

# Notes on reading xv6 code

**Abhijit A. M.**  
**[abhijit.comp@coep.ac.in](mailto:abhijit.comp@coep.ac.in)**

**Credits:**  
**xv6 book by Cox, Kaashoek, Morris**  
**Notes by Prof. Sorav Bansal**

**Introduction to xv6**

**Structure of xv6 code**

**Compiling and executing xv6 code**

# About xv6

- Unix Like OS
- Multi tasking, Single user
- On x86 processor
- Supports some system calls
- Small code, 7 to 10k
- Meant for learning OS concepts
- **No** : demand paging, no copy-on-write fork, no shared-memory, fixed size stack for user programs

# Use cscope and ctags with VIM

- Go to folder of xv6 code and run  
`cscope -q *. [chS]`
- Also run  
`ctags *. [chS]`
- Now download the file [http://cscope.sourceforge.net/cscope\\_maps.vim](http://cscope.sourceforge.net/cscope_maps.vim) as `.cscope_maps.vim` in your `~` folder
- And add line "`source ~/.cscope_maps.vim`" in your `~/.vimrc` file
- Read this tutorial [http://cscope.sourceforge.net/cscope\\_vim\\_tutorial.html](http://cscope.sourceforge.net/cscope_vim_tutorial.html)

# Use call graphs (using doxygen)

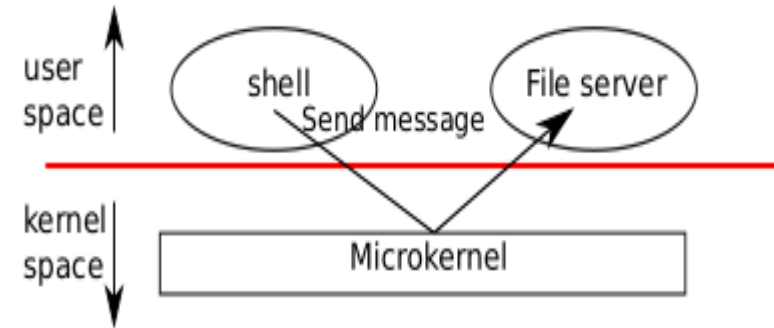
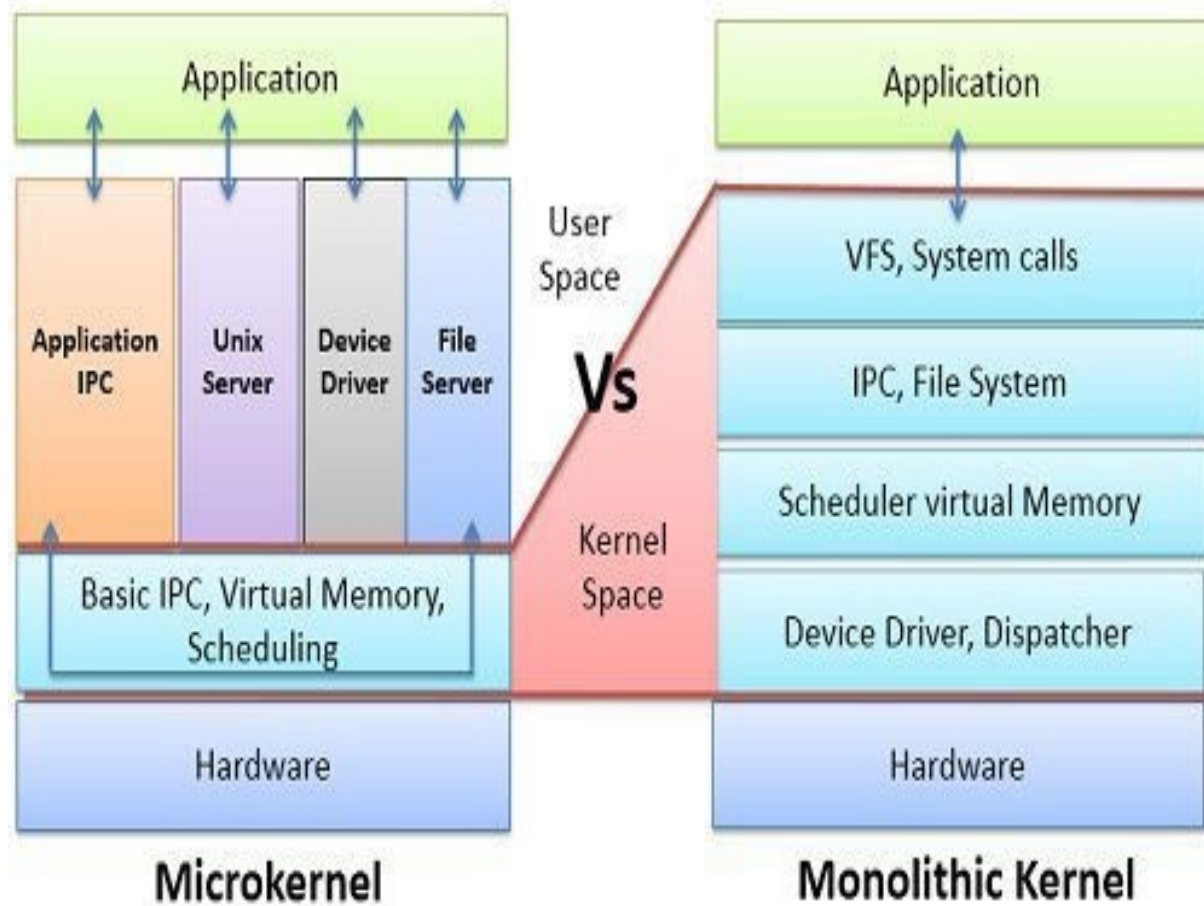
- **Doxygen – a documentation generator.**
- **Can also be used to generate “call graphs” of functions**
- **Download xv6**
- **Install doxygen on your Ubuntu machine.**
- **cd to xv6 folder**
- **Run “doxygen -g doxyconfig”**
  - **This creates the file “doxyconfig”**

