7376 HW4

Name: Xuebao Zhao NUID: 002108354

Problem1:

To add a new system call to xv6 to obtain the system date and time, we need to perform the following steps:

1. 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.

2. 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

3. A new entry is added to the syscalls array in the “syscall.c” file and declare the “sys_date()” function using the extern keyword. The “syscalls” array is used to map system call numbers to their corresponding handler functions in the kernel.

4. The “sys_date()” function is defined in “sysproc.c” file. This function reads its arguments from the user stack using the argptr() function, which is provided by xv6 to fetch a pointer argument from the user stack. It will then call the cmostime() function defined in lapic.c to obtain the current date and time from the system clock.

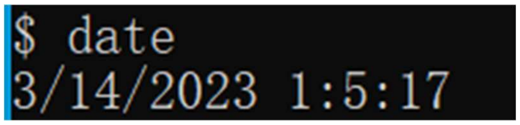
5. Added the user-space wrapper function for the “date()” system call in the “user.h” file. The user-space wrapper function is a C function that provides a convenient interface for user-space programs to invoke the system call.

Problem2:

In the main() function, create an instance of the “rtcdate” structure and call the date() system call to populate it with the current date and time. Check if the date() system call succeeded, and if not, print an error message and exit the program with a non-zero status code. Print the date and time in the desired format using the printf() function.

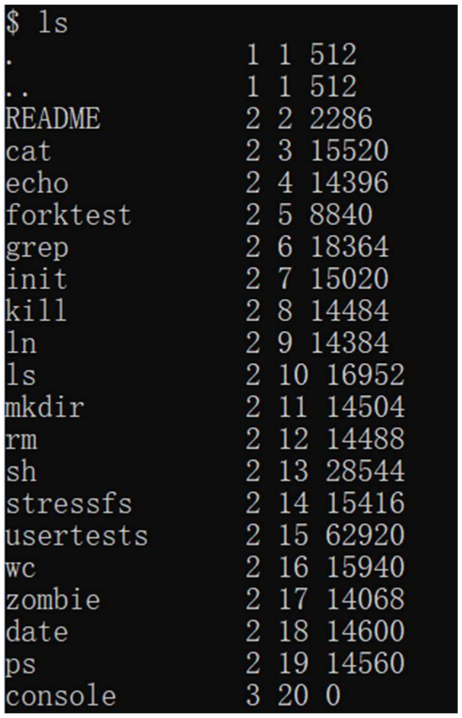
Result:

The program is invoked from the xv6 shell without any arguments, and provides the current date and time, in the format mm/dd/yyyy hh:mm:ss as follows:



```
$ date
3/14/2023 1:5:17
```

The new date user program is visible from the xv6 shell when running command ls. It is as shown below:



```
$ ls
.              1 1 512
..             1 1 512
README         2 2 2286
cat            2 3 15520
echo           2 4 14396
forktest       2 5 8840
grep           2 6 18364
init           2 7 15020
kill           2 8 14484
ln             2 9 14384
ls             2 10 16952
mkdir          2 11 14504
rm             2 12 14488
sh             2 13 28544
stressfs       2 14 15416
usertests      2 15 62920
wc             2 16 15940
zombie         2 17 14068
date           2 18 14600
ps             2 19 14560
console        3 20 0
```

Problem3:

The basic steps are the same as Problem 1 and Problem 2, but here I put the implementation in the file “proc.c”, because the ptable is created in this file, and it can be called directly in “proc.c” without passing it to “sysproc.c”.

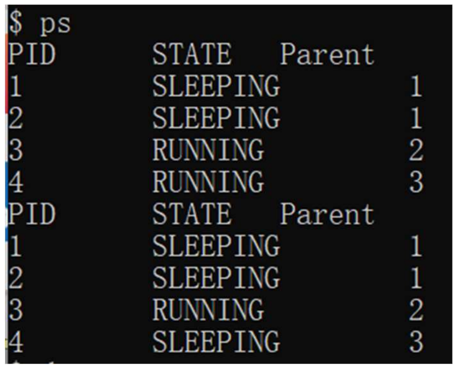
The function starts by declaring a pointer to a struct proc, which is presumably a data structure representing a process in the kernel. Then acquires a lock on the kernel's process table by calling acquire(&ptable.lock). This is done to ensure that the process table is not modified while the function is iterating over it. The function then prints out a header line indicating the format of the output it will generate. Then loops over each process in the process table, skipping over any processes that are in the UNUSED state. For each process that is not UNUSED, the function prints out information about the process, including its PID, its state (which could be EMBRYO, SLEEPING, RUNNABLE, RUNNING, or ZOMBIE), and the PID of its parent process (if it has one).

In “ps.c”, the main() function first calls “fork()” to create a new child process. If it returns 0, the code inside the if-statement is executed, which means that the process is the child process. In this case, the child process calls the “sleep()” function to sleep for 200 ticks.

If fork() returns a negative value, the code inside the else-if statement is executed, which means that an error occurred while forking.

If fork() returns a positive value, the code inside the else statement is executed, which means that the process is the parent process. The parent process calls getprocs() to display information about the running processes, sleeps for 10 ticks, and then calls getprocs() again to display the updated information. Finally, the parent process calls wait() to wait for the child process to complete before exiting.

Result:



```
$ ps
PID        STATE    Parent
1          SLEEPING        1
2          SLEEPING        1
3          RUNNING         2
4          RUNNING         3
PID        STATE    Parent
1          SLEEPING        1
2          SLEEPING        1
3          RUNNING         2
4          SLEEPING        3
```