Started on	Saturday, 12 February 2022, 10:00:21 AM
State	Finished
Completed on	Saturday, 12 February 2022, 11:25:53 AM
Time taken	1 hour 25 mins
Grade	**4.94** out of 10.00 (**49**%)

Question **1**

Complete

Mark 0.00 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 elf->entry is set by the linker in the kernel file and it's 0x80000000
- [ ] b. The condition if(ph->memsz > ph->filesz) is never true.
- [x] c. The readseg finally invokes the disk I/O code using assembly instructions
- [x] d. The elf->entry is set by the linker in the kernel file and it's 0x80000000
- [ ] e. The kernel ELF file contains actual physical address where particular sections of 'kernel' file should be loaded
- [x] f. The kernel file gets loaded at the Physical address 0x10000 +0x80000000 in memory.
- [x] g. The kernel file in memory is not necessarily a continuously filled in chunk, it may have holes in it.
- [ ] h. The elf->entry is set by the linker in the kernel file and it's 8010000c
- [ ] i. The kernel file gets loaded at the Physical address 0x10000 in memory.
- [ ] j. The stosb() is used here, to fill in some space in memory with zeroes
- [ ] k. The kernel file has only two program headers

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**

Complete

Mark 0.50 out of 0.50

What's the trapframe in xv6?

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

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**

Complete

Mark 0.21 out of 0.50

Order the events that occur on a timer interrupt:

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

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

Question **4**

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

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

Question **5**

Complete

Mark 0.50 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 value 8 is stored in code segment
- 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 ljmp instruction does a divide by 8 on the first argument
- e. The code segment is 16 bit and only lower 13 bits are used for segment number

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

Question **6**

Complete

Mark 0.40 out of 0.50

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

Select No otherwise.

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

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

Question **7**

Complete

Mark 0.40 out of 1.00

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

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

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

Question **8**

Complete

Mark 0.00 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. "ls" runs on current directory
- b. Execution fails as the call to execl() in exec1 fails
- c. Execution fails as one exec can't invoke another exec
- d. Program prints hello
- e. Execution fails as the call to execl() in exec2 fails

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

Question **9**

Complete

Mark 0.00 out of 0.50

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

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

The correct answer is: cli
in bootasm.S → Immaterial as the stack is not used here, 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, call bootmain
in bootasm.S → 0x7c00 to 0, jmp *%eax
in entry.S → The 4KB area in kernel image, loaded in memory, named as 'stack'

Question **10**

Complete

Mark 0.63 out of 1.00

Select the correct statements about interrupt handling in xv6 code

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

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 **11**

Complete

Mark 0.50 out of 0.50

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

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

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

Question **12**

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:

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

- [ ] c. 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

- [x] d. 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

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

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 **13**

Complete

Mark 0.50 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

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

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 **14**

Complete

Mark 0.17 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	
◉	○	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
◉	○	This line loads the kernel code at ADDRESS
◉	○	It the value of ADDRESS is changed to a lower number (upto a limit), the program could still work
◉	○	This line effectively loads the ELF header and the program headers at ADDRESS
◉	○	If the value of ADDRESS is changed, then the program will not work

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

Question **15**

Complete

Mark 0.50 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.

We can use a scheduling algorithm. We can use the kernel stack in scheduler function in entrty.S and bootmain.c .

Question **16**

Complete

Mark 0.13 out of 0.50

Select all the correct statements about zombie processes

Select one or more:

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

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

◄ 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 ►

Started on	Saturday, 20 February 2021, 2:51 PM
State	Finished
Completed on	Saturday, 20 February 2021, 3:55 PM
Time taken	1 hour 3 mins
Grade	**7.30** out of 20.00 (**37**%)

Question **1**

Partially correct

Mark 0.80 out of 1.00

Select all the correct statements about the state of a process.

