Started on	Saturday, 12 February 2022, 10:01:13 AM
State	Finished
Completed on	Saturday, 12 February 2022, 11:52:25 AM
Time taken	1 hour 51 mins
Grade	**4.61** out of 10.00 (**46**%)

Question **1**
Complete
Mark 0.08 out of 0.50

Order the sequence of events, in scheduling process P1 after process P0

timer interrupt occurs	1
context of P0 is saved in P0's PCB	3
context of P1 is loaded from P1's PCB	6
Process P1 is running	5
Process P0 is running	2
Control is passed to P1	4

The correct answer is: timer interrupt occurs → 2, context of P0 is saved in P0's PCB → 3, context of P1 is loaded from P1's PCB → 4, Process P1 is running → 6, Process P0 is running → 1, Control is passed to P1 → 5

Question **2**

Complete

Mark 0.50 out of 0.50

Suppose a program does a scanf() call.

Essentially the scanf does a read() system call.

This call will obviously "block" waiting for the user input.

In terms of OS data structures and execution of code, what does it mean?

Select one:

- a. read() will return and process will be taken to a wait queue
- b. read() returns and process calls scheduler()
- c. OS code for read() will move the PCB of this process to a wait queue and return from the system call
- d. OS code for read() will call scheduler
- e. OS code for read() will move PCB of current process to a wait queue and call scheduler

The correct answer is: OS code for read() will move PCB of current process to a wait queue and call scheduler

Question **3**

Complete

Mark 0.50 out of 0.50

What's the trapframe in xv6?

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

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 **4**

Complete

Mark 0.45 out of 0.50

Select Yes if the mentioned element should be a part of PCB

Select No otherwise.

Yes	No	
○	◉	Function pointers to all system calls
◉	○	Process context
○	◉	EIP at the time of context switch
◉	○	Pointer to the parent process
◉	○	Memory management information about that process
◉	○	List of opened files
○	◉	Pointer to IDT
◉	○	Process state
◉	○	PID
○	◉	PID of Init

Function pointers to all system calls: No
Process context: Yes
EIP at the time of context switch: Yes
Pointer to the parent process: Yes
Memory management information about that process: Yes
List of opened files: Yes
Pointer to IDT: No
Process state: Yes
PID: Yes
PID of Init: No

Question **5**

Complete

Mark 0.60 out of 1.00

Mark the statements, w.r.t. the scheduler of xv6 as True or False

True	False	
◉	○	`swtch` is a function that saves old context, loads new context, and returns to last EIP in the new context
◉	○	`swtch` is a function that does not return to the caller
◉	○	`sched()` and `scheduler()` are co-routines
◉	○	The variable c->scheduler on first processor uses the stack allocated entry.S
○	◉	The function `scheduler()` executes using the kernel-only stack
◉	○	`sched()` *calls* `scheduler()` and `scheduler()` *calls* `sched()`
○	◉	the control returns to `switchkvm();` after `swtch(&(c->scheduler), p->context);` in `scheduler()`
◉	○	When a process is scheduled for execution, it resumes execution in `sched()` after the call to `swtch()`
○	◉	The work of selecting and scheduling a process is done only in `scheduler()` and not in `sched()`
◉	○	the control returns to `mycpu()->intena = intena; ();` after `swtch(&p->context, mycpu()->scheduler);` in `sched()`

`swtch` is a function that saves old context, loads new context, and returns to last EIP in the new context: True
`swtch` is a function that does not return to the caller: True
`sched()` and `scheduler()` are co-routines: True
The variable c->scheduler on first processor uses the stack allocated entry.S: True
The function `scheduler()` executes using the kernel-only stack: True
`sched()` *calls* `scheduler()` and `scheduler()` *calls* `sched()`: False
the control returns to `switchkvm();` after `swtch(&(c->scheduler), p->context);` in `scheduler()`: False
When a process is scheduled for execution, it resumes execution in `sched()` after the call to `swtch()`
: True
The work of selecting and scheduling a process is done only in `scheduler()` and not in `sched()`: True

the control returns to `mycpu()->intena = intena;` `();` after `swtch(&p->context, mycpu()->scheduler);` in `sched()`: False

