QEMU installation

$sudo apt-get install qemu

Remove object files and executables

Note: the names of directories are for public xv6 and differs in Mamans

$cd xv6-public
$make clean

```
rm -f *.tex *.dvi *.idx *.aux *.log *.ind *.ilg\
.*o *.d *.asm *.sym vectors.S bootblock entryother\
initcode initcode.out kernel xv6.img fs.img kernelmemfs\
xv6memfs.img mkfs .gdbinit\
_cat _echo _forktest _grep _init _kill _ln _ls _mkdir _rm _sh _stressfs _usertests _wc _zombie _uthread
```

Build xv6

$cd xv6-public
$make

```
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 -fno-pic -O -nostdinc -I. -c bootmain.c
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 -fno-pic -nostdinc -I. -c bootasm.S
ld -m    elf_i386 -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
boot block is 448 bytes (max 510)
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 bio.o bio.c
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 console.o console.c
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 exec.o exec.c
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 file.o file.c
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 fs.o fs.c
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 ide.o ide.c
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 ioapic.o ioapic.c
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 kalloc.o kalloc.c
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 kbd.o kbd.c
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 lapic.o lapic.c
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 log.o log.c
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 main.o main.c
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 mp.o mp.c
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 picirq.o picirq.c
gcc -fno-pic -static -fno-builtin -fno-strict-aliasing -O2 -Wall -MD -ggdb -m32 -Werror -fno-omit-frame-pointer -fno-stack-
```

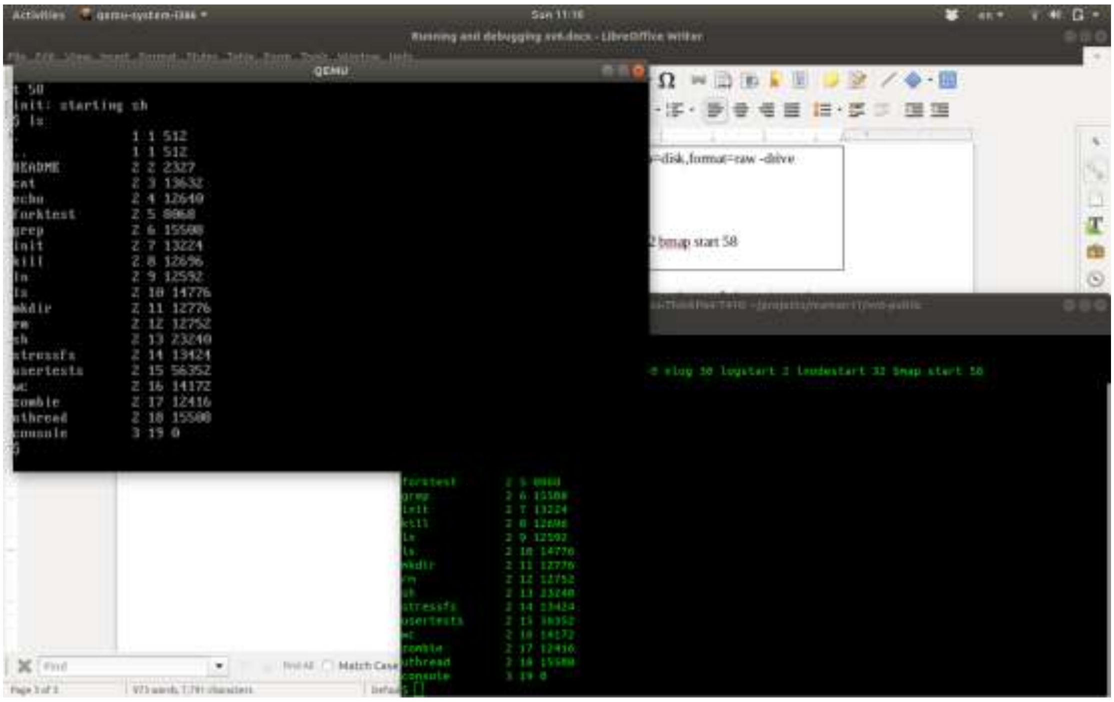
```
protector -fno-pie -no-pie   -c -o pipe.o pipe.c
gcc -fno-pic -static -fno-builtin -fno-strict-aliasing -O2 -Wall -MD -ggdb -m32 -Werror -fno-omit-frame-pointer -fno-stack-protector -fno-pie -no-pie   -c -o proc.o proc.c
gcc -fno-pic -static -fno-builtin -fno-strict-aliasing -O2 -Wall -MD -ggdb -m32 -Werror -fno-omit-frame-pointer -fno-stack-protector -fno-pie -no-pie   -c -o sleeplock.o sleeplock.c
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 spinlock.o spinlock.c
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 string.o string.c
gcc -m32 -gdwarf-2 -Wa,-divide   -c -o swtch.o swtch.S
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 syscall.o syscall.c
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 sysfile.o sysfile.c
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 sysproc.o sysproc.c
gcc -m32 -gdwarf-2 -Wa,-divide   -c -o trapasm.o trapasm.S
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 trap.o trap.c
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 uart.o uart.c
/.vectors.pl > vectors.S
gcc -m32 -gdwarf-2 -Wa,-divide   -c -o vectors.o vectors.S
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 vm.o vm.c
gcc -m32 -gdwarf-2 -Wa,-divide   -c -o entry.o entry.S
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 -fno-pic -nostdinc -I. -c entryother.S
ld -m    elf_i386 -N -e start -Ttext 0x7000 -o bootblockother.o entryother.o
objcopy -S -O binary -j .text bootblockother.o entryother
objdump -S bootblockother.o > entryother.asm
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 -nostdinc -I. -c initcode.S
ld -m    elf_i386 -N -e start -Ttext 0 -o initcode.out initcode.o
objcopy -S -O binary initcode.out initcode
objdump -S initcode.o > initcode.asm
ld -m    elf_i386 -T kernel.ld -o kernel entry.o bio.o console.o exec.o file.o fs.o ide.o ioapic.o kalloc.o kbd.o lapic.o log.o main.o mp.o picirq.o pipe.o proc.o sleeplock.o spinlock.o string.o swtch.o syscall.o sysfile.o sysproc.o trapasm.o trap.o uart.o vectors.o vm.o  -b binary initcode entryother
objdump -S kernel > kernel.asm
objdump -t kernel | sed '1,/SYMBOL TABLE/d; s/ .* / /; /^$/d'> kernel.sym
dd if=/dev/zero of=xv6.img count=10000
10000+0records in
10000+0records out
5120000bytes (5.1 MB, 4.9 MiB) copied, 0.0598836 s, 85.5 MB/s
dd if=bootblock of=xv6.img conv=notrunc
1+0records in
1+0records out
512bytes copied, 0.00028549 s, 1.8 MB/s
dd if=kernel of=xv6.img seek=1 conv=notrunc
349+1records in
349+1records out
178916bytes (179 kB, 175 KiB) copied, 0.0017677 s, 101 MB/s
```

