Homework: intro to xv6

This lecture is the introduction to xv6, our re-implementation of Unix v6. Read the source code in the assigned files. You won't have to understand the details yet; we will focus on how the first user-level process comes into existence after the computer is turned on.

Hand-In Procedure for Sep 18

You are to turn in this homework during lecture. Please write up your answers to the exercises below and hand them in to a 6.828 staff member at the beginning of lecture.

Assignment:
Download xv6 and expand the tar ball:

Download xv6_rev0.zip and xv6.pdf from the assignments section page.
Extract the folder xv6 from xv6_rev0.zip

Build xv6:

```
$ cd xv6
$ make
cc -o mkfs mkfs.c
gcc -fno-builtin -O2 -Wall -MD   -c -o usertests.o usertests.c
gcc -fno-builtin -O2 -Wall -MD   -c -o ulib.o ulib.c
...
$
```

Find the address of the `main0` function by looking in `kernel.asm`:

```
sh-3.00$ grep main0 kernel.asm
001015b0 <main0>:
  1015f0:      7e ee                  jle    1015e0 <main0+0x30>
  1016a7:      74 22                  je     1016cb <main0+0x11b>
  1016d0:      eb dc                  jmp    1016ae <main0+0xfe>
sh-3.00$
```

In this case, the address is `001015b0`. Note that this address may be different on
Run the kernel inside Bochs, setting a breakpoint at the beginning of `main0` (i.e., the address you just found).

```
$ make bochs
if [ ! -e .bochsrc ]; then ln -s dot-bochsrc .bochsrc; fi
bochs -q
======================================================================
=
                       Bochs x86 Emulator 2.2.6
              Build from CVS snapshot on January 29, 2006
======================================================================
=
00000000000i[     ] reading configuration from .bochsrc
```

```
00000000000i[     ] installing x module as the Bochs GUI
00000000000i[     ] Warning: no rc file specified.
00000000000i[     ] using log file bochsout.txt
Next at t=0
(0) [0xfffffff0] f000:fff0 (unk. ctxt): jmp far f000:e05b         ;
ea5be000f0
(1) [0xfffffff0] f000:fff0 (unk. ctxt): jmp far f000:e05b         ;
ea5be000f0
<bochs> vb 0x8:0x1015b0
<bochs> c
(0) Breakpoint 1, 0x001015b0 (0x0008:0x001015b0)
Next at t=901856
(0) [0x001015b0] 0008:0x001015b0 (unk. ctxt): push ebp
; 55
(1) [0xfffffff0] f000:fff0 (unk. ctxt): jmp far f000:e05b         ;
ea5be000f0
<bochs>
```

Look at the registers and the stack contents:

```
<bochs> info reg
...
<bochs> print-stack
...
<bochs>
```

Which part of the stack printout is actually the stack? (Hint: not all of it.) Identify all the non-zero values on the stack.

Turn in: the output of print-stack with the valid part of the stack marked. Write a short (3-5 word) comment next to each non-zero value explaining what it is.

Now look at kernel.asm for the instructions in main0 that read:

```
  1015fc:       ba 7c a6 10 00          mov    $0x10a67c,%edx
  101601:       89 d4                   mov    %edx,%esp
  101603:       ba 9c a6 10 00          mov    $0x10a69c,%edx
  101608:       89 d5                   mov    %edx,%ebp
```

(The addresses and constants might be different for you. Look for the moves into `%esp` and `%ebp`).

Which lines in `main.c` do these instructions correspond to?

Set a breakpoint at the first of those instructions and let the program run until the breakpoint:

```
<bochs> vb 0x8:0x1015fc
<bochs> s
Next at t=901858
(0) [0x00102ea8] 0008:0x00102ea8 (unk. ctxt): jnz .+0xfffffff7
; 75f7
(1) [0xfffffff0] f000:fff0 (unk. ctxt): jmp far f000:e05b         ;
ea5be000f0
<bochs> c
(0) Breakpoint 2, 0x001015fc (0x0008:0x001015fc)
Next at t=1191513
```

```
(0) [0x001015fc] 0008:0x001015fc (unk. ctxt): mov edx, 0x0010a67c
; ba7ca61000
(1) [0xfffffff0] f000:fff0 (unk. ctxt): jmp far f000:e05b         ;
ea5be000f0
<bochs>
```
(The first `s` command is necessary to single-step past the breakpoint, or else `c` will not make any progress.)

Inspect the registers and stack again (`info reg` and `print-stack`). Then step past those four instructions (`s 4`) and inspect them again. Convince yourself that the stack has changed correctly.

Turn in: answers to the following questions. Look at the assembly for the call to `lapic_init` that immediately follows the stack switch. Where does the `bcpu` argument come from? What would have happened if the compiler had instead chosen to save `bcpu` on the stack before those four assembly instructions? Would the code still work? Why or why not?

(You can test your answer to the last two questions by running

```
$ make clean
$ make 'CFLAGS=-fno-builtin -Wall -MD'
```
to build a kernel without optimizations (the default `CFLAGS` in the Makefile also says `-O2`). Without optimization, the compiler will use the stack for every variable reference. Be sure to run `make clean` once you're finished experimenting.