| | |
|---|---|
| **Started on** | Thursday, 9 March 2023, 6:20 PM |
| **State** | Finished |
| **Completed on** | Thursday, 9 March 2023, 7:27 PM |
| **Time taken** | 1 hour 7 mins |
| **Overdue** | 11 mins 7 secs |
| **Grade** | **8.51** out of 10.00 (**85.08**%) |

Question **1**

Partially correct

Mark 0.86 out of 1.00

Select all the correct statements about code of bootmain() in xv6

```
void
bootmain(void)
{
  struct elfhdr *elf;
  struct proghdr *ph, *eph;
  void (*entry)(void);
  uchar* pa;

  elf = (struct elfhdr*)0x10000;  // scratch space

  // Read 1st page off disk
  readseg((uchar*)elf, 4096, 0);

  // Is this an ELF executable?
  if(elf->magic != ELF_MAGIC)
    return;  // let bootasm.S handle error

  // Load each program segment (ignores ph flags).
  ph = (struct proghdr*)((uchar*)elf + elf->phoff);
  eph = ph + elf->phnum;
  for(; ph < eph; ph++){
    pa = (uchar*)ph->paddr;
    readseg(pa, ph->filesz, ph->off);
    if(ph->memsz > ph->filesz)
      stosb(pa + ph->filesz, 0, ph->memsz - ph->filesz);
  }

  // Call the entry point from the ELF header.
  // Does not return!
  entry = (void(*)(void))(elf->entry);
  entry();
}
```

Also, inspect the relevant parts of the xv6 code. binary files, etc and run commands as you deem fit to answer this question.

- [ ] a. The elf->entry is set by the linker in the kernel file and it's 0x80000000
- [ ] b. The stosb() is used here, to fill in some space in memory with zeroes
- [ ] c. The elf->entry is set by the linker in the kernel file and it's 0x80000000
- [ ] d. The condition if(ph->memsz > ph->filesz) is never true.
- [ ] e. The kernel file gets loaded at the Physical address 0x10000 +0x80000000 in memory.
- [x] f. The kernel file gets loaded at the Physical address 0x10000 in memory. ✔
- [x] g. The kernel file in memory is not necessarily a continuously filled in chunk, it may have holes in it. ✔
- [x] h. The kernel ELF file contains actual physical address where particular sections of 'kernel' file should be loaded ✔
- [x] i. The elf->entry is set by the linker in the kernel file and it's 8010000c ✔
- [x] j. The kernel file has only two program headers ✔
- [x] k. The readseg finally invokes the disk I/O code using assembly instructions ✔

Your answer is partially correct.

You have correctly selected 6.
The correct answers are: The kernel file gets loaded at the Physical address 0x10000 in memory., The kernel file in memory is not necessarily a continuously filled in chunk, it may have holes in it., The elf->entry is set by the linker in the kernel file and it's 8010000c, The readseg finally invokes the disk I/O code using assembly instructions, The stosb() is used here, to fill in some space in memory with zeroes, The kernel ELF file contains actual physical address where particular sections of 'kernel' file should be loaded, The kernel file has only two program headers

Question **2**

Correct

Mark 1.00 out of 1.00

What's the trapframe in xv6?

- a. The sequence of values, including saved registers, constructed on the stack when an interrupt occurs, built by code in trapasm.S only
- b. The sequence of values, including saved registers, constructed on the stack when an interrupt occurs, built by hardware + code in trapasm.S ✔
- c. The sequence of values, including saved registers, constructed on the stack when an interrupt occurs, built by hardware only
- d. A frame of memory that contains all the trap handler code
- e. A frame of memory that contains all the trap handler code's function pointers
- f. A frame of memory that contains all the trap handler's addresses
- g. The IDT table

Your answer is correct.

