EECE7376 Sp24 Course Project [35325.202430]

Qucheng Jiang 001569593
Shihan Zhao 002772845

Part 1 – Shell

**** To build and run this pseudo shell, you need to follow the instructions located in Readme.md under the project folder.***
or you can download all file from https://github.com/Moran0103/Psh.git

In this part, we implement a simple version of the shell program relying on the string operations implemented in the previous assignment. Next the functions that the shell can achieve will be explained step by step:

1. The shell should print a $ prompt symbol followed by a space when it is ready to receive a new command from the user.
   CShell::run() shows that the shell will print a $ when it is ready to receive a new command by using ***while(true)*** function.

2. 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. A command is considered to be running in the foreground when operator & is not included at the end of the command line.
   To approach this goal, this project set two pipes in the function ***CShell::executeCommand().*** In this function, the system will collect child PIDs and wait for the last one finished. And then print a $.

```
$ ls -l | wc
      9      74     455
```

3. When a command is launched in the background using the suffix &, the shell should print the pid of the process corresponding to the last sub-command. The pid should be printed in square brackets. It should then immediately display the prompt and accept new commands, even if any of the child processes are still running.
   Use ***CShell::isBackgroundCommand*** to detect whether the last symbol is &. If the judgment is true, execute the conditional statement in ***CShell::executeCommand()*** and output the pid number first.

```
(base) enthalpy@DESKTOP-11R4F1B:~/7376/EECE7376/Prj/p1/psh$ make run
g++ -Wall -g  -Iinc -c src/ast/CPipelineNode.cpp -o obj/ast/CPipelineNode.o
g++ -Wall -g  -Iinc -c src/ast/CCommandNode.cpp -o obj/ast/CCommandNode.o
g++ -Wall -g  -Iinc -c src/ast/CPshParser.cpp -o obj/ast/CPshParser.o
g++ -Wall -g  -Iinc -c src/ast/CScriptNode.cpp -o obj/ast/CScriptNode.o
g++ -Wall -g  -Iinc -c src/CShell.cpp -o obj/CShell.o
g++ -Wall -g  -Iinc -c src/main.cpp -o obj/main.o
g++ -Wall -g  -o bin/psh obj/ast/CPipelineNode.o obj/ast/CCommandNode.o obj/ast/CPshParser.o obj/ast/CScriptNode.o ob
j/CShell.o obj/main.o
bin/psh
$ ls  -l | wc &
ls: cannot access ' -l': No such file or directory
[2371]

$       0       0       0
```

4. When a sub-command is not found, a proper error message should be displayed, immediately followed by the next prompt.
   This part is the program to be executed by accessing the command line through ***execvp()*** in ***CShell::executeCommand()***. If not found, -1 will be returned, which means that the system may have an error and the system will start to return **not found** information.

```
$ hello
Command not found: hello
```

5. The input redirection operator < should redirect the standard input of the first sub-command of a command. If the input file is not found, a proper message should be displayed.
   In this program, the CShell::redirectIO() function has the functionality to manage by redirecting the standard input of a command to read from a specified file. This code will parse the command line, redirect it in the child process, and detect errors. The error file processing process of this code is: 1. Open the file; 2. Error check; 3. Print the error reason; 4. Exit.

```
$ wc < hello.txt
 0  2 11

$ wc hello_word.txt
wc: hello_word.txt: No such file or directory
```