Question **6**

Complete

Mark 0.20 out of 0.50

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

readseg((uchar*)elf, 4096, 0); in bootmain.c	0x7c00 to 0
ljmp $(SEG_KCODE<<3), $start32 in bootasm.S	Immaterial as the stack is not used here
cli in bootasm.S	The 4KB area in kernel image, loaded in memory, named as 'stack'
jmp *%eax in entry.S	0x7c00 to 0x10000
call bootmain in bootasm.S	0x10000 to 0x7c00

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

Question **7**

Complete

Mark 0.30 out of 0.50

Select all the correct statements about zombie processes

Select one or more:

- [x] a. A process becomes zombie when it finishes, and remains zombie until parent calls wait() on it
- [ ] b. If the parent of a process finishes, before the process itself, then after finishing the process is typically attached to 'init' as parent
- [ ] c. Zombie processes are harmless even if OS is up for long time
- [x] d. A zombie process occupies space in OS data structures
- [ ] e. A zombie process remains zombie forever, as there is no way to clean it up
- [x] f. init() typically keeps calling wait() for zombie processes to get cleaned up
- [ ] g. A process can become zombie if it finishes, but the parent has finished before it
- [ ] h. A process becomes zombie when it's parent finishes

The correct answers are: A process becomes zombie when it finishes, and remains zombie until parent calls wait() on it, A process can become zombie if it finishes, but the parent has finished before it, A zombie process occupies space in OS data structures, If the parent of a process finishes, before the process itself, then after finishing the process is typically attached to 'init' as parent, init() typically keeps calling wait() for zombie processes to get cleaned up

Question **8**

Complete

Mark 0.68 out of 1.00

Select the correct statements about interrupt handling in xv6 code

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

The correct answers are: All the 256 entries in the IDT are filled, 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 kernel stack, The function trap() is the called irrespective of hardware interrupt/system-call/exception, The CS and EIP are changed only after pushing user code's SS,ESP on stack

Question **9**

Complete

Mark 0.08 out of 0.50

The bootmain() function has this code

```
elf = (struct elfhdr*)0x10000;  // scratch space
readseg((uchar*)elf, 4096, 0);
```

Mark the statements as True or False with respect to this code.

In these statements 0x1000 is referred to as ADDRESS

True	False	
◉	○	This line loads the kernel code at ADDRESS
○	◉	The value ADDRESS is changed to a 0 the program could still work
○	◉	It the value of ADDRESS is changed to a higher number (upto a limit), the program could still work
◉	○	If the value of ADDRESS is changed, then the program will not work
◉	○	This line effectively loads the ELF header and the program headers at ADDRESS
○	◉	It the value of ADDRESS is changed to a lower number (upto a limit), the program could still work

This line loads the kernel code at ADDRESS: False
The value ADDRESS is changed to a 0 the program could still work: False
It the value of ADDRESS is changed to a higher number (upto a limit), the program could still work: True
If the value of ADDRESS is changed, then the program will not work: False
This line effectively loads the ELF header and the program headers at ADDRESS: False
It the value of ADDRESS is changed to a lower number (upto a limit), the program could still work: True

Question **10**

Complete

Mark 0.00 out of 0.50

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 ljmp instruction does a divide by 8 on the first argument
- b. The code segment is 16 bit and only lower 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 code segment is 16 bit and only upper 13 bits are used for segment number
- e. The value 8 is stored in code segment

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

Question **11**

Complete

Mark 0.02 out of 0.50

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.

