Project 1
System Calls

Introduction

In this project, you will become familiar with:

1. The xv6 Makefile
2. Conditional compilation
3. Implementing a new system call
4. Creating a new user command
5. The xv6 process structure
6. The control-p console command
7. The format and content for project reports

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

The conditional compilation flag for this project is **CS333_P1**.

The xv6 Makefile

The xv6 Makefile is structured so that all student modifications are at the top of the Makefile.

For this project, you will change "CS333_PROJECT ?= 0" to "CS333_PROJECT ?= 1". To activate the facility to trace system calls, you will change "PRINT_SYSCALLS ?= 0" to "PRINT_SYSCALLS ?= 1" and set the value back to 0 to turn off system call tracing.

The variable "CS333_UPROGS" is a listing of new shell command names (user programs), with a leading underscore. The variable "CS333_TPROGS" is where you list any test programs that you write for the assignment using the same naming scheme as "CS333_UPROGS". Both are used to update the Makefile variable "UPROGS".

As should be clear, the correct project compilation flags and correct "UPROGS" variable are set via the value of "CS333_PROJECT". You will need to correctly set this value in subsequent projects.

Conditional Compilation

You are required to write your code in such a way so that if the compilation flag CS333_PROJECT is turned off (set to 0) then xv6 will function the same as the initial distribution. This means that your code will need to be *wrapped* in preprocessor directives. For example:

```
#ifdef CS333_P1
int
sys_date(void)
{
  struct rtcdate *d;
  // code removed
}
#endif
```

The statement `#ifdef CS333_P1` is a GNU GCC preprocessor directive that indicates that the code between this line and the matching `#endif` will only be included in the compiled program if the flag is *turned on*; that is, defined. Preprocessor directives are defined in the project Makefile.

Caution: Preprocessor directives do not apply to assembly language programs (those ending in .S). Because of this, you cannot use a preprocessor directive in the file `usys.S`. Fortunately, the system is designed such that any extra code in this file will not impact xv6 functionality when conditional compilation flags are turned off. Because of this restriction on `usys.S`, conditional compilation cannot be used in the file `syscall.h` either. **These are the only two files where you will add code that is not subject to conditional compilation.**

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 completed, 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 invoked 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`.

Hint: Modify the `syscall()` function in `syscall.c` to do the printing. There is an array named `syscallnames[]` already defined in `syscall.c` that is enabled when the `PRINT_SYSCALLS` define is set in the Makefile. Use this structure and add the information for any new system calls that you create in the class so that system call tracing will print correctly.

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 principal 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()`.

[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.

You will need a shell command that invokes your new `date()` system call and we have provided one in the file `date.c`.

Adding a New System Call

Adding a new system call to xv6, while straightforward, requires that several files be modified. The following files will need to be modified in order to add the `date()` system call.

- **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 is

```
#ifdef CS333_P1
int date(struct rtcdate*);
#endif // CS333_P1
```

- **usys.S** contains the list of system calls made available (exported) by the kernel. Add any new system calls at the end of this list. Preprocessor directives cannot be used in this file.
- **syscall.h** contains the mappings of system call *name* to system call *number*. This is a critical part of adding a system call as system calls are invoked within the operating system by number, not name. Follow the existing pattern for adding new system call numbers.
- **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 the "extern" keyword.

```
#ifdef CS333_P1
// internally, the function prototype must be 'int' not 'uint' for sys_date()
extern int sys_date(void);
#endif // CS333_P1
```

  which you will add to the end of the list of similar `extern` declarations. As an example, we have included the conditional compilation directives here; they will be left out of examples going forward. **You must declare the routine as taking a void argument!** More on this in class.

  2. Add to the syscall function definition. The function dispatch table

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

  declares the mapping from symbol name (as used by `usys.S` and declared in `syscall.h`) to the function name, which is the name of the function that will be called for that symbol. Add

```
[SYS_date]    sys_date,
```

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

to the end of this structure.

The routine `syscall()`, at the bottom of `syscall.c`, invokes the correct system call based on these definitions; see `vectors.pl`. The data structure `syscalls[]` is an array of function pointers which is often called 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[4]; 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[5]. Note that since you passed into the kernel a *pointer* to the `rtcdate` 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(struct rtcdate)) < 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. Be sure to read and understand the comments for the `argptr()` routine in `syscall.c` as this will become critical in later projects.