Run xv6 under QEMU

$cd xv6-public
$make qemu

```
qemu-system-i386 -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
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
```

A separate window should appear containing the display of the virtual machine. After a few seconds, QEMU's virtual BIOS will load xv6's boot loader from a virtual hard drive image contained in the file xv6.img, and the boot loader will in turn load and run the xv6 kernel. After everything is loaded, you should get a '$' prompt in the xv6 display window and be able to enter commands into the rudimentary but functional xv6 shell. For example, try ‘ls’.



```
t 58
init: starting sh
$ ls
.              1 1 512
..             1 1 512
README         2 2 2327
cat            2 3 13632
echo           2 4 12640
forktest       2 5 8068
grep           2 6 15508
init           2 7 13224
kill           2 8 12696
ln             2 9 12592
ls             2 10 14776
mkdir          2 11 12776
rm             2 12 12752
sh             2 13 23240
stressfs       2 14 13424
usertests      2 15 56352
wc             2 16 14172
zombie         2 17 12416
uthread        2 18 15508
console        3 19 0
$
```

Remote Debugging xv6 under QEMU

The basic idea is that the main debugger (GDB in this case) runs separately from the program being debugged (the xv6 kernel atop QEMU) - they could be on completely separate machines, in fact. The debugger and the target environment communicate over some simple communication medium, such as a network socket or a serial cable, and a small *remote debugging stub* handles the "immediate supervision" of the program being debugged in the target environment

To run xv6 under QEMU and enable remote debugging, type

$cd xv6-public
$make qemu-gdb

```
sed "s/localhost:1234/localhost:26000/"< .gdbinit.tmpl > .gdbinit
 ***Now run 'gdb.'
qemu-system-i386 -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  -S -gdb tcp::26000
```

You will notice that while a window appears representing the virtual machine's display, nothing appears on that display: that is because QEMU initialized the virtual machine but stopped it before executing the first instruction, and is now waiting for an instance of GDB to connect to its remote debugging stub and supervise the virtual machine's execution. In particular, QEMU is listening for connections on a TCP network socket, at port 26000 in this example, because of the '-p 26000' in the qemu command line above (see the GDBPORT variable in xv6's Makefile).

