xv6 inside-out

An OS-internals hands-on workshop

12-14 Dec 2024

what is an operating system?

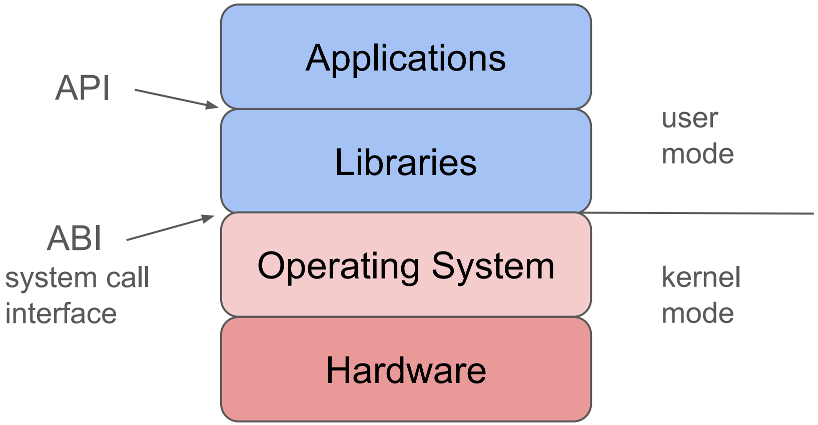
a piece of software, a program!

purpose

- enable sharing of hardware between multiple programs

- provide a set of abstractions and services

  process, files, address space, pipe, network endpoints, …



about xv6

xv6 - a simple, Unix-like teaching operating system

- based on Unix v6
- implemented in C
- two versions, one for x86 and one for RISC-V

  this hands-on – based on **x86**

- an example OS for hands-on understanding and usage

https://pdos.csail.mit.edu/6.1810/2024/xv6.html

where to run/use xv6?

OS runs on (real/physical) hardware.

xv6 runs on a (virtual) emulated machine provided via QEMU

benefits

- run xv6 in any machine (ARM, x86, RISC, etc.)
- kernel crashes can be handled gracefully.
- restarts are quicker

…

xv6 setup

1. via virtualbox
   load a pre-configure virtual machine image

2. source install on native Linux
   fetch xv6 source, fetch dependencies (qemu, gcc, …)

3. source install on Windows via WSL
   install wsl, fetch

4. source install on MAC
   install xcode, qemu, gcc, …

more details on workshop webpage

xv6 source directory

README

Makefile

source files of programs/tools

source files of the operating system

```
$ cd xv6-public/
$ ls
```

let's get started

```
$ cd xv6-public/
$ make
```

```
dd if=/dev/zero of=xv6.img count=10000
10000+0 records in
10000+0 records out
5120000 bytes (5.1 MB, 4.9 MiB) copied, 0.0914214 s, 56.0 MB/s
dd if=bootblock of=xv6.img conv=notrunc
1+0 records in
1+0 records out
512 bytes copied, 0.0411468 s, 12.4 kB/s
dd if=kernel of=xv6.img seek=1 conv=notrunc
349+1 records in
349+1 records out
179096 bytes (179 kB, 175 KiB) copied, 0.0349424 s, 5.1 MB/s
```

xv6.img is the emulated boot disk for qemu (look for QEMUOPTS in Makefile)
kernel is the compiled xv6 kernel to boot from
fs.img is the mounted file system after xv6 boot up

booting into xv6

```
$ cd xv6-public/
$ make qemu-nox
$ ls
$ cat README
$ wc README

Ctrl-a x (to quit)
```

```
qemu-system-i386 -nographic -drive file=fs.img,index=1,media=disk,format=raw
-drive file=xv6.img,index=0,media=disk,format=raw -smp 2 -m 512
xv6...
cpu1: starting 1
cpu0: starting 0
sb: size 1000 nblocks 941 ninodes 200 nlog 30 logstart 2 inodestart 32 bmap
start 58
init: starting sh
$ ls
.              1 1 512
..             1 1 512
README         2 2 2290
cat            2 3 13588
echo           2 4 12600
forktest       2 5 8032
grep           2 6 15464
init           2 7 13180
kill           2 8 12652
ln             2 9 12548
ls             2 10 14736
mkdir          2 11 12732
rm             2 12 12708
sh             2 13 23196
stressfs       2 14 13380
usertests      2 15 56312
wc             2 16 14128
zombie         2 17 12372
console        3 18 0
$
```

init and sh

After bootup, xv6 creates a **init** program which opens a **shell** in which common commands and other user programs can be run

See contents of `init.c` and `sh.c`

what is happening via the Makefile?

