xv6 Overview

CS 202: Advanced Operating Systems

UCR
UNIVERSITY OF CALIFORNIA, RIVERSIDE

xv6

- xv6 is MIT's re-implementation of Unix v6
  - Written in ANSI C
  - Runs on RISC-V and x86
    - We will use the RISC-V version with the QEMU simulator
  - Smaller than v6
  - Preserve basic structure (processes, files, pipes. etc.)
  - Runs on multicores
  - Got paging support in 2011

Ken Thompson & Dennis Ritchie, 1975

xv6

- To understand it, you'll need to read its source code
  - It's not that hard
  - Source code:
    - https://github.com/rtenlab/xv6-riscv (for our projects)
      - Forked from https://github.com/mit-pdos/xv6-riscv; to avoid unexpected updates during this course
  - Book/commentary
    - xv6: a simple, Unix-like teaching operating system
    - https://pdos.csail.mit.edu/6.828/2022/xv6/book-riscv-rev3.pdf

Why xv6?

- Why study an old OS instead of Linux, Solaris, or Windows?

1. Big enough
   - To illustrate basic OS design & implementation
2. Small enough
   - To be (relatively) easily understandable
3. Similar enough
   - To modern OSes
   - Once you've explored xv6, you will find your way inside kernels such as Linux

Why RISC-V?

RISC-V®

RISC-V: The Free and Open RISC Instruction Set Architecture

- RISC-V: open standard instruction set architecture (ISA) based on RISC principles
  - High quality, loyalty free, license free
  - Multiple proprietary and open-source core implementations
  - Supported by growing software ecosystem
  - Appropriate for all levels of computing system, from microcontrollers to supercomputers

- Fun to use toolchains for the new architecture

Apple shows interest in RISC-V chips, a competitor to iPhones' Arm tech

RISC-V chip technology could be used for tasks like AI and computer vision.

Intel Will Offer SiFive RISC-V CPUs on 7nm, Plans Own Dev Platform

By Joel Hruska on June 24, 2021 at 8:36 am | Comments

xv6 Structure

- Monolithic kernel
  - Provides services to running programs
- Processes uses system calls to access system services
- When a process call a system call
  - Execution will enter the kernel space
  - Perform the service
  - Return to the user space

xv6 System Calls

System call	Description
fork()	Create process
exit()	Terminate current process
wait()	Wait for a child process to exit
kill(pid)	Terminate process pid
getpid()	Return current process's id
sleep(n)	Sleep for n seconds
exec(filename, *argv)	Load a file and execute it
sbrk(n)	Grow process's memory by n bytes
open(filename, flags)	Open a file; flags indicate read-/write

xv6 System Calls (2)

System call	Description
`read(fd, buf, n)`	Read n bytes from an open file into buf
`write(fd, buf, n)`	Write n bytes to an open file
`close(fd)`	Release open file fd
`dup(fd)`	Duplicate fd
`pipe(p)`	Create a pipe and return fd's in p
`chdir(dirname)`	Change the current directory
`mkdir(dirname)`	Create a new directory
`mknod(name, major, minor)`	Create a device file
`fstat(fd)`	Return info about an open file
`link(f1, f2)`	Create another name (f2) for the file f1
`unlink(filename)`	Remove a file

xv6 kernel source files

- /kernel directory

bio.c	Disk block cache for the file system.
console.c	Connect to the user keyboard and screen.
entry.S	Very first boot instructions.
exec.c	exec() system call.
file.c	File descriptor support.
fs.c	File system.
kalloc.c	Physical page allocator.
kernelvec.S	Handle traps from kernel, and timer interrupts.
log.c	File system logging and crash recovery.
main.c	Control initialization of other modules during boot.
pipe.c	Pipes.
plic.c	RISC-V interrupt controller.
printf.c	Formatted output to the console.
proc.c	Processes and scheduling.
sleeplock.c	Locks that yield the CPU.
spinlock.c	Locks that don't yield the CPU.
start.c	Early machine-mode boot code.
string.c	C string and byte-array library.
swtch.S	Thread switching.
syscall.c	Dispatch system calls to handling function.
sysfile.c	File-related system calls.
sysproc.c	Process-related system calls.
trampoline.S	Assembly code to switch between user and kernel.
trap.c	C code to handle and return from traps and interrupts.
uart.c	Serial-port console device driver.
virtio_disk.c	Disk device driver.
vm.c	Manage page tables and address spaces.