The correct answer is: The sequence of values, including saved registers, constructed on the stack when an interrupt occurs, built by hardware + code in trapasm.S

Question **3**

Partially correct

Mark 0.55 out of 1.00

Consider the following command and it's output:

```
$ ls -lht xv6.img kernel
-rw-rw-r-- 1 abhijit abhijit 4.9M Feb 15 11:09 xv6.img
-rwxrwxr-x 1 abhijit abhijit 209K Feb 15 11:09 kernel*
```

Following code in bootmain()

```
  readseg((uchar*)elf, 4096, 0);
```

and following selected lines from Makefile

```
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

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

Also read the code of bootmain() in xv6 kernel.
Select the options that describe the meaning of these lines and their correlation.

- [ ] a. The bootmain() code does not read the kernel completely in memory
- [ ] b. Althought the size of the kernel file is 209 Kb, only 4Kb out of it is the actual kernel code and remaining part is all zeroes.
- [x] c. The kernel is compiled by linking multiple .o files created from .c files; and the entry.o, initcode, entryother files ✔
- [x] d. The kernel disk image is ~5MB, the kernel within it is 209 kb, but bootmain() initially reads only first 4kb, and the later part is not read as it is user programs. ✖
- [ ] e. The kernel.asm file is the final kernel file
- [ ] f. The kernel.ld file contains instructions to the linker to link the kernel properly
- [x] g. readseg() reads first 4k bytes of kernel in memory ✔
- [x] h. Althought the size of the xv6.img file is ~5MB, only some part out of it is the bootloader+kernel code and remaining part is all zeroes. ✔
- [x] i. The kernel disk image is ~5MB, the kernel within it is 209 kb, but bootmain() initially reads only first 4kb, and the later part is read using program headers in bootmain(). ✔

Your answer is partially correct.

You have correctly selected 4.
The correct answers are: The kernel disk image is ~5MB, the kernel within it is 209 kb, but bootmain() initially reads only first 4kb, and the later part is read using program headers in bootmain()., readseg() reads first 4k bytes of kernel in memory, The kernel is compiled by linking multiple .o files created from .c files; and the entry.o, initcode, entryother files, The kernel.ld file contains instructions to the linker to link the kernel properly, Althought the size of the xv6.img file is ~5MB, only some part out of it is the bootloader+kernel code and remaining part is all zeroes.

Question **4**

Correct

Mark 1.00 out of 1.00

For each line of code mentioned on the left side, select the location of sp/esp that is in use

| | |
|---|---|
| call bootmain in bootasm.S | 0x7c00 to 0 ✔ |
| cli in bootasm.S | Immaterial as the stack is not used here ✔ |
| readseg((uchar*)elf, 4096, 0); in bootmain.c | 0x7c00 to 0 ✔ |
| jmp *%eax in entry.S | The 4KB area in kernel image, loaded in memory, named as 'stack' ✔ |
| ljmp $(SEG_KCODE<<3), $start32 in bootasm.S | Immaterial as the stack is not used here ✔ |

Your answer is correct.

The correct answer is: call bootmain in bootasm.S → 0x7c00 to 0, cli in bootasm.S → Immaterial as the stack is not used here, readseg((uchar*)elf, 4096, 0); in bootmain.c → 0x7c00 to 0, jmp *%eax in entry.S → The 4KB area in kernel image, loaded in memory, named as 'stack', ljmp $(SEG_KCODE<<3), $start32 in bootasm.S → Immaterial as the stack is not used here

Question **5**

Correct

Mark 1.00 out of 1.00

Some part of the bootloader of xv6 is written in assembly while some part is written in C. Why is that so?
Select all the appropriate choices

- [ ] a. The code for reading ELF file can not be written in assembly
- [x] b. The code in assembly is required for transition to protected mode, from real mode; after that calling convention applies, hence code can be written in C ✔
- [x] c. The setting up of the most essential memory management infrastructure needs assembly code ✔
- [ ] d. The code in assembly is required for transition to protected mode, from real mode; but calling convention was applicable all the time

Your answer is correct.

The correct answers are: The code in assembly is required for transition to protected mode, from real mode; after that calling convention applies, hence code can be written in C, The setting up of the most essential memory management infrastructure needs assembly code

