Project 1
System Calls

Introduction

In this project, you will become familiar with:

1. Using conditional compilation.
2. The xv6 system call invocation path.
3. Implementing a new system call.
4. Creating a new shell command.
5. The xv6 process structure.
6. Control sequences for the xv6 console.
7. The format and content for project reports.

The reading for this project is chapters 1 – 3 from the xv6 book.

System Call Tracing

Your first task is to modify the xv6 kernel to print out a line for each system call invocation. It is enough to print the name of the system call and the return value; you don't need to print the system call arguments. When you're done, you should see output like this when booting xv6:

```
...
fork -> 2
exec -> 0
open -> 3
close -> 0
$write -> 1
 write -> 1
```

That's the `init` process using the `fork()` system call to create a new process and the `exec()` system call to run the shell (sh) followed by the shell writing the $ prompt. The last two lines show the output from the shell and the kernel being intermixed. This is a *concurrency* issue; more about that in class.

This new and wonderful feature of your kernel will cause a trace to be printed every time a system call is invoked. You will find this "feature" **annoying**, so you want to be able to turn this feature on or off as necessary. You are required to implement your syscall tracing facility (code and data structures) using *conditional compilation* so that it can easily be turned on or off with a simple compilation flag. The flag name is `PRINT_SYSCALLS`. Uncomment the declaration in the `Makefile` to turn this flag on.

Hint: Modify the `syscall()` function in `syscall.c` and consider adding a second table (array) similar to one for system call function pointers just above the `syscall()` function (we suggest you name it "syscallnames"). Be sure to print the system call name (e.g. "date") and not the kernel-side routine name (e.g., "sys_date"). Use conditional compilation so that the code and data structure are only included in the compiled code if the flag is turned on in the Makefile.

For the x86 architecture, the return code is the value of the `%eax` register on return from the callee.

The `date()` System Call

Your second task is to add a new system call[1] to xv6. The goal of the exercise is for you to become familiar with the principle parts of the system call infrastructure. Your new system call will get the current UTC time and return it to the invoking user program. You must use the function, `cmostime()`[2], defined in lapic.c, to read the real-time clock. The supplied include file `date.h` contains the definition of the `rtcdate` data structure, a pointer to which you will provide as an argument to `cmostime()`.

You will need a shell command that invokes your new `date()` system call and we have provided one in the file `date.c`. You can also write your own, but since this command is so simple, we provide it as an example.

Adding a New System Call

Adding a new system call to your xv6 kernel, while straightforward, requires that several files be modified. The following files will need to be modified in order to add the `date()` system call. Note that all other system calls in xv6 are defined in a similar manner.

- **user.h** contains the function prototypes for the xv6 system calls and library functions[3]. This file is required to compile user programs that invoke these routines. The function prototype for `date()` is `int date(struct rtcdate*);`

- **usys.S** contains the list of system calls made available (exported) by the kernel. We recommend that you add any new system calls at the end of this list as it will make it easier for course staff to find your additions should they need to help debug some problem.

  This file contains the code that *actually invokes* your system call[4]. Be sure to understand how the code here will cause the system to change mode to kernel for your process.

- **syscall.h** contains the mappings of system call *name* to system call *number*. This is a critical part of adding a system call. In xv6, we require that the kernel entry point for system calls be named with the `SYS_` prefix. This is a very good technique for a kernel developer to use. Use the next integer value for the system call number of any new system call that you add. It should be clear, at this point, that system calls are actually invoked via number and not name.

- **syscall.c** is where the system call entry point will be defined. There are several steps to defining the entry point in this file.

  1. The entry point will point to the implementation. The implementation for our system call will be in another file. You indicate that with

  ```
  extern int sys_date(void);
  ```

  which you will add to the end of the list of similar `extern` declarations. **You must declare the routine to take a void argument!**

  2. Add to the syscall function definition. The table

  ```
  int (*syscalls[])(void) = {
  ```

  declares the mapping from symbol name (as used by `usys.S`) to the function name, which is the name of the function that you want to call for that symbol. You will want to add

[1] Having both the system call and test program named `date` can be confusing. We will always call a program that we run from the shell prompt a "command" or "program" and always call a system call that your programs invoke to obtain kernel services a "system call".

[2] The time resolution for `cmostime()` is one second.

[3] The files ulib.c, printf.c and ls.c comprise the C library for xv6. Pretty small.

[4] In xv6, a system call is invoked via a specific trap `int 0x40`

```
[SYS_date]    sys_date,
```

to the end of this structure. The routine `syscall()` at the bottom of `syscall.c` causes the correct system call to be invoked based on these definitions; see `vectors.pl`. You have just defined this system call number for the `date()` system call. The mapping in `syscalls[]` constitutes what is known as a *function dispatch table*.