# Use call graphs (using doxygen)

- Create a folder “doxygen”
- Open “doxyconfig” file and make these changes.

```
PROJECT_NAME           = "XV6"
OUTPUT_DIRECTORY       = ./doxygen
CREATE_SUBDIRS         = YES
EXTRACT_ALL            = YES
EXCLUDE                = usertests.c cat.c yes.c echo.c forktest.c
                        grep.c init.c kill.c ln.c ls.c mkdir.c rm.c sh.c stressfs.c wc.c
                        zombie.c
CALL_GRAPH             = YES
CALLER_GRAPH           = YES
```
- Now run “doxygen doxyconfig”
- Go to “doxygen”/html and open “firefox index.html” --> See call graphs in files  
-> any file

# Xv6 follows monolithic kernel approach



# qemu

- A virtual machine manager, like Virtualbox
- Qemu provides us
  - BIOS
  - Virtual CPU, RAM, Disk controller, Keyboard controller
  - IOAPIC, LAPIC
- Qemu runs xv6 using this command

```
qemu -serial mon:stdio -drive
file=fs.img,index=1,media=disk,format=raw -drive
file=xv6.img,index=0,media=disk,format=raw -smp 2 -m 512
```

  - Invoked when you run “make qemu”



# qemu

- **Understanding qemu command**
  - **-serial mon:stdio**
    - the window of xv6 is also multiplexed in your normal terminal.
    - Run “make qemu”, then Press “Ctrl-a” and “c” in terminal and you get qemu prompt
  - **-drive file=fs.img,index=1,media=disk,format=raw**
    - Specify the hard disk in “fs.img”, accessible at first slot in IDE(or SATA, etc), as a “disk” , with “raw” format
  - **-smp 2**
    - Two cores in SMP mode to be simulated
  - **-m 512**
    - Use 512 MB ram

# About files in XV6 code

- `cat.c echo.c forktest.c grep.c init.c kill.c ln.c ls.c mkdir.c rm.c sh.c stressfs.c usertests.c wc.c yes.c zombie.c`
  - User programs for testing xv6
- `Makefile`
  - To compile the code
- `dot-bochsrc`
  - For running with emulator bochs

# About files in XV6 code

- **bootasm.S   entryother.S   entry.S  
initcode.S   swtch.S   trapasm.S  
usys.S**
  - Kernel code written in Assembly. Total 373 lines
- **kernel.ld**
  - Instructions to Linker, for linking the kernel properly
- **README   Notes   LICENSE**
  - Misc files