- [x] a. A process can self-terminate only when it's running ✔
- [x] b. Typically, it's represented as a number in the PCB ✔
- [x] c. A process that is running is not on the ready queue ✔
- [x] d. Processes in the ready queue are in the ready state ✔
- [ ] e. It is not maintained in the data structures by kernel, it is only for conceptual understanding of programmers
- [x] f. Changing from running state to waiting state results in "giving up the CPU" ✔
- [ ] g. A process in ready state is ready to receive interrupts
- [x] h. A waiting process starts running after the wait is over ✖
- [x] i. A process changes from running to ready state on a timer interrupt ✔
- [x] j. A process in ready state is ready to be scheduled ✔
- [x] k. A running process may terminate, or go to wait or become ready again ✔
- [x] l. A process waiting for I/O completion is typically woken up by the particular interrupt handler code ✔
- [ ] m. A process waiting for any condition is woken up by another process only
- [ ] n. A process changes from running to ready state on a timer interrupt or any I/O wait

Your answer is partially correct.

You have selected too many options.
The correct answers are: Typically, it's represented as a number in the PCB, A process in ready state is ready to be scheduled, Processes in the ready queue are in the ready state, A process that is running is not on the ready queue, A running process may terminate, or go to wait or become ready again, A process changes from running to ready state on a timer interrupt, Changing from running state to waiting state results in "giving up the CPU", A process can self-terminate only when it's running, A process waiting for I/O completion is typically woken up by the particular interrupt handler code

Question **2**

Incorrect

Mark 0.00 out of 1.00

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

jmp *%eax in entry.S	0x7c00 to 0x10000 ✖
ljmp $(SEG_KCODE<<3), $start32 in bootasm.S	0x10000 to 0x7c00 ✖
call bootmain in bootasm.S	0x7c00 to 0x10000 ✖
cli in bootasm.S	0x7c00 to 0 ✖
readseg((uchar*)elf, 4096, 0); in bootmain.c	The 4KB area in kernel image, loaded in memory, named as 'stack' ✖

Your answer is incorrect.

The correct answer is: 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, 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

Question **3**

Correct

Mark 0.25 out of 0.25

Order the following events in boot process (from 1 onwards)

Boot loader	2 ✔
Shell	6 ✔
BIOS	1 ✔
OS	3 ✔
Init	4 ✔
Login interface	5 ✔

Your answer is correct.

The correct answer is: Boot loader → 2, Shell → 6, BIOS → 1, OS → 3, Init → 4, Login interface → 5

Question **4**

Partially correct

Mark 0.30 out of 0.50

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. 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] b. The kernel is compiled by linking multiple .o files created from .c files; and the entry.o, initcode, entryother files ✔
- [x] c. The kernel.ld file contains instructions to the linker to link the kernel properly ✔
- [ ] d. The bootmain() code does not read the kernel completely in memory
- [ ] e. readseg() reads first 4k bytes of kernel in memory
- [ ] f. 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.
- [ ] g. The kernel.asm file is the final kernel file
- [ ] h. 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.
- [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 3.
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 **5**

Partially correct

Mark 0.50 out of 1.00

```
int f() {
    int count;
    for (count = 0; count< 2; count ++) {
        if (fork() ==0)
            printf(“Operating-System\n”);
     }
     printf(“TYCOMP\n”);
 }
```

The number of times "Operating-System" is printed, is:

Answer: 3

The correct answer is: 7.00

Question **6**

Partially correct

Mark 0.40 out of 0.50

Select Yes/True if the mentioned element must be a part of PCB

Select No/False otherwise.