Add to *~/.gdbinit* the line "*set auto-load safe-path /*" and start the debugger and connect it to QEMU's waiting remote debugging stub. Open a new, separate terminal window, change to the same xv6 directory, and type *gdb kernel* (to debug xv6 kernel) or *gdb xv6_executable_file* (to debug usermode utils like *ls*, *cat*, *ps* etc.)

```
gdb kernel $
GNU gdb (Ubuntu 8.1-0ubuntu3) 8.1.0.20180409-git
Copyright (C) 2018 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html<
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law.  Type "show copying"
and "show warranty" for details.
This GDB was configured as "x86_64-linux-gnu."
Type "show configuration" for configuration details.
For bug reporting instructions, please see:
>http://www.gnu.org/software/gdb/bugs .</
Find the GDB manual and other documentation resources online at:
>http://www.gnu.org/software/gdb/documentation.</
For help, type "help."
Type "apropos word" to search for commands related to "word."
 +target remote localhost:26000
warning: No executable has been specified and target does not support
determining executable automatically.  Try using the "file" command.
The target architecture is assumed to be i8086
]f000:fff0    [0xffff0:   ljmp    $0x3630,$0xf000e05b
0x0000fff0 in() ??
 +symbol-file kernel
warning: A handler for the OS ABI "GNU/Linux" is not built into this configuration
of GDB.  Attempting to continue with the default i8086 settings.
```

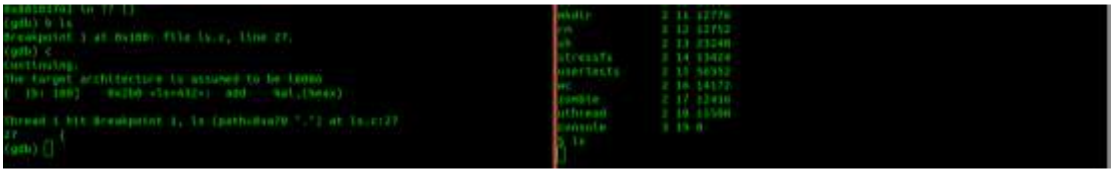
In the same manner, if userspace executable was specified as a *gdb* parameter, the appropriate symbol table were loaded. In the example above, the *kernel* symbol table was loaded. Let's load the *ls* userspace utility symbol table by running *symbol-file _ls* from the *gdb* command line

```
Reading symbols from _ls...done
```

Note, all xv6 urespace utilities have _ prepended to their names on the host operating system

The GDB command *target remote localhost:26000* connects to a remote debugging stub, given the waiting stub's TCP host name and port number. As mentioned earlier, QEMU's remote debugging stub stops the virtual machine before it executes the first instruction

Run the *b ls* and *c* commands from the *gdb* command line to set a breakpoint at the beginning of the *ls* function from the *ls.c* file and then continue the virtual machine's execution until it hits that breakpoint and run *ls* from the *xv6* command line. Now, you should see GDB appear to "hang" on *ls* function.

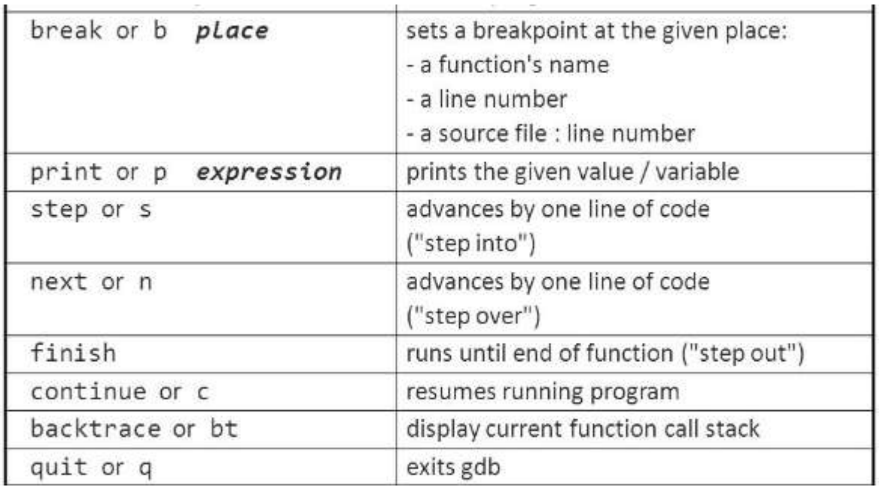


```
[illegible] in ?? ()
(gdb) b ls
Breakpoint 1 at 0x100: file ls.c, line 27.
(gdb) c
Continuing.
The target architecture is assumed to be i8086
[   1b: 100]    0x2b0 <ls+432>:  add    %al,(%eax)

Thread 1 hit Breakpoint 1, ls (path=0xa70 ".") at ls.c:27
27      {
(gdb) 
```

```
mkdir          2 11 12776
rm             2 12 12752
sh             2 13 23248
stressfs       2 14 13424
usertests      2 15 56352
wc             2 16 14172
zombie         2 17 12416
uthread        2 18 15508
console        3 19 0
$ ls
```

GNU GDB Debugger Command Cheat Sheet



break or b ***place***	sets a breakpoint at the given place: - a function's name - a line number - a source file : line number
print or p ***expression***	prints the given value / variable
step or s	advances by one line of code ("step into")
next or n	advances by one line of code ("step over")
finish	runs until end of function ("step out")
continue or c	resumes running program
backtrace or bt	display current function call stack
quit or q	exits gdb

References

https://www.youtube.com/watch?v=GQrNks_T0N8

http://zoo.cs.yale.edu/classes/cs422/2010/lec/l2-hw