# Using Makefile

- **make qemu**
  - Compile code and run using “qemu” emulator
- **make xv6.pdf**
  - Generate a PDF of xv6 code
- **make mkfs**
  - Create the mkfs program
- **make clean**
  - Remove all intermediary and final build files

# Files generated by Makefile

- **.o files**
  - Compiled from each .c file
  - No need of separate instruction in Makefile to create .o files
  - `_%: %.o $(ULIB)` line is sufficient to build each .o for a `_xyz` file
-

# Files generated by Makefile

- **asm files**

- Each of them has an equivalent object code file or C file. For example

```
bootblock: bootasm.S bootmain.c
```

```
    $(CC) $(CFLAGS) -fno-pic -O -nostdinc -I. -c  
bootmain.c
```

```
    $(CC) $(CFLAGS) -fno-pic -nostdinc -I. -c bootasm.S
```

```
    $(LD) $(LDFLAGS) -N -e start -Ttext 0x7C00 -o  
bootblock.o bootasm.o bootmain.o
```

```
    $(OBJDUMP) -S bootblock.o > bootblock.asm
```

```
    $(OBJCOPY) -S -O binary -j .text bootblock.o  
bootblock
```

```
    ./sign.pl bootblock
```

# Files generated by Makefile

- **\_ln, \_ls, etc**
  - Executable user programs
  - Compilation process is explained after few slides

# Files generated by Makefile

- **xv6.img**
    - Image of xv6 created
- xv6.img: bootblock kernel**
- ```
dd if=/dev/zero of=xv6.img  
count=10000  
dd if=bootblock of=xv6.img  
conv=notrunc  
dd if=kernel of=xv6.img seek=1  
conv=notrunc
```



# Files generated by Makefile

- **bootblock**

```
bootblock: bootasm.S bootmain.c
```

```
    $(CC) $(CFLAGS) -fno-pic -O -nostdinc -I. -c  
bootmain.c
```

```
    $(CC) $(CFLAGS) -fno-pic -nostdinc -I. -c bootasm.S
```

```
    $(LD) $(LDFLAGS) -N -e start -Ttext 0x7C00 -o  
bootblock.o bootasm.o bootmain.o
```

```
    $(OBJDUMP) -S bootblock.o > bootblock.asm
```

```
    $(OBJCOPY) -S -O binary -j .text bootblock.o  
bootblock
```

```
    ./sign.pl bootblock
```

# Files generated by Makefile

## kernel

```
kernel: $(OBS) entry.o entryother initcode kernel.ld
        $(LD) $(LDFLAGS) -T kernel.ld -o
kernel  entry.o $(OBS) -b binary
initcode entryother
        $(OBJDUMP) -S kernel > kernel.asm
        $(OBJDUMP) -t kernel | sed
'1,/SYMBOL TABLE/d; s/ .* / /; /^$$/d' >
kernel.sym
```

# Files generated by Makefile

- **fs.img**

- A disk image containing user programs and README

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

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

- **.sym files**

- Symbol tables of different programs

- E.g. for file “kernel”

```
$(OBJDUMP) -t kernel | sed '1,/SYMBOL TABLE/d; s/  
.* / /; /^$$/d' > kernel.sym
```

# Size of xv6 C code

- **wc \*[ch] | sort -n**
  - **10595 34249 278455 total**
  - **Out of which**
    - **738 4271 33514 dot-bochsrc**
- **wc cat.c echo.c forktest.c grep.c init.c kill.c  
ln.c ls.c mkdir.c rm.c sh.c stressfs.c  
usertests.c wc.c yes.c zombie.c**
  - **2849 6864 51993 total**
- **So total code is  $10595 - 2849 - 738 = 7008$  lines**

# List of commands to try (in given order)

**usertests # Runs lot of tests and takes upto 10 minutes to run**

**stressfs # opens , reads and writes to files in parallel**

**ls # out put is file type, inode number, type**

**cat README**

**ls;ls**

**cat README | grep BUILD**

**echo hi there**

**echo hi there | grep hi**

**echo "hi there**

# List of commands to try (in this order)

echo README | grep Wa

echo README | grep Wa |  
grep ty # does not work

cat README | grep Wa |  
grep bl # works

ls > out # takes time!

mkdir test

cd test

ls # fails

ls ../ # works from inside  
test

cd # fails

cd / # works

wc README

rm out

ls . test # listing both  
directories

ln cat xyz; ls

rm xyz; ls

# User Libraries: Used to link user land programs

- **Ulib.c**
  - Strcpy, strcmp, strlen, memset, strchr, stat, atoi, memmove
  - Stat uses open()
- **Usys.S -> compiles into usys.o**
  - Assembly code file. Basically converts all calls like open() (e.g. used in ulib.c) into assembly code using “int” instruction.