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

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

Question **7**

Incorrect

Mark 0.00 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 kernel file gets loaded at the Physical address 0x10000 +0x80000000 in memory. ✗
- b. The elf->entry is set by the linker in the kernel file and it's 0x80000000 ✗
- c. The kernel ELF file contains actual physical address where particular sections of 'kernel' file should be loaded ✓
- d. The kernel file in memory is not necessarily a continuously filled in chunk, it may have holes in it. ✓
- e. The kernel file has only two program headers ✓
- f. The elf->entry is set by the linker in the kernel file and it's 0x80000000 ✗
- g. The readseg finally invokes the disk I/O code using assembly instructions ✓
- h. The elf->entry is set by the linker in the kernel file and it's 8010000c ✓
- i. The kernel file gets loaded at the Physical address 0x10000 in memory. ✓
- j. The condition if(ph->memsz > ph->filesz) is never true. ✗
- k. The stosb() is used here, to fill in some space in memory with zeroes ✓

Your answer is incorrect.

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 **8**

Partially correct

Mark 0.13 out of 0.25

Which of the following are NOT a part of job of a typical compiler?

- [x] a. Check the program for logical errors
- [ ] b. Convert high level langauge code to machine code
- [ ] c. Process the # directives in a C program
- [ ] d. Invoke the linker to link the function calls with their code, extern globals with their declaration
- [ ] e. Check the program for syntactical errors
- [ ] f. Suggest alternative pieces of code that can be written

Your answer is partially correct.

You have correctly selected 1.
The correct answers are: Check the program for logical errors, Suggest alternative pieces of code that can be written

Question **9**

Correct

Mark 0.25 out of 0.25

Rank the following storage systems from slowest (first) to fastest(last)

Cache	6
Hard Disk	3
RAM	5
Optical Disks	2
Non volatile memory	4
Registers	7
Magnetic Tapes	1

Your answer is correct.

The correct answer is: Cache → 6, Hard Disk → 3, RAM → 5, Optical Disks → 2, Non volatile memory → 4, Registers → 7, Magnetic Tapes → 1

Question **10**

Partially correct

Mark 0.21 out of 0.50

Which of the following parts of a C program do not have any corresponding machine code ?

- [ ] a. local variable declaration
- [ ] b. global variables
- [x] c. function calls ✖
- [x] d. #directives ✔
- [ ] e. expressions
- [ ] f. pointer dereference
- [x] g. typedefs ✔

Your answer is partially correct.

You have correctly selected 2.
The correct answers are: #directives, typedefs, global variables

Question **11**

Correct

Mark 0.25 out of 0.25

Match a system call with it's description

pipe	create an unnamed FIFO storage with 2 ends - one for reading and another for writing ✔
dup	create a copy of the specified file descriptor into smallest available file descriptor ✔
dup2	create a copy of the specified file descriptor into another specified file descriptor ✔
exec	execute a binary file overlaying the image of current process ✔
fork	create an identical child process ✔

Your answer is correct.

The correct answer is: pipe → create an unnamed FIFO storage with 2 ends - one for reading and another for writing, dup → create a copy of the specified file descriptor into smallest available file descriptor, dup2 → create a copy of the specified file descriptor into another specified file descriptor, exec → execute a binary file overlaying the image of current process, fork → create an identical child process

Question **12**

Correct

Mark 0.25 out of 0.25

Match the register with the segment used with it.

eip	cs ✔
edi	es ✔
esi	ds ✔
ebp	ss ✔
esp	ss ✔

Your answer is correct.

The correct answer is: eip → cs, edi → es, esi → ds, ebp → ss, esp → ss

Question **13**

Correct

Mark 0.25 out of 0.25

What's the trapframe in xv6?

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

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 **14**

Incorrect

Mark 0.00 out of 0.50

Select all the correct statements about linking and loading.

Select one or more:

- a. Continuous memory management schemes can support dynamic linking and dynamic loading. ✗
- b. Loader is last stage of the linker program ✗
- c. Continuous memory management schemes can support static linking and dynamic loading. (may be inefficiently) ✓
- d. Dynamic linking and loading is not possible without demand paging or demand segmentation. ✓
- e. Dynamic linking essentially results in relocatable code. ✓
- f. Continuous memory management schemes can support static linking and static loading. (may be inefficiently) ✓
- g. Loader is part of the operating system ✓
- h. Static linking leads to non-relocatable code ✗
- i. Dynamic linking is possible with continous memory management, but variable sized partitions only. ✗

Your answer is incorrect.

