Northeastern University
College of Engineering
Department of Electrical & Computer Engineering

EECE7376: Operating Systems: Interface and Implementation

Homework 4

Problem 1 (30 Points)

Add a new system call to xv6 to obtain the system date and time. The interface for this system call is the following:

```
struct rtcdate
{
        uint second;
        uint minute;
        uint hour;
        uint day;
        uint month;
        uint year;
};
int date(struct rtcdate *d);
```

As shown above, the system call takes a structure of type `struct rtcdate` as its only argument. The definition of the `struct rtcdate` is already available in file `date.h`. The `date` system call should always return 0.

The list below shows the xv6 files affected by the addition of a new system call. Follow these steps in your implementation:

- `syscall.h`. Add a unique numeric identifier for the new system call.
- `usys.S`. Add a new line of code corresponding to the new system call, following the pattern you observe for existing system calls. The new line of code includes the use of 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`.

- `syscall.c`. Add a new entry to the `syscalls` array. Also, add the "extern" declaration of function `sys_date(void)`. The body of the system call reads its arguments from the user, and not from the currently active kernel stack. Therefore, the function implementing the system call takes no explicit arguments in its header.
- `sysproc.c`. Use this kernel source file to append the implementation of the new system call. The body of the system call needs to grab the arguments from the user stack. Function `argint()` is an example of a helper function provided by xv6 for this purpose, which fetches a 32-bit value from the user stack and interprets it as an integer. You can see an example of its invocation in function `sys_kill()`, also available in this file.

  However, in our case, we are interested in interpreting the argument passed by the user as a pointer. For this purpose, you may use function `argptr()` instead. In the body of the system call, you can obtain the current date and time with a call to kernel-level function `cmostime()`, defined in `lapic.c`. This function accesses the system clock directly using privileged I/O instructions.
- `user.h`. Add the user prototype of the new system call here. This file is a user-level header file, analogous to the header files provided by LibC in a modern Linux distribution (e.g., `stdio.h`). One of the purposes of this file is providing the prototype for every available system call.

At this point, you may check that your code compiles correctly, but you need to continue to work on the following exercise before you can test the system call.

I modified all files mentioned above.

```
Booting from Hard Disk..xv6...
cpu0: starting 0
sb: size 1000 nblocks 941 ninodes 200 nlog 30 logstart 2 inodestart 32 bmap start 58
init: starting sh
$ date
3/16/2024 3:36:7
$
```

Problem 2 (20 Points)

Create a new user program named `date.c` that serves as an **xv6 command-line** tool to obtain the result of the `date()` system call and print the date info on the screen. The program should

be invoked from the xv6 shell without any arguments, and should provide the current date and time, in the format `mm/dd/yyyy hh:mm:ss` as follows:

```
$ date
5/4/2015 20:38:6
```

Your new `date` user program should be visible from the xv6 shell when running command `ls`. Test the new `date` command and verify that the output matches the sample output above.

```
Booting from Hard Disk..xv6...
cpu0: starting 0
sb: size 1000 nblocks 941 ninodes 200 nlog 30 logstart 2 inodestart 32 bmap start 58
init: starting sh
$ date
3/16/2024 3:36:7
$ ls
.              1 1 512
..             1 1 512
README         2 2 2286
cat            2 3 15552
echo           2 4 14436
forktest       2 5 8880
grep           2 6 18396
init           2 7 15056
kill           2 8 14520
ln             2 9 14416
ls             2 10 16984
mkdir          2 11 14544
rm             2 12 14524
sh             2 13 28580
stressfs       2 14 15452
usertests      2 15 62952
wc             2 16 15980
zombie         2 17 14100
date           2 18 14660
ps             2 19 14644
console        3 20 0
$
```

Problem 3 (25 Points)

Following the same approach that you used in solving Problems 1 and 2, add a new system call and a user command to xv6 to allow users to call a command line similar to the **ps** command in

Linux. However, your xv6 **ps** command should not take any arguments. The command lists all the **id** of the current processes in the system. For each process, the command also displays the process **state** and the **id** of its parent. When testing it, the command should list the information of at least two processes: the shell process and the process of the **ps** user command itself.

For a testing purpose, add the needed code in your user command **ps** program to fork a child and allow the code in the parent to perform the main **ps** task. Now your **ps** command should also display the status of this child. Write your code so that the child is presented in different states.

```
$
$ ps
PID[PPID] Pri STATE  NAME
1[0]      (0) sleep  init
2[1]      (1) sleep  sh
6[2]      (1) run    ps
7[6]      (0) sleep  ps
$
```

Problem 4 (25 Points)

Modify the default round-robin (RR) scheduler implemented in xv6 to test a scheduling policy that gives higher priority to processes with an **odd** `pid`. The scheduler runs processes with an **even** `pid` only if no processes with odd pid are in the `RUNNABLE` state. It uses RR to run the `RUNNABLE` processes with the same priority level.

Show how your scheduler is affected by this modification by adding a user program to xv6 to test the new scheduler. The program needs to fork multiple processes. Utilize some programming techniques, such as `sleep()` and long running loops, to simulate a situation to how your new scheduler works differently than the default RR scheduler. You might need to utilize the xv6 `wait()` system call to collect information about the order on which processes are completed.