Run following command see the last 4 lines in the output

**objdump -d usys.o**

**00000048 <open>:**

|            |                       |            |                   |
|------------|-----------------------|------------|-------------------|
| <b>48:</b> | <b>b8 0f 00 00 00</b> | <b>mov</b> | <b>\$0xf,%eax</b> |
| <b>4d:</b> | <b>cd 40</b>          | <b>int</b> | <b>\$0x40</b>     |
| <b>4f:</b> | <b>c3</b>             | <b>ret</b> |                   |

# User Libraries: Used to link user land programs

- **printf.c**
  - Code for printf()!
  - Interesting to read this code.
    - Uses variable number of arguments. Normal technique in C is to use va\_args library, but here it uses pointer arithmetic.
  - Written using two more functions: printint() and putc() - both call write()
    - Where is code for write()?



# User Libraries: Used to link user land programs

- **umalloc.c**
  - This is an implementation of malloc() and free()
  - Almost same as the one done in “The C Programming Language” by Kernighan and Ritchie
  - Uses sbrk() to get more memory from xv6 kernel

# Understanding the build process in more details

- **Run**

`make qemu | tee make-output.txt`

- **You will get all compilation commands in  
make-output.txt**

# Compiling user land programs

Normally when you compile a program on Linux

You compile it for the same 'target' machine ( = CPU + OS)

The compiler itself runs on the same OS

To compile a user land program for xv6, we don't have a compiler on xv6,

So we compile the programs (using make, cc) on Linux , for xv6

Obviously they can't link with the standard libraries on Linux

# Compiling user land programs

```
ULIB = ulib.o usys.o printf.o umalloc.o
```

```
_%: %.o $(ULIB)
```

```
$(LD) $(LDFLAGS) -N -e main -Ttext 0 -o $@ $^
```

```
$(OBJDUMP) -S $@ > $*.asm
```

```
$(OBJDUMP) -t $@ | sed '1,/SYMBOL TABLE/d;  
s/ .* / /; /^$$/d' > $*.sym
```

**`$@` is the name of the file being generated**

**`$^` is dependencies . i.e. `$(ULIB)` and `%.o` in this case**

# Compiling user land programs

```
gcc -fno-pic -static -fno-builtin -fno-strict-aliasing  
-O2 -Wall -MD -ggdb -m32 -Werror -fno-omit-frame-  
pointer -fno-stack-protector -fno-pie -no-pie -c -o  
cat.o cat.c
```

```
ld -m elf_i386 -N -e main -Ttext 0 -o _cat cat.o  
ulib.o usys.o printf.o umalloc.o
```

```
objdump -S _cat > cat.asm
```

```
objdump -t _cat | sed '1,/SYMBOL TABLE/d;  
s/ .* / /; /^$/d' > cat.sym
```

# Compiling user land programs

**Mkfs is compiled like a Linux program !**

**gcc -Werror -Wall -o mkfs mkfs.c**

# How to read kernel code ?

- **Understand the data structures**
  - Know each global variable, typedefs, lists, arrays, etc.
  - Know the purpose of each of them
- **While reading a code path, e.g. exec()**
  - Try to 'locate' the key line of code that does major work
  - Initially (but not forever) ignore the 'error checking' code
- **Keep summarising what you have read**
  - Remembering is important !
- **To understand kernel code, you should be good with concepts in OS , C, assembly, hardware**

# Pre-requisites for reading the code

- **Understanding of core concepts of operating systems**
  - Memory Management, processes, fork-exec, file systems, synchronization, x86 architecture, calling convention , computer organization
- **2 approaches:**
  - 1) Read OS basics first, and then start reading xv6 code
    - **Good approach, but takes more time !**
  - 2) Read some basics, read xv6, repeat
    - **Gives a headstart, but you will always have gaps in your understanding of the code, until you are done with everything**
    - **We normally follow this approach**
- **Good knowledge of C, pointers, **function pointers** particularly**
  - Data structures: doubly linked lists, queues, structures and pointers