Lab Assignment 5

Part1 : Maintaining History of commands in xv6

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 is the **console.c** file.
- Notice that this features will only work on the QEMU console and not on the terminal. Running QEMU in nox mode

Part2 : Statistics

In this part 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*.

Note : 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' 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 - pointer to an integer in which wait_2 will assign**:**

The aggregated number of clock ticks during which the process waited (was able to run but did not get CPU) The aggregated number of clock ticks during which the process was running. 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