- **sysproc.c** is where you will implement `sys_date()`. The implementation of your `date()` system call is very straightforward. All you will need to do is add the new routine to `sysproc.c`; get the argument off the stack[5]; call the `cmostime()` routine correctly; and return a code indicating SUCCESS or FAILURE, which will most likely be 0 because this routine cannot actually fail[6]. Note that since you passed into the kernel a *pointer* to the structure, any changes made to the structure within the kernel will be reflected in user space. The first part of your implementation is provided for you

```
int
sys_date(void)
{
  struct rtcdate *d;

  if(argptr(0, (void*)&d, sizeof(*d)) < 0)
    return -1;
```

The rest of the routine is left as an exercise for the student. See the xv6 book for how to properly use the `argptr()` routine.

The `date` Command

We cannot directly test a system call and instead have to write a program that we can invoke as a command at the shell prompt. An example program, `date.c` has been provided. This file has not yet been added to the xv6 build environment since the required system call does not exist yet.

The `date.c` program is designed to mimic "`date -u`" under Linux.

```
#include "types.h"
#include "user.h"
#include "date.h"

static char *months[] = {"NULL", "Jan", "Feb", "Mar", "Apr",
  "May", "Jun", "Jul", "Aug", "Sep", "Oct", "Nov", "Dec"};
static char *days[] = {"Sun", "Mon", "Tue", "Wed", "Thu", "Fri", "Sat"};

int
dayofweek(int y, int m, int d)
{
  return (d+=m<3?y--:y-2,23*m/9+d+4+y/4-y/100+y/400)%7;
}

int
main(int argc, char *argv[])
{
```

[5] This is why the function argument is `void`.

[6] Your date command, however, must check for an error and do something sane. More in class.

```
  int day;
  struct rtcdate r;

  if (date(&r)) {
    printf(2,"Error: date call failed. %s at line %d\n", __FILE__, __LINE__);
    exit();
  }

  day = dayofweek(r.year, r.month, r.day);

  printf(1, "%s %s %d", days[day], months[r.month], r.day);
  printf(1, " %d:%d:%d UTC %d\n", r.hour, r.minute, r.second, r.year);

  exit();
}
```

The `printf()` routine takes a file descriptor as its first argument: 1 is standard output (`stdout`) and 2 is error output (`stderr`). Use them appropriately.

The function `main()` **must** terminate with a call to `exit()` and not `return()` or simply fall off the end of the routine. This is a *very* common source of compilation errors.

In order to make your new date program available to run from the xv6 shell, add _date[7] to the UPROGS definition in the xv6 Makefile. Be sure to also add the correct entry to the `runoff.list` file as well.

CTL – P

The xv6 kernel supports a special control sequence, control – p, which displays process information on the console[8]. It is intended as a debugging tool and you will expand the information reported as you add new features to the xv6 process structure. Note that the routine `procdump()`, at the end of `proc.c`, implements the bulk of the control – p functionality.

You will modify `struct proc` in `proc.h` to include a new field, `uint start_ticks`. You will modify the routine `allocproc()` in `proc.c` so that this field is initialized to the value for the existing global kernel variable `ticks`. This allows the process to "know" when it was created. This will allow us to later calculate how long the process has been alive. Put the initialization code right before the return at the end of the routine. The PSU version of xv6 does not require a lock around accesses to the `ticks` global variable. If you look around you can figure out why.

You will then modify the `procdump()` routine to print out the amount of time that each process has been active in system. This is fairly straightforward: each "tick" represents $\frac{1}{100}$ of a second. Report the elapsed time in seconds and partial seconds. The longest running process in xv6 will always be the `init` process, but the first shell will be close behind. There is no need to modify the `printf()` or `cprintf()` routines to print this information.

The first line of output should be a set of labels for each column. You will be adding to the control–p output in later projects and the header will help with interpreting the information. The last set of information printed from `procdump()` contains the program counters as returned by the routine `getcallerpcs()`. The header for this last section can just be labeled "PCs".

Try to keep the code in `procdump()` as clean as you can. It may be useful for you to create a helper function so as not to clutter the code in that routine.

[7] Or whatever you name the file preceded by an underscore and drop the ".c" extension.

[8] See console.c, `case C('P'):`

Example Output

PID	State	Name	Elapsed	PCs
1	sleep	init	1.34	80104e82 80104c54 801066ba 80105786 80106ac9 801068ba
2	sleep	sh	2.45	80104e82 801009b6 80101f11 801011b7 80105960 80105786 80106ac9 801068ba

Before You Submit

As described in the *Survival Guide*, you must test that your submission is working properly before submission. There is a `Makefile` target to help you. Be sure to run `make dist-test-nox` to verify that your kernel compiles and runs correctly, including all tests. Once you have verified that the submission is correct, use `make tar` to build the archive file that you will submit. Submit your project report as a separate PDF document.