OS-344 Assignment-2A

Instructions

- **This assignment has to be done in a group.**
- **Group members should ensure that one member submits the completed assignment within the deadline.**
- **Each group should submit a report, with relevant screenshots/ necessary data and findings related to the assignments.**
- **We expect a sincere and fair effort from your side. All submissions will be checked for plagiarism through a software and plagiarised submissions will be penalised heavily, irrespective of the source and copy.**
- **There will be a viva associated with assignment. Attendance of all group members is mandatory.**
- **Assignment code, report and viva all will be considered for grading.**
- **Early start is recommended, any extension to complete the assignment will not be given.**

Task 1: Improving the console

Modern consoles support many features that help users work more efficiently. In the first part of the assignment, you will implement two basic features that will hopefully allow you a more efficient workflow when testing your code.

In the first part you will extend the kernel to support Caret navigation (text cursor navigation).

In the second part you will add partial support of "Shell History Ring".

1.1 Caret navigation

In the current xv6 console, pressing the keyboard keys ← or → will result in their appropriate ASCII signs to appear on the console. When working inside your Linux terminal those keys will cause the caret to move to the appropriate location (left or right) allowing the user to edit what he wrote more efficiently.

In the first part of your assignment, you will need to implement caret navigation.

Notice:

- End line must result in moving to the next line no matter where the caret is and it’s ascii value entered at the end of the buffer.
- When editing from the middle of a buffer you must shift the text accordingly.

1.2 Shell History Ring

History of past shell commands allows terminal users to evaluate multiple requests very fast without writing the entire command. In this part of the assignment, you will have to implement the history feature and the ability to easily update the console to fit the needed history. In modern operating

systems the history is implemented in the shell, to allow for a simple implementation you will implement history in kernel. Your implementation should support a maximum of 16 commands. To do so you can add: #define MAX_HISTORY 16 to your code.

Once history is implemented, we need a way to access the history. You will implement two mechanisms to do so:

1) The ↑ / ↓ keys will need to retrieve the next / last item in the history respectively. The item retrieved should now appear in the console.

2) Add a history **system call**:

int history (char * buffer, int historyId)

Input:

char * buffer - a pointer to a buffer that will hold the history command,

Assume max buffer size 128.

historyId - The history line requested, values 0 to 15

Output:

0 - History copied to the buffer properly

1 - No history for the given id

2 - historyId illegal

Once this is implemented add a “history” command to the shell user program (see sh.c) so that it upon writing the command a full list of the history should be printed to screen like in common.

● A good place to start for both 1.1 and 1.2 is the **console.c** file.

● Notice that this feature will only work on the QEMU console and not on the terminal. Running QEMU in noxmode for ssh is not advised while testing part 1.

Task 2: Statistics

In future tasks you will implement various scheduling policies. However, before that, we will implement an infrastructure that will allow us to examine how these policies affect performance under different evaluation metrics.

The first step is to extend the proc struct (see **proc.h**). Extend the proc struct by adding the following fields to it: **ctime**, **stime**, **retime** and **rutime**. These will respectively represent the

creation time and the time the process was at one of following states: SLEEPING, READY(RUNNABLE) and RUNNING.

➢ **Tip:** These fields retain sufficient information to calculate the turnaround time and waiting time of each process.

Upon the creation of a new process the kernel will update the process's creation time. The fields (for each process state) should be updated for all processes whenever a clock tick occurs (you can assume that the process' state is SLEEPING only when the process is waiting for I/O). Finally, care should be taken in marking the termination time of the process (note: a process may stay in the 'ZOMBIE' state for an arbitrary length of time. Naturally this should not affect the process' turnaround time, wait time, etc.). Since all this information is retained by the kernel, we are left with the question of extracting this information and presenting it to the user. To do so, create a new system call wait2 which extends the wait system call:

int wait2(int *retime, int *rutime, int *stime)

Input:

int * retime / *rutime / *stime - pointers to an integer in which wait2 will assign:

***retime** : The aggregated number of clock ticks during which the process **waited**

***rutime** : The aggregated number of clock ticks during which the process was **running**

***stime** : The aggregated number of clock ticks during which the process was **waiting for I/O** (was not able to run).

Output:

Pid - of the terminated child process - if successful

1 - upon failure

Submission instructions

- Place all the files including Makefile that you modified for each part into separate subfolders named as Task[1 or 2] and put all of them into a zip file, with the name being your group number (say, G10.zip).
- Create a patch of your modified files.
- report.pdf should contain a detailed description of all of your implementation including screenshots of code and output.

- Further, you must describe your test cases in some detail, and the observations you made from them.

--End of Assignment-2A--