The correct answers are: Continuous memory management schemes can support static linking and static loading. (may be inefficiently), Continuous memory management schemes can support static linking and dynamic loading. (may be inefficiently), Dynamic linking essentially results in relocatable code., Loader is part of the operating system, Dynamic linking and loading is not possible without demand paging or demand segmentation.

Question **15**

Incorrect

Mark 0.00 out of 0.25

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

Your answer is incorrect.

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

Question **16**

Partially correct

Mark 0.07 out of 0.50

Order the events that occur on a timer interrupt:

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

Your answer is partially correct.

You have correctly selected 1.

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

Question **17**

Incorrect

Mark 0.00 out of 1.00

Consider the two programs given below to implement the command (ignore the fact that error checks are not done on return values of functions)

$ ls . /tmp/asdfksdf >/tmp/ddd 2>&1

Program 1

```
int main(int argc, char *argv[]) {
    int fd, n, i;
    char buf[128];

    fd = open("/tmp/ddd", O_WRONLY | O_CREAT, S_IRUSR | S_IWUSR);
    close(1);
    dup(fd);
    close(2);
    dup(fd);
    execl("/bin/ls", "/bin/ls", ".", "/tmp/asldjfaldfs", NULL);
}
```

Program 2

```
int main(int argc, char *argv[]) {
    int fd, n, i;
    char buf[128];

    close(1);
    fd = open("/tmp/ddd", O_WRONLY | O_CREAT, S_IRUSR | S_IWUSR);
    close(2);
    fd = open("/tmp/ddd", O_WRONLY | O_CREAT, S_IRUSR | S_IWUSR);
    execl("/bin/ls", "/bin/ls", ".", "/tmp/asldjfaldfs", NULL);
}
```

Select all the correct statements about the programs

Select one or more:

- a. Both programs are correct ✗
- b. Program 2 makes sure that there is one file offset used for '2' and '1' ✗
- c. Only Program 2 is correct ✗
- d. Program 2 does 1>&2 ✗
- e. Program 2 ensures 2>&1 and does not ensure > /tmp/ddd ✗
- f. Program 1 makes sure that there is one file offset used for '2' and '1' ✓
- g. Program 1 is correct for > /tmp/ddd but not for 2>&1 ✗
- h. Program 1 does 1>&2 ✗
- i. Both program 1 and 2 are incorrect ✗
- j. Program 2 is correct for > /tmp/ddd but not for 2>&1 ✗
- k. Only Program 1 is correct ✓
- l. Program 1 ensures 2>&1 and does not ensure > /tmp/ddd ✗

Your answer is incorrect.

The correct answers are: Only Program 1 is correct, Program 1 makes sure that there is one file offset used for '2' and '1'

Question **18**

Correct

Mark 0.25 out of 0.25

Select the option which best describes what the CPU does during it's powered ON lifetime

- a. Ask the user what is to be done, and execute that task
- b. Ask the OS what is to be done, and execute that task
- c. Fetch instructions specified by location given by PC, Decode and Execute it, during execution increment PC or change PC as per the instruction itself, Ask the User or the OS what is to be done next, repeat
- d. Fetch instructions specified by location given by PC, Decode and Execute it, during execution increment PC or change PC as per the instruction itself, repeat ✔
- e. Fetch instruction specified by OS, Decode and execute it, repeat
- f. Fetch instructions specified by location given by PC, Decode and Execute it, during execution increment PC or change PC as per the instruction itself, Ask OS what is to be done next, repeat

The correct answer is: Fetch instructions specified by location given by PC, Decode and Execute it, during execution increment PC or change PC as per the instruction itself, repeat