Note: The `pid` of a new process is returned from the `fork()` function that creates the process. The `fork()` function calls the `allocproc()` function. One of the tasks of the `allocproc()` function is to assign a new `pid` to the new process. Both functions are defined in `proc.c`

```
Booting from Hard Disk..xv6...
cpu0: starting 0
sb: size 1000 nblocks 941 ninodes 200 nlog 30 logstart 2 inodestart 3
init: starting sh
$ HW4Q4
PID[PPID] Pri STATE  NAME
1[0]      (0) sleep  init
2[1]      (1) sleep  sh
3[2]      (0) run    HW4Q4
4[3]      (1) runble HW4Q4
5[4]      (0) runble HW4Q4
6[4]      (1) runble HW4Q4
7[4]      (0) runble HW4Q4
8[5]      (1) runble HW4Q4
9[5]      (0) runble HW4Q4
10[8]     (1) runble HW4Q4
11[6]     (0) runble HW4Q4
PID[PPID] Pri STATE  NAME
1[0]      (0) sleep  init
2[1]      (1) sleep  sh
3[2]      (0) run    HW4Q4
4[3]      (1) runble HW4Q4
5[4]      (0) runble HW4Q4
6[4]      (1) runble HW4Q4
7[4]      (0) runble HW4Q4
8[5]      (1) runble HW4Q4
9[5]      (0) sleep  HW4Q4
10[8]     (1) runble HW4Q4
11[6]     (0) sleep  HW4Q4
PID[PPID] Pri STATE  NAME
1[0]      (0) sleep  init
2[1]      (1) sleep  sh
3[2]      (0) run    HW4Q4
4[3]      (1) runble HW4Q4
5[4]      (0) runble HW4Q4
6[4]      (1) runble HW4Q4
7[4]      (0) runble HW4Q4
8[5]      (1) runble HW4Q4
9[5]      (0) sleep  HW4Q4
10[8]     (1) runble HW4Q4
11[6]     (0) sleep  HW4Q4
PID[PPID] Pri STATE  NAME
1[0]      (0) sleep  init
2[1]      (1) sleep  sh
3[2]      (0) run    HW4Q4
4[3]      (1) runble HW4Q4
5[4]      (0) runble HW4Q4
6[4]      (1) runble HW4Q4
7[4]      (0) runble HW4Q4
8[5]      (1) runble HW4Q4
9[5]      (0) sleep  HW4Q4
10[8]     (1) runble HW4Q4
11[6]     (0) sleep  HW4Q4
PID[PPID] Pri STATE  NAME
1[0]      (0) sleep  init
2[1]      (1) sleep  sh
3[2]      (0) run    HW4Q4
4[3]      (1) runble HW4Q4
5[4]      (0) runble HW4Q4
6[4]      (1) runble HW4Q4
7[4]      (0) runble HW4Q4
8[5]      (1) runble HW4Q4
9[5]      (0) sleep  HW4Q4
10[8]     (1) runble HW4Q4
11[6]     (0) sleep  HW4Q4
PID[PPID] Pri STATE  NAME
1[0]      (0) sleep  init
2[1]      (1) sleep  sh
3[2]      (0) run    HW4Q4
4[3]      (1) runble HW4Q4
5[4]      (0) runble HW4Q4
6[4]      (1) runble HW4Q4
7[4]      (0) runble HW4Q4
8[5]      (1) runble HW4Q4
9[5]      (0) sleep  HW4Q4
10[8]     (1) runble HW4Q4
11[6]     (0) sleep  HW4Q4
PID[PPID] Pri STATE  NAME
1[0]      (0) sleep  init
2[1]      (1) sleep  sh
3[2]      (0) run    HW4Q4
4[3]      (1) runble HW4Q4
5[4]      (0) runble HW4Q4
6[4]      (1) runble HW4Q4
7[4]      (0) runble HW4Q4
8[5]      (1) runble HW4Q4
9[5]      (0) sleep  HW4Q4
10[8]     (1) runble HW4Q4
11[6]     (0) sleep  HW4Q4
PID[PPID] Pri STATE  NAME
1[0]      (0) sleep  init
2[1]      (1) sleep  sh
3[2]      (0) run    HW4Q4
4[3]      (1) runble HW4Q4
5[4]      (0) runble HW4Q4
6[4]      (1) runble HW4Q4
7[4]      (0) runble HW4Q4
8[5]      (1) runble HW4Q4
9[5]      (0) sleep  HW4Q4
10[8]     (1) runble HW4Q4
11[6]     (0) sleep  HW4Q4
PID[PPID] Pri STATE  NAME
1[0]      (0) sleep  init
2[1]      (1) sleep  sh
3[2]      (0) run    HW4Q4
4[3]      (1) runble HW4Q4
5[4]      (0) runble HW4Q4
6[4]      (1) runble HW4Q4
7[4]      (0) runble HW4Q4
8[5]      (1) runble HW4Q4
9[5]      (0) sleep  HW4Q4
10[8]     (1) runble HW4Q4
11[6]     (0) sleep  HW4Q4
zombie!
zombie!
zombie!
PID[PPID] Pri STATE  NAME
1[0]      (0) sleep  init
2[1]      (1) sleep  sh
3[2]      (0) run    HW4Q4
4[3]      (1) zombie HW4Q4
PID[PPID] Pri STATE  NAME
1[0]      (0) sleep  init
2[1]      (1) sleep  sh
3[2]      (0) run    HW4Q4
4[3]      (1) zombie HW4Q4
PID[PPID] Pri STATE  NAME
1[0]      (0) sleep  init
2[1]      (1) sleep  sh
3[2]      (0) run    HW4Q4
4[3]      (1) zombie HW4Q4
$
```

I made priority as attribute for each process. 0 as high priority, 1 as low. As we can see in the picture, PID with its num is odd, that has a higher priority, get execution time first. (Going to sleep state), and then zombie first. This meaning the scheduler pick those process with high priority to run first, then the one with lower priority.