLE process with the correct priority is found, the scheduler performs a context switch to the process by setting up the process's virtual memory and updating the state to RUNNING. When the process yields the CPU or finishes executing, the scheduler regains control, and the process state is checked. If the process is still RUNNABLE, the scheduler increments the process's timeslice counter. If the process has used up its entire time slice (reached the QUANTUM limit), the scheduler resets the timeslice counter and decreases the process's priority (moves it down one queue) if it's not already at the lowest priority level (2). If the process is not RUNNABLE, the scheduler resets the timeslice counter.
  1. **Add a system call**
* Modify “syscall.h” to add a unique numeric identifier for the new system call. This identifier will be used to map the system call to its corresponding handler function in the kernel. The identifier should be unique and not conflict with any of the existing system call numbers. Here we add value 22 for “SYS\_renice”.
* A new line of code is added in the “usys.S” file to correspond to the new system call. The purpose of this file is to define the user-side assembly code that invokes system calls in the kernel. To add the new system call, we need to follow the pattern of the existing system calls in the file. Specifically, we need to use a macro that automatically converts a user-friendly function invocation into a set of instructions that set up the function arguments and execute a trap (int $64) instruction with the appropriate system call number in %eax
* A new entry is added to the syscalls array in the “syscall.c” file and declare the “sys\_renice()” function using the extern keyword. The “syscalls” array is used to map system call numbers to their corresponding handler functions in the kernel. Also, “sys\_renice()” function prototype is added and the syscall table is updated.
* The “sys\_ renice ()” function is defined in “sysproc.c” file. This function reads its argument (the priority we want to modify) from the user stack using the argptr() function, which is provided by xv6 to fetch a pointer argument from the user stack. It will check the validity of the parameter and then pass the value to be changed to the process table.
* Added the user-space wrapper function for the “renice()” system call in the “user.h” file. The user-space wrapper function is a C function that provides a convenient interface for user-space programs to invoke the system call.
  1. **Add a user program to xv6 to test**
* N is defined as the number of child processes to create. The “long\_running\_task()” function simulates a long-running task by running two loops.
* In the “main()” function, an array “initial\_priority” is defined to store the initial priority levels of the child processes. The first for loop iterates N times, creating a child process with “fork()” in each iteration. If the pid is 0, it means the child process is executing. If the pid is less than 0, it means “fork()” failed, so an error message is printed, and the program exits.
* The child process does the following. Calls “renice()” to set its priority to the corresponding value in the “initial\_priority” array. Calls “long\_running\_task()” to simulate a CPU-intensive task. Prints a message to indicate the completion of the process with its process ID and priority level. Exits the child process.
* In the parent process, another for loop iterates N times and calls wait() in each iteration. This loop ensures that the parent process waits for all child processes to complete before exiting. After all child processes have completed, the parent process exits.

1. **Result**

Text

Description automatically generated

The output shows the order in which processes complete their execution. The MLFQ scheduler prioritizes processes based on their priority levels. The processes with the highest priority level (0) will be executed first, followed by processes with priority level 1, and then processes with priority level 2.

The MLFQ scheduler ensures that higher priority processes are executed before lower priority processes. This behavior is different from the default Round-Robin scheduler, which would execute processes in a circular order, regardless of their priorities or workload characteristics. In contrast, the MLFQ scheduler better handles processes with different priorities and characteristics, leading to improved responsiveness and overall system performance.