A makefile consists of set of rules

```
target: prerequisites
    command
    command
```

On change of any prerequisite files the commands associated with the target are executed

e.g.,

```
helloworld: helloworld.c
    gcc helloworld.c -o helloworld
```

Check prerequisites of **qemu-nox** target in the xv6 Makefile

the `qemu-nox` target

```
qemu-nox: fs.img xv6.img
    $(QEMU) -nographic $(QEMUOPTS)
```

`fs.img`: List of files to be added to the xv6 startup disk (imagefile).

```
fs.img: mkfs README $(UPROGS)
    ./mkfs fs.img README $(UPROGS)
```

`UPROGS` is variable with list all user programs in the file system after xv6 boot up
`README` is a file to be added to the file system as well

What are the prerequisites of `xv6.img?`

1. **Adding** a new file to xv6 environment

Create a new file `abc.txt` with contents
`“OS for world peace!”`

Our task is to add the file to the xv6 file system

Should be able to boot into xv6, find `abc.txt` and `cat` (display) the file

Lookup usage of UPROG and EXTRA variables in Makefile

```
init: starting sh
$ ls
.              1 1 512
..             1 1 512
README         2 2 2286
cat            2 3 15464
echo           2 4 14348
forktest       2 5 8792
grep           2 6 18308
init           2 7 14968
kill           2 8 14432
ln             2 9 14328
ls             2 10 16896
mkdir          2 11 14456
rm             2 12 14436
sh             2 13 28492
stressfs       2 14 15364
usertests      2 15 62864
wc             2 16 15892
zombie         2 17 14012
abc.txt             2 2 15
console        3 18 0
$ cat abc.txt
OS for world peace!
$
```

2. **Adding** a new userspace program to xv6

Create a new userspace program hw.c which should print “Hello world!”.

Our task is to put the file and its compiled userspace program inside the xv6 file system

Should be able to boot into xv6, find the `hw.c`, display the file and run the executable hw.

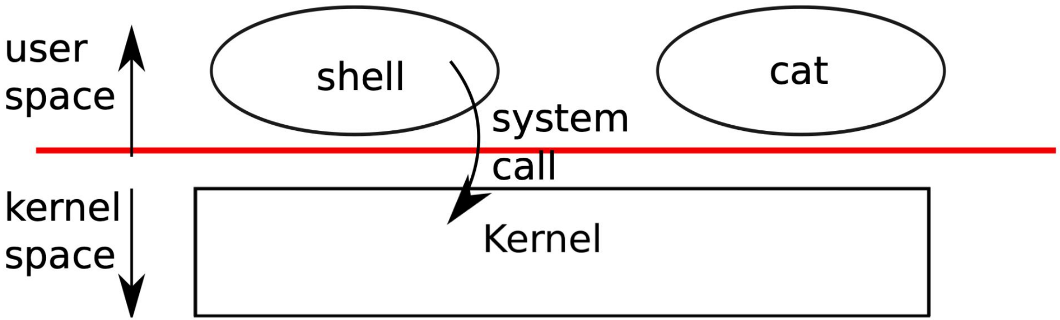
To get started look into `user.h`, `types.h, wc.c .`

Adding a new userspace program to xv6

```
$ cat hw.c
#include "types.h"
#include "user.h"

int main()
{
  printf(1, "Hello World\n");
  exit();
}
```

system calls



The system call interface allows user programs to request OS services/functions

Sample system calls listed in `user.h` for user programs …

`fork(), exec(), wait(), getpid(), kill(), pipe(), read(), write(), open(), close()` etc.

xv6 system calls

Syscall listing - can also be found in `user.h`

`fork()`, `exec()`, `wait()`, `getpid()`, `kill()`, `pipe()`, `read()`, `write()`, `open()`, `close()` etc.

3. **Using** system calls in an userspace program

Implement a program that uses a system call to start a new process (name it `myshell.c`)

Use **fork** system call to create the child process

The child process prints it PID and returns,

the parent (forking) process waits this child exits.

```
$ myshell
[P] Parent process PID: 3
[P] Waiting for child process w/ PID 4
[C] Child process PID: 4
[P] Child process with PID 4 exited
$
```

Use system calls in an userspace program (2)

Re-implement a version of the cat command (name it `mycat.c`)

Use **fork** system call to create the child process

Child process reads contents from **STDIN** writes them to **STDOUT**

Use system calls read and write **(NOT printf and scanf)**.

```
$ mycat
>>> OS is critical for world peace!
OS is critical for world peace!
>>>
```

implement your own system call!