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

- [x] i. The kernel file gets loaded at the Physical address 0x10000 +0x80000000 in memory.
- [ ] j. The elf->entry is set by the linker in the kernel file and it's 0x80000000
- [ ] k. The kernel ELF file contains actual physical address where particular sections of 'kernel' file should be loaded

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 **12**

Complete

Mark 0.00 out of 0.50

Order the events that occur on a timer interrupt:

Jump to a code pointed by IDT	1
Select another process for execution	7
Jump to scheduler code	3
Execute the code of the new process	5
Set the context of the new process	4
Save the context of the currently running process	6
Change to kernel stack of currently running process	2

The correct answer is: Jump to a code pointed by IDT → 2, Select another process for execution → 5, Jump to scheduler code → 4, Execute the code of the new process → 7, Set the context of the new process → 6, Save the context of the currently running process → 3, Change to kernel stack of currently running process → 1

Question **13**

Complete

Mark 0.50 out of 0.50

Consider the following programs

exec1.c

```
#include <unistd.h>
#include <stdio.h>
int main() {
    execl("./exec2", "./exec2", NULL);
}
```

exec2.c

```
#include <unistd.h>
#include <stdio.h>
int main() {
    execl("/bin/ls", "/bin/ls", NULL);
   printf("hello\n");
}
```

Compiled as

```
cc   exec1.c  -o exec1
cc   exec2.c  -o exec2
```

And run as

```
$ ./exec1
```

Explain the output of the above command (./exec1)

Assume that /bin/ls , i.e. the 'ls' program exists.

Select one:

- a. Execution fails as the call to execl() in exec2 fails
- b. Program prints hello
- c. Execution fails as one exec can't invoke another exec
- d. "ls" runs on current directory
- e. Execution fails as the call to execl() in exec1 fails

The correct answer is: "ls" runs on current directory

Question **14**

Complete

Mark 0.00 out of 0.50

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; but calling convention was applicable all the time
- [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; after that calling convention applies, hence code can be written in C

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 **15**

Complete

Mark 0.00 out of 1.00

Select the sequence of events that are NOT possible, assuming a non-interruptible kernel code

(Note: non-interruptible kernel code means, if the kernel code is executing, then interrupts will be disabled).

Note: A possible sequence may have some missing steps in between. An impossible sequence will will have n and n+1th steps such that n+1th step can not follow n'th step.

Select one or more:

- [x] a. P1 running
P1 makes system call
timer interrupt
Scheduler
P2 running
timer interrupt
Scheuler
P1 running
P1's system call return

- [ ] b. P1 running
P1 makes sytem call and blocks
Scheduler
P2 running
P2 makes sytem call and blocks
Scheduler
P1 running again

- [x] c. P1 running
keyboard hardware interrupt
keyboard interrupt handler running
interrupt handler returns
P1 running
P1 makes sytem call
system call returns
P1 running
timer interrupt
scheduler
P2 running

- [x] d. P1 running
P1 makes sytem call and blocks
Scheduler
P2 running
P2 makes sytem call and blocks
Scheduler
P3 running
Hardware interrupt
Interrupt unblocks P1
Interrupt returns
P3 running
Timer interrupt
Scheduler
P1 running

- [ ] e.
P1 running

P1 makes sytem call
Scheduler
P2 running
P2 makes sytem call and blocks
Scheduler
P1 running again

- [ ] f. P1 running
P1 makes system call
system call returns
P1 running
timer interrupt
Scheduler running
P2 running

The correct answers are: P1 running
P1 makes sytem call and blocks
Scheduler
P2 running
P2 makes sytem call and blocks
Scheduler
P1 running again, P1 running
P1 makes system call
timer interrupt
Scheduler
P2 running
timer interrupt
Scheuler
P1 running
P1's system call return,
P1 running
P1 makes sytem call
Scheduler
P2 running
P2 makes sytem call and blocks
Scheduler
P1 running again

Question **16**

Complete

Mark 0.70 out of 1.00

Which parts of the xv6 code in bootasm.S bootmain.c , entry.S and in the codepath related to scheduler() and trap handling() can also be written in some other way, and still ensure that xv6 works properly?

Writing code is not necessary. You only need to comment on which part of the code could be changed to something else or written in another fashion.

Maximum two points to be written.

0x8a00 -> port 0x8a00 unnecessary in bootasm.S

#hard-code PATH=/

#can use open instead of dups

◄ Extra Reading on Linkers: A writeup by Ian Taylor (keep changing url string from 38 to 39, and so on)

Jump to...

(Code) IPC - Shm, Messages ►