Question **6**

Partially correct

Mark 0.64 out of 1.00

W.r.t. Memory management in xv6,

xv6 uses physical memory upto 224 MB onlyMark statements True or False

| True | False | | |
|---|---|---|---|
| ◉☑ | ○✖ | The kernel code and data take up less than 2 MB space | ✔ |
| ◉☑ | ○✖ | PHYSTOP can be increased to some extent, simply by editing memlayout.h | ✔ |
| ◉✖ | ○☑ | The switchkvm() call in scheduler() changes CR3 to use  page directory of new process | ✖ |
| ◉✖ | ○☑ | The kernel's page table given by kpgdir variable is used as stack for scheduler's context | ✖ |
| ○☑ | ◉✖ | The free page-frame are created out of nearly 222 MB | ✖ |
| ◉☑ | ○✖ | The switchkvm() call in scheduler() changes CR3 to use page directory kpgdir | ✔ |
| ◉☑ | ○✖ | The stack allocated in entry.S is used as stack for scheduler's context for first processor | ✔ |
| ○☑ | ◉✖ | The switchkvm() call in scheduler() is invoked after control comes to it from sched(), thus demanding execution in kernel's context | ✖ |
| ○✖ | ◉☑ | The switchkvm() call in scheduler() is invoked after control comes to it from swtch() scheduler(), thus demanding execution in new process's context | ✔ |
| ◉☑ | ○✖ | xv6 uses physical memory upto 224 MB only | ✔ |
| ◉☑ | ○✖ | The process's address space gets mapped on frames, obtained from ~2MB:224MB range | ✔ |

The kernel code and data take up less than 2 MB space: True
PHYSTOP can be increased to some extent, simply by editing memlayout.h: True
The switchkvm() call in scheduler() changes CR3 to use  page directory of new process: False
The kernel's page table given by kpgdir variable is used as stack for scheduler's context: False
The free page-frame are created out of nearly 222 MB: True
The switchkvm() call in scheduler() changes CR3 to use page directory kpgdir: True
The stack allocated in entry.S is used as stack for scheduler's context for first processor: True
The switchkvm() call in scheduler() is invoked after control comes to it from sched(), thus demanding execution in kernel's context: True
The switchkvm() call in scheduler() is invoked after control comes to it from swtch() scheduler(), thus demanding execution in new process's context: False
xv6 uses physical memory upto 224 MB only: True
The process's address space gets mapped on frames, obtained from ~2MB:224MB range: True

Question **7**

Partially correct

Mark 0.75 out of 1.00

Select the correct statements about interrupt handling in xv6 code

- [ ] a. Before going to alltraps, the kernel stack contains upto 5 entries.
- [x] b. Each entry in IDT essentially gives the values of CS and EIP to be used in handling that interrupt ✔
- [x] c. xv6 uses the 64th entry in IDT for system calls ✔
- [ ] d. On any interrupt/syscall/exception the control first jumps in trapasm.S
- [ ] e. The trapframe pointer in struct proc, points to a location on process's kernel stack
- [x] f. The CS and EIP are changed only after pushing user code's SS,ESP on stack ✔
- [x] g. All the 256 entries in the IDT are filled in xv6 code ✔
- [ ] h. The function trap() is the called only in case of hardware interrupt
- [x] i. On any interrupt/syscall/exception the control first jumps in vectors.S ✔
- [ ] j. The CS and EIP are changed immediately (as the first thing) on a hardware interrupt
- [x] k. The function trap() is the called even if any of the hardware interrupt/system-call/exception occurs ✔
- [ ] l. The trapframe pointer in struct proc, points to a location on user stack
- [ ] m. xv6 uses the 0x64th entry in IDT for system calls

Your answer is partially correct.

You have correctly selected 6.

The correct answers are: All the 256 entries in the IDT are filled in xv6 code, Each entry in IDT essentially gives the values of CS and EIP to be used in handling that interrupt, xv6 uses the 64th entry in IDT for system calls, On any interrupt/syscall/exception the control first jumps in vectors.S, Before going to alltraps, the kernel stack contains upto 5 entries., The trapframe pointer in struct proc, points to a location on process's kernel stack, The function trap() is the called even if any of the hardware interrupt/system-call/exception occurs, The CS and EIP are changed only after pushing user code's SS,ESP on stack

Question **8**

Correct

Mark 1.00 out of 1.00

In bootasm.S, on the line