6. The output redirection operator > should redirect the standard output of the last sub-command of a command. If the output file cannot be created, a proper message should be displayed.
   The specific steps are the same as in section 5.

```
$ ls  -l > invalid/path
fd open: No such file or directory

$ ls -l src
total 16
-rw-r--r-- 1 enthalpy enthalpy 7978 Apr 12 22:29 CShell.cpp
drwxr-xr-x 2 enthalpy enthalpy 4096 Apr 12 22:29 ast
-rw-r--r-- 1 enthalpy enthalpy   89 Apr 12 22:29 main.cpp

$ ls -l >src
fd open: Is a directory

$ cat src
cat: src: Is a directory

$ ls
Makefile  README.md  bin  hello.txt  inc  obj  src  srcs  unittest

$ ls -l > hello.txt

$ cat hello.txt
total 32
-rw-r--r-- 1 enthalpy enthalpy 1295 Apr 12 22:29 Makefile
-rw-r--r-- 1 enthalpy enthalpy 1193 Apr 12 22:29 README.md
drwxr-xr-x 2 enthalpy enthalpy 4096 Apr 12 22:35 bin
-rw-r--r-- 1 enthalpy enthalpy    0 Apr 13 13:53 hello.txt
drwxr-xr-x 3 enthalpy enthalpy 4096 Apr 12 22:29 inc
drwxr-xr-x 3 enthalpy enthalpy 4096 Apr 12 22:30 obj
drwxr-xr-x 3 enthalpy enthalpy 4096 Apr 12 22:29 src
-rw------- 1 enthalpy enthalpy  509 Apr 13 04:39 srcs
```

Part 2

Asymmetric Multiprocessing & OS Scheduling

Our YouTube link: **EECE7376 Sp24 - Asymmetric Multiprocessing (ASMP) & OS Scheduling**

Large and small core technology is a heterogeneous multi-core processing technology proposed by ARM. Within each chip, there are better-performing processors, called big cores, and more energy-efficient processors, called little cores. Large cores are generally used to handle computationally intensive tasks, such as games. Small cores are used to handle tasks that take a long time to run but require a small amount of calculation, such as calls and videos. Based on different tasks, using different cores for processing can ensure that the computer can perform high-performance computing while reducing daily energy consumption as much as possible. This is the original intention of the large and small core technology design.

Judging from the development history of large and small cores, large and small cores are divided into three types: Cluster Switching, In kernel Switching (IKS), and Heterogeneous switching (HMP). Among them, cluster switching refers to a cluster for large cores and a cluster for small cores. Every time you switch between large and small cores, you need to migrate all processes on the large core to the small core, or migrate all processes on the small core to the large core. Only one cluster of the two can be running at the same time. The disadvantages of this are the high cost of transfer and the difficulty of scheduling. Because the running program of the computer is relatively complex, it is difficult to find a time point where there is only work with a large amount of calculation or work with a small amount of calculation. Therefore, this method of only allowing one cluster at a time is unreasonable.

In IKS, Little and big cores are split into pairs. The operating system will detect the performance and consumption required by the program during the running of the process in order to allocate appropriate cores according to the needs of its process. Compared with cluster switching, this method allows large cores and small cores in the CPU to run at the same time. However, in each pair, only one core can be running at the same time. This leads to the fact that there are always general cores in the CPU that are not working. This still greatly affects the efficiency of the CPU.

Nowadays, the large and small core structure successfully used on ARM, INTEL, APPLE M1, and M2 chips is Heterogeneous switching (HMP). HMP allows all cores to run simultaneously. This obviously greatly improves CPU utilization and efficiency. But for different tasks, core allocation has become the main issue that needs to be considered in this architecture. In the next part, we will focus on explaining the scheduler of the large and small core architecture.

The Energy Aware Scheduling is used in the big little core architecture. Energy Aware

Scheduling (EAS) fundamentally alters the assignment of Completely Fair Scheduler (CFS) tasks to CPUs by utilizing the Energy Model (EM) to make informed decisions. The EM's purpose is to enable the scheduler to assess the consequences of its decisions, using a much simpler design to minimize the impact on scheduler latency. This model aids in choosing the most energy-efficient CPU for task execution without compromising system performance, using knowledge of CPU capacities and energy costs.

During task wake-up, EAS leverages the EM along with Per-Entity Load Tracking (PELT) signals to select an optimal CPU. The function select_task_rq_fair() invokes find_energy_efficient_cpu(), which identifies the CPU with the highest available capacity (remaining capacity after current utilizations) in each performance domain—this choice helps maintain lower operational frequencies, thereby saving energy. The decision process evaluates if moving the task to a new CPU will be more energy-efficient compared to its previous CPU.

The compute_energy() function estimates the energy impact of migrating the waking task, simulating the new task distribution across CPUs. The em_pd_energy() API from the EM framework calculates the expected energy consumption for each performance domain based on this simulated distribution.