relevant xv6 source files …

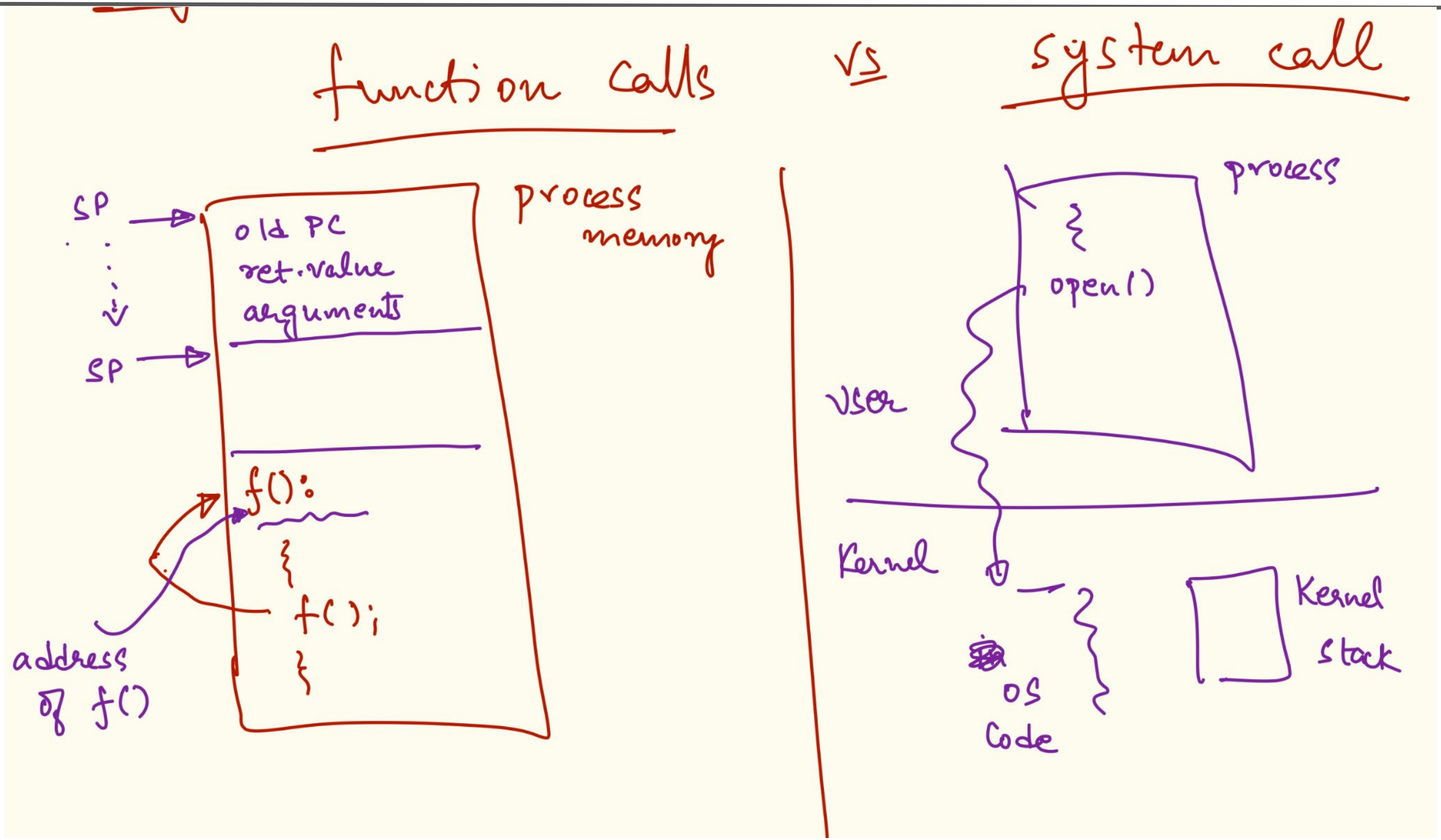
`user.h` xv6 system call declarations
`usys.S` assembly code for system call wrappers

`syscall.h` contains mapping of system call name to system call number
`syscall.c` contains helper functions to handle the system call entry, parse arguments, and pointers to the actual system call implementations
`sysproc.c` contains the implementations of process related system calls
`defs.h` a header file with function declarations in the xv6 kernel

`proc.h` contains the process abstraction related variable definitions
`proc.c` contains implementations of various process related system calls, functions and the scheduler

`sysfile.c` other system call implementation functions

function call vs. system call



xv6 system call details

need mechanism to invoke system call and switch to kernel mode

- ISA dependent
- via assembly instruction (e.g., `int 0x80`)

need information about system call (system call number, arguments)

- passed via hardware registers stored on stack

need support to save-restore process execution state

- CPU registers stored on kernel stack (the xv6 trapframe)

system call action

system calls maintain and manipulate kernel state (variables)

and perform kernel functionality (e.g., create new process, add memory etc.)

```
// Per-process state
struct proc {
  uint sz;                     // Size of process memory (bytes)
  pde_t* pgdir;                // Page table
  char *kstack;                // Bottom of kernel stack for this
process
  enum procstate state;        // Process state
  int pid;                     // Process ID
  struct proc *parent;         // Parent process
  struct trapframe *tf;        // Trap frame for current syscall
  struct context *context;     // swtch() here to run process
  void *chan;                  // If non-zero, sleeping on chan
  int killed;                  // If non-zero, have been killed
  struct file *ofile[NOFILE];  // Open files
  struct inode *cwd;           // Current directory
  char name[16];               // Process name (debugging)
};
```

What's next?

- Implementing your own system call
- Understanding memory management
- Adding your own memory management ideas
- Understanding the scheduling mechanism
- Updating the scheduler with new policies
-
- Synchronization primitives
- File system implementation

- Your imagination…..