Setup

- Toolchain
  - You need a RISC-V tool chain and QEMU for RISC-V
- Linux (Ubuntu 20.04)

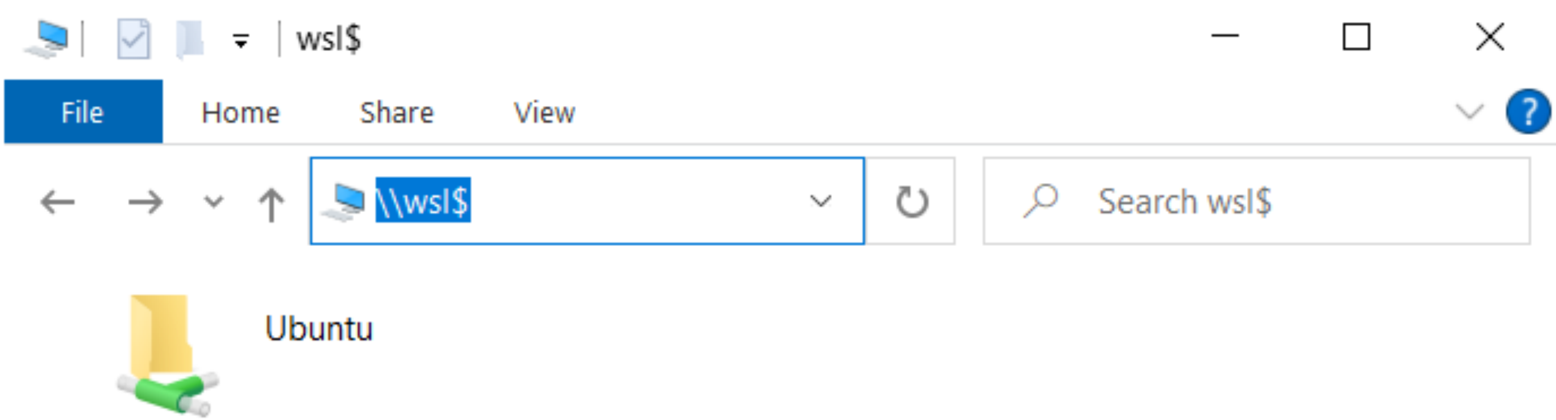
```
$ sudo apt update
$ sudo apt install git build-essential gdb-multiarch qemu-system-misc gcc-riscv64-linux-gnu binutils-riscv64-linux-gnu
```

- Windows
  - You can use Windows Subsystem for Linux (WSL) with Ubuntu 20.04
  - Unsure what version of Ubuntu you have? Open WSL terminal and type “lsb_release -a”
  - Follow the above Linux instruction for package installation

```
~$ lsb_release -a
No LSB modules are available.
Distributor ID: Ubuntu
Description:    Ubuntu 20.04 LTS
Release:        20.04
Codename:       focal
```

Setup

- Windows (cont’)
  - All your WSL Linux files are accessible as \\wsl$ in File Explorer
    - Exposed as network shared files
  - Your home directory is \\wsl$\home\<username>



Setup

- macOS
  - Install developer tools:

```
$ xcode-select --install
```

  - Install Homebrew (package manager)

```
$ /usr/bin/ruby -e "$(curl -fsSL https://raw.githubusercontent.com/Homebrew/install/master/install)"
```