- **defs.h** is where function prototypes for kernel-wide function calls, that are not in sysproc.c or sysfile.c, are defined. This file is not modified in project one but will be modified in subsequent projects.

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.

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

The `printf()` routine[6] takes a file descriptor as its first argument: 1 is standard output (`stdout`) and 2 is error output (`stderr`).

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.

The `date.c` and `date.h` files are already added to `runoff.list` so that the commands `make dist-test` and `make tar` will work correctly. In future projects, you will need to add new source code files to `runoff.list` yourself.

Control – P

The xv6 kernel supports a special control sequence, control – p, which displays process information on the console[7]. It is intended as a debugging tool and you will expand the information reported

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

[5]Your date command, however, must check for an error and do something reasonable.

[6]The `printf()` library routine in xv6 differs substantially from the Linux standard C library.

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

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 active. Put the initialization code right before the return at the end of the routine. Our version of xv6 does not require a lock around accesses to the `ticks` global variable[8].

You will then modify the `procdumpP1()` stub routine in `proc.c` to print out the amount of time that has passed since the process was allocated (i.e. how long it has been active, or the elapsed time), along with the other fields in the example output below. See `procdump()` to understand the context in which `procdumpP1()` is called, and for an example of how to handle the existing fields. Note that this routine handles printing the program counters (more below) at the end of each row for you.

This change is fairly straightforward, since each "tick" is a millisecond ($\frac{1}{1000}$ of a second)[9]. Report the elapsed time in seconds, accurate to the millisecond. 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. You will also modify the routine to print the size of the process. The process size is already part of the process structure, so you are just printing it out, not calculating it.

The first line of output is a set of labels for each column, see `procdump()` for the code that prints the headers. You will be adding to the control−p output in later projects and the header will help with interpreting the output. The last set of information printed from `procdump()` for each row contains the program counters as returned by the routine `getcallerpcs()`. The header for this last section is labeled "PCs".

Keep the code in `procdumpP1()` as clean as you can. You do not need to modify `procdump()`.

Example Output

```
PID     Name          Elapsed    State   Size     PCs
1       init          1.543      sleep   12288    80104bf3 80104968 80106560 80105767 801068f0 801
2       sh            1.499      sleep   16384    80104bf3 80100a37 80101f68 80101265 8010591e 801
```

For each subsequent project, you'll need to similarly modify the corresponding `procdumpP*()` stub to maintain the existing functionality and add support for new fields. Feel free to use helper functions or other refactoring techniques to keep the code as clean as possible.

Coding Style

There is more detail on coding style in the Survival Guide, which is required reading.

1. Indentation must use spaces, not tabs. Function declarations must be in correct form.

   You can check for leading tabs (a tab of a control sequence) in your code with the following regular shell command. Note that this only finds a control sequence if it is the first character on the line. This will not find all places where tabs are uses for indentation.

[8] This is just one area where the PDX version of xv6 differs from the version released by MIT.

[9] `ticks` is incremented approximately every millisecond by the timer interrupt, see `trap()` in `trap.c` if you are curious about this.

```
grep -P ^'\t' *.h *.c
```

Here is the output when run on our version of the xv6 source code (PDX_XV6)

```
$ grep -P ^'\t' *.h *.c
$
```

Note that no lines beginning with a control sequence were found.

2. For vim users, see the vim wiki for information on converting tabs to spaces. You can automatically set C-style indentation to 2 spaces and automatically convert tabs to 2 spaces with this entry in your .vimrc file

   ```
   :set smartindent
   set expandtab autoindent shiftwidth=2 tabstop=2
   ```

3. Code must match the general code style of xv6.

Required Tests

The list of required tests are in the project rubric.

Before You Submit

As described in the *Survival Guide*, you are required to test that your submission works properly before submission. There is a `Makefile` target to help you. You are **required** to run "`make dist-test`" to verify that your kernel compiles and runs correctly, including all tests. A common error is to not put the name of new files into `runoff.list`. 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. **Failure to submit your report in PDF format will result in a grade of zero for the report. No exceptions.**