Ayça Elif Aktaş - 27802

In handling command-line arguments for execvp, special consideration was given to arguments that may start with a special character like a dash character (-). To prevent misinterpretation of such arguments as options, the convention of inserting" --" after the first option was employed. This signals the end of options and ensures that any subsequent argument starting with a dash character is treated as a positional argument rather than an option. This approach safeguards against unintended behavior caused by the interpretation of special characters as options.

For the implementation of the Shell, each command in the command.txt file was implemented as a separate child process from the Shell process. Input and output redirection of the command depends on the presence of redirection operators (> or <). According to these operators, the appropriate file descriptors are closed and reopened to associate the standard input or output with the specified file. In the child process, if output redirection (> file.txt) is specified, the standard output is closed, and the file is opened for writing. The dup2 system call is used to associate the file descriptor of the opened file with the standard output. If input redirection (< file.txt) is specified, the standard input is closed, and the file is opened for reading. The dup2 system call is used to associate the file descriptor of the opened file with the standard input.

To manage and track all child processes executing background commands, the shell process employs a bookkeeping mechanism. This mechanism involves the use of an array of structures named **backgroundJobs** to keep a record of each background job's process ID (pid) and its execution status (isRunning). Additionally, a counter **numJobs** is maintained to keep track of the total number of background jobs. The **struct BackgroundJob** is defined with two fields:

- pid: Process ID of the background job
- isRunning: A flag indicating whether the background job is currently running (1 if running, 0 if terminated).

The **backgroundJobs array** is initialized to store information about background jobs. The array has a maximum capacity of **MAX_JOBS**. When a background command is encountered, the shell checks whether the backgroundJobs array has reached its maximum capacity. If so, it dynamically reallocates memory to expand the array. The child process spawned to execute the background command is then added to the **backgroundJobs array** along with its process ID and the indication that it is currently running (isRunning set to 1). The **numJobs counter** is incremented to reflect the addition of a new background job. The shell provides a mechanism to wait for the completion of all background jobs when the wait command is executed. This is achieved by iterating through **the backgroundJobs array** and using the **waitpid system call** to wait for each background job to finish.

The shell process dynamically creates a new pipe **(pipefd)** for each command that does not involve output redirection. There is a pipe between each command and the Shell process. The pipe system call is used within the **executeCommand** function to establish communication channels between the shell and the respective command process. In the case of no output redirection in the command, the read end of the pipe in the command process is closed, and the write end of the pipe is specified with the standard output of the command process.

Threads are created within the context of the shell process to handle the output of each command. These threads are spawned for each command that does not involve output redirection. The **pthread_create** function is used to create threads, and each thread is associated with a unique

listenerThreadArgs struct that is responsible for storing the thread id and the pipe that the command output from the command process will arrive.

With the creation of the thread, the **commandListenerThread** function is responsible for the thread that listens to the output of the command and prints to the console of the shell. This function is executed concurrently with the main shell process and allows the shell to print the output of commands as they execute. The purpose of the **commandListenerThread** function is to read the output of a command from a pipe and print it to the console. The function takes one parameter:

- void *args: A pointer to a structure **(struct ListenerThreadArgs**) containing information needed by the listener thread. The structure includes the file descriptor (pipefd) of the pipe from which the thread will read the command output and the thread ID (tid).

The use of mutexes inside the **commandListenerThread** function ensures thread safety during the printing process. The function provides a structured way to manage and synchronize the execution of listener threads within the shell.

Two mutexes are used in the shell: **printMutex and completedThreadsMutex**. The printMutex is employed to ensure thread-safe printing to the console. This mutex is acquired before printing to avoid conflicts when multiple threads attempt to write concurrently. The printMutex is acquired before performing any printing operations to stdout within the thread functions. This ensures that only one thread can access the printing resources at a time, preventing output from different threads from being interleaved. The completedThreadsMutex is employed to synchronize access to the completedThreads counter. This counter keeps track of the number of threads that have completed execution. The mutex ensures that the increment operation on the counter is atomic, preventing race conditions.