```
ljmp    $(SEG_KCODE<<3), $start32
```

The SEG_KCODE << 3, that is shifting of 1 by 3 bits is done because

- ○ a. The code segment is 16 bit and only lower 13 bits are used for segment number
- ◉ b. The code segment is 16 bit and only upper 13 bits are used for segment number ✔
- ○ c. While indexing the GDT using CS, the value in CS is always divided by 8
- ○ d. The value 8 is stored in code segment
- ○ e. The ljmp instruction does a divide by 8 on the first argument

Your answer is correct.

The correct answer is: The code segment is 16 bit and only upper 13 bits are used for segment number

Question **9**

Correct

Mark 1.00 out of 1.00

For each function/code-point, select the status of segmentation setup in xv6

| | |
|---|---|
| kvmalloc() in main() | gdt setup with 3 entries, at start32 symbol of bootasm.S ✔ |
| after seginit() in main() | gdt setup with 5 entries (0 to 4) on one processor ✔ |
| bootasm.S | gdt setup with 3 entries, at start32 symbol of bootasm.S ✔ |
| after startothers() in main() | gdt setup with 5 entries (0 to 4) on all processors ✔ |
| entry.S | gdt setup with 3 entries, at start32 symbol of bootasm.S ✔ |
| bootmain() | gdt setup with 3 entries, at start32 symbol of bootasm.S ✔ |

Your answer is correct.

The correct answer is: kvmalloc() in main() → gdt setup with 3 entries, at start32 symbol of bootasm.S, after seginit() in main() → gdt setup with 5 entries (0 to 4) on one processor, bootasm.S → gdt setup with 3 entries, at start32 symbol of bootasm.S, after startothers() in main() → gdt setup with 5 entries (0 to 4) on all processors, entry.S → gdt setup with 3 entries, at start32 symbol of bootasm.S, bootmain() → gdt setup with 3 entries, at start32 symbol of bootasm.S

Question **10**

Partially correct

Mark 0.71 out of 1.00

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
```

Consider above lines from the Makefile. Which of the following is INCORRECT?

- [ ] a. The xv6.img is the virtual disk that is created by combining the bootblock and the kernel file.
- [x] b. The size of the xv6.img is nearly 5 MB ✖
- [ ] c. The bootblock may be 512 bytes or less (looking at the Makefile instruction)
- [x] d. The xv6.img is of the size 10,000 blocks of 512 bytes each and occupies 10,000 blocks on the disk. ✖
- [ ] e. The bootblock is located on block-0 of the xv6.img
- [x] f. The size of the kernel file is nearly 5 MB ✔
- [x] g. The size of xv6.img is exactly = (size of bootblock) + (size of kernel) ✔
- [ ] h. The kernel is located at block-1 of the xv6.img
- [x] i. The xv6.img is of the size 10,000 blocks of 512 bytes each and occupies upto 10,000 blocks on the disk. ✔
- [ ] j. Blocks in xv6.img after kernel may be all zeroes.
- [x] k. xv6.img is the virtual processor used by the qemu emulator ✔

Your answer is partially correct.

You have selected too many options.
The correct answers are: xv6.img is the virtual processor used by the qemu emulator, The xv6.img is of the size 10,000 blocks of 512 bytes each and occupies upto 10,000 blocks on the disk., The size of the kernel file is nearly 5 MB, The size of xv6.img is exactly = (size of bootblock) + (size of kernel)

◄ Random Quiz 4 : Scheduling, signals, segmentation, paging, compilation, process-state

Jump to...

Random Quiz - 6 (xv6 file system) ►