  - Install the RISC-V compiler toolchain:

```
$ brew tap riscv/riscv
$ brew install riscv-tools
```

  - Update path; open ~/.bashrc and add the following line

```
PATH=$PATH:/usr/local/opt/riscv-gnu-toolchain/bin
```

  - Install QEMU

```
$ brew install qemu
```

Setup

- **Download xv6:**

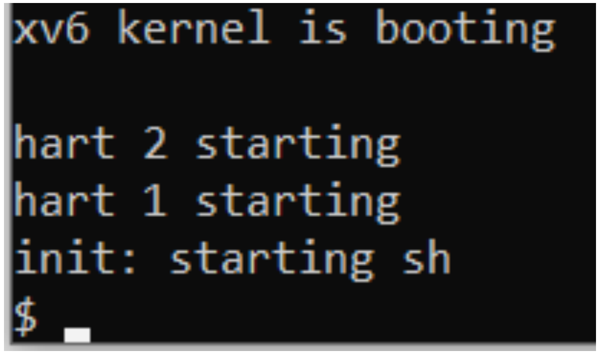
  $ git clone https://github.com/rtenlab/xv6-riscv

  $ cd xv6-riscv

- **Compile and run xv6:**

  $ make qemu

  (“make” to compile only)



(xv6 shell)

What does “hart” mean?

- In RISC-V, hart refers to a *hardware thread*
- xv6 boots on hard 0 and turns on other harts

- **Exit from QEMU**
  - Press Ctrl+a and then press c to get the QEMU console
  - Then type “quit” to exit

Create a System Call

- Goal: create a system call "sys_hello" that call a kernel function that displays: "Hello from the kernel space!"

- To do that, open the following files and add the lines with "// hello" comment:

Create a System Call (2)

- kernel/syscall.h: define new syscall number

```
#define SYS_open    15
#define SYS_write   16
#define SYS_mknod   17
#define SYS_unlink 18
#define SYS_link    19
#define SYS_mkdir   20
#define SYS_close   21
#define SYS_hello   22 // hello
```

Create a System Call (3)

- kernel/syscall.c: update system call table

```
extern uint64 sys_write(void);
extern uint64 sys_uptime(void);
extern uint64 sys_hello(void); // hello: declaration

static uint64 (*syscalls[])(void) = {
[SYS_fork]    sys_fork,
[SYS_exit]    sys_exit,
[SYS_wait]    sys_wait,
[SYS_pipe]    sys_pipe,
[SYS_read]    sys_read,
[SYS_kill]    sys_kill,
[SYS_exec]    sys_exec,
[SYS_fstat]   sys_fstat,
[SYS_chdir]   sys_chdir,
[SYS_dup]     sys_dup,
[SYS_getpid]  sys_getpid,
[SYS_sbrk]    sys_sbrk,
[SYS_sleep]   sys_sleep,
[SYS_uptime]  sys_uptime,
[SYS_open]    sys_open,
[SYS_write]   sys_write,
[SYS_mknod]   sys_mknod,
[SYS_unlink]  sys_unlink,
[SYS_link]    sys_link,
[SYS_mkdir]   sys_mkdir,
[SYS_close]   sys_close,
[SYS_hello]   sys_hello, // hello: syscall entry
};
```

UCR

Create a System Call (4)

- kernel/sysproc.c: define syscall function

```
uint64 sys_hello(void) // hello syscall definition
{
  int n;
  argint(0, &n);
  print_hello(n);
  return 0;
}
```

- kernel/proc.c: new kernel function

```
// hello: printing hello msg
void print_hello(int n)
{
  printf("Hello from the kernel space %d\n", n);
}
```

Create a System Call (5)