Question **19**

Partially correct

Mark 0.86 out of 1.00

Consider the following code and MAP the file to which each fd points at the end of the code.

```
int main(int argc, char *argv[]) {
   int fd1, fd2 = 1, fd3 = 1, fd4 = 1;

   fd1 = open("/tmp/1", O_WRONLY | O_CREAT, S_IRUSR|S_IWUSR);
   fd2 = open("/tmp/2", O_RDDONLY);
   fd3 = open("/tmp/3", O_WRONLY | O_CREAT, S_IRUSR|S_IWUSR);
   close(0);
   close(1);
   dup(fd2);
   dup(fd3);
   close(fd3);
   dup2(fd2, fd4);
   printf("%d %d %d %d\n", fd1, fd2, fd3, fd4);
   return 0;
}
```

1	closed	✖
fd4	/tmp/2	✔
fd2	/tmp/2	✔
fd1	/tmp/1	✔
2	stderr	✔
0	/tmp/2	✔
fd3	closed	✔

Your answer is partially correct.

You have correctly selected 6.

The correct answer is: 1 → /tmp/3, fd4 → /tmp/2, fd2 → /tmp/2, fd1 → /tmp/1, 2 → stderr, 0 → /tmp/2, fd3 → closed

Question **20**

Incorrect

Mark 0.00 out of 2.00

Following code claims to implement the command

/bin/ls -l | /usr/bin/head -3 | /usr/bin/tail -1

Fill in the blanks to make the code work.

Note: Do not include space in writing any option. x[1][2] should be written without any space, and so is the case with [1] or [2]. Pay attention to exact syntax and do not write any extra character like ';' or = etc.

```
int main(int argc, char *argv[]) {
   int pid1, pid2;
   int pfd[
```
1
✖][2];

```
   pipe(
```
2
✖);
```
   pid1 =
```
3
✖ ;
```
   if(pid1 != 0) {
      close(pfd[0]
```
0
✖);
```
      close(
```
pid1
✖);
```
      dup(
```
pid2
✖);
```
      execl("/bin/ls", "/bin/ls", "
```
1
✖ ", NULL);
```
   }
   pipe(
```

✖);

✖ = fork();
```
   if(pid2 == 0) {
      close(
```

✖ ;
```
      close(0);
      dup(
```

✖);
```
      close(pfd[1]
```

✖);
close(

✖);
dup(

✖);
execl("/usr/bin/head", "/usr/bin/head", "

✖ ", NULL);
} else {
close(pfd

✖);
close(

✖);
dup(

✖);
close(pfd

✖);
execl("/usr/bin/tail", "/usr/bin/tail", "

✖ ", NULL);
}
}

Question **21**

Partially correct

Mark 0.11 out of 1.00

Select all the correct statements about calling convention on x86 32-bit.

- [x] a. Return address is one location above the ebp ✔
- [x] b. Parameters may be passed in registers or on stack ✔
- [x] c. Space for local variables is allocated by substracting the stack pointer inside the code of the called function ✔
- [x] d. The ebp pointers saved on the stack constitute a chain of activation records ✔
- [x] e. The two lines in the beginning of each function, "push %ebp; mov %esp, %ebp", create space for local variables ✘
- [x] f. Parameters may be passed in registers or on stack ✔
- [x] g. The return value is either stored on the stack or returned in the eax register ✘
- [ ] h. Paramters are pushed on the stack in left-right order
- [ ] i. during execution of a function, ebp is pointing to the old ebp
- [x] j. Space for local variables is allocated by substracting the stack pointer inside the code of the caller function ✘
- [x] k. Compiler may allocate more memory on stack than needed ✔

Your answer is partially correct.