- kernel/defs.h

```
// proc.c
int             cpuid(void);
void            exit(int);
int             fork(void);
int             growproc(int);
void            proc_mapstacks(pagetable_t);
pagetable_t     proc_pagetable(struct proc *);
void            proc_freepagetable(pagetable_t, uint64);
int             kill(int);
int             killed(struct proc*);
void            setkilled(struct proc*);
struct cpu*     mycpu(void);
struct cpu*     getmycpu(void);
struct proc*    myproc();
void            procinit(void);
void            scheduler(void) __attribute__((noreturn));
void            sched(void);
void            sleep(void*, struct spinlock*);
void            userinit(void);
int             wait(uint64);
void            wakeup(void*);
void            yield(void);
int             either_copyout(int user_dst, uint64 dst, void *src, uint64 len);
int             either_copyin(void *dst, int user_src, uint64 src, uint64 len);
void            procdump(void);
void            print_hello(int); // hello
```

Create a System Call (6)

- Update user-space syscall interface
- user/usys.pl

```
entry("sbrk");
entry("sleep");
entry("uptime");
# hello syscall for user
entry("hello");

```

- user/user.h

```
// system calls
int fork(void);
int exit(int) __attribute__((noreturn));
int wait(int*);
int pipe(int*);
int write(int, const void*, int);
int read(int, void*, int);
int close(int);
int kill(int);
int exec(char*, char**);
int open(const char*, int);
int mknod(const char*, short, short);
int unlink(const char*);
int fstat(int fd, struct stat*);
int link(const char*, const char*);
int mkdir(const char*);
int chdir(const char*);
int dup(int);
int getpid(void);
char* sbrk(int);
int sleep(int);
int uptime(void);
int hello(int); // hello
```

Test a System Call

1. Write a user program: Create “ test.c ” file in the user directory of “xv6-riscv” (user/test.c)

```c
#include "kernel/types.h"
#include "kernel/stat.h"
#include "user/user.h"

int main(int argc, char *argv[])
{
  int n = 0;
  if (argc >= 2) n = atoi(argv[1]);

  printf("Say hello to kernel %d\n", n);
  hello(n);
  exit(0);
}
```

Test a System Call (2)

2. Edit " Makefile " and append "$U/_test\" to UPROGS

```
UPROGS=\
	$U/_cat\
	$U/_echo\
	$U/_forktest\
	$U/_grep\
	$U/_init\
	$U/_kill\
	$U/_ln\
	$U/_ls\
	$U/_mkdir\
	$U/_rm\
	$U/_sh\
	$U/_stressfs\
	$U/_usertests\
	$U/_grind\
	$U/_wc\
	$U/_zombie\
	$U/_test\
```

Test a System Call (3)

3. Type:

   $ make qemu

4. After xv6 boots, type:

   $ test

```
xv6 kernel is booting

hart 1 starting
hart 2 starting
init: starting sh
$ test 123
Say hello to kernel 123
Hello from the kernel space 123
$ 
```

How to use GDB

- To run Qemu with GDB, you need to open another terminal at the same xv6-riscv folder.
- In the first terminal, type:

  $ make qemu-gdb

- In the second second terminal, type:

  $ gdb-multiarch -q -iex "set auto-load safe-path . "

```
~/xv6-riscv$ gdb-multiarch -q -iex "set auto-load safe-path . "
The target architecture is assumed to be riscv:rv64
warning: No executable has been specified and target does not support
determining executable automatically.  Try using the "file" command.
0x0000000000001000 in ?? ()
(gdb) continue_
```

Use “ break <address> ” to set a breakpoint
Type “ continue ” to run until breakpoint

MacOS: If gdb-multiarch doesn’t exist, try "riscv64-unknown-elf-gdb"

Change # of CPUs

- By default, xv6 is compiled for three CPUs. To change the number of CPUs, edit Makefile:

```
ifndef CPUS
# CPUS := 3
CPUS := 1
endif
```

- Unless otherwise mentioned, we will use a single-core system, so change to " CPUS := 1 "

```
xv6 kernel is booting

init: starting sh
$
```

No other harts (cores) after this change