You have selected too many options.
The correct answers are: Compiler may allocate more memory on stack than needed, Parameters may be passed in registers or on stack, Parameters may be passed in registers or on stack, Return address is one location above the ebp, during execution of a function, ebp is pointing to the old ebp, Space for local variables is allocated by substracting the stack pointer inside the code of the called function, The ebp pointers saved on the stack constitute a chain of activation records

Question **22**

Correct

Mark 1.00 out of 1.00

Match the program with it's output (ignore newlines in the output. Just focus on the count of the number of 'hi')

`main() { int i = fork(); if(i == 0) execl("/usr/bin/echo", "/usr/bin/echo", "hi\n", NULL); }`	hi ✔
`main() { fork(); execl("/usr/bin/echo", "/usr/bin/echo", "hi\n", NULL); }`	hi hi ✔
`main() { int i = NULL; fork(); printf("hi\n"); }`	hi hi ✔
`main() { execl("/usr/bin/echo", "/usr/bin/echo", "hi\n", NULL); }`	hi ✔

Your answer is correct.

The correct answer is: `main() { int i = fork(); if(i == 0) execl("/usr/bin/echo", "/usr/bin/echo", "hi\n", NULL); }` → hi, `main() { fork(); execl("/usr/bin/echo", "/usr/bin/echo", "hi\n", NULL); }` → hi hi, `main() { int i = NULL; fork(); printf("hi\n"); }` → hi hi, `main() { execl("/usr/bin/echo", "/usr/bin/echo", "hi\n", NULL); }` → hi

Question **23**

Incorrect

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

Your answer is incorrect.

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 **24**

Incorrect

Mark 0.00 out of 0.50

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?

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

Your answer is incorrect.

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)

Question **25**

Incorrect

Mark 0.00 out of 1.00

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

Select one or more:

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

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

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

- [x] f. 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

Your answer is incorrect.

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 **26**

Correct

Mark 0.25 out of 0.25

Which of the following are the files related to bootloader in xv6?

- a. bootasm.s and entry.S
- b. bootasm.S and bootmain.c ✔
- c. bootasm.S, bootmain.c and bootblock.c
- d. bootmain.c and bootblock.S

Your answer is correct.

The correct answer is: bootasm.S and bootmain.c

Question **27**

Correct

Mark 0.25 out of 0.25

Match the following parts of a C program to the layout of the process in memory

Instructions	Text section ✔
Local Variables	Stack Section ✔
Dynamically allocated memory	Heap Section ✔
Global and static data	Data section ✔

Your answer is correct.

The correct answer is:
Instructions → Text section, Local Variables → Stack Section,
Dynamically allocated memory → Heap Section,
Global and static data → Data section

Question **28**

Incorrect

Mark 0.00 out of 0.50

What will this program do?

```
int main() {
fork();
execl("/bin/ls", "/bin/ls", NULL);
printf("hello");
}
```

- a. one process will run ls, another will print hello
- b. run ls once ✖
- c. run ls twice
- d. run ls twice and print hello twice
- e. run ls twice and print hello twice, but output will appear in some random order

Your answer is incorrect.

The correct answer is: run ls twice

Question **29**

Correct

Mark 0.25 out of 0.25

What is the OS Kernel?

- a. The code that controls hardware, abstracts access to hardware resources using system calls, creates an environment for processes to be created and run ✔ correct
- b. The set of tools like compiler, linker, loader, terminal, shell, etc.
- c. Only the system programs like compiler, linker, loader, etc.
- d. Everything that I see on my screen

The correct answer is: The code that controls hardware, abstracts access to hardware resources using system calls, creates an environment for processes to be created and run

Question **30**

Correct

Mark 0.50 out of 0.50

Which of the following is/are not saved during context switch?

- [ ] a. Program Counter
- [ ] b. General Purpose Registers
- [x] c. Bus ✔
- [ ] d. Stack Pointer
- [ ] e. MMU related registers/information
- [x] f. Cache ✔
- [x] g. TLB ✔

Your answer is correct.

The correct answers are: TLB, Cache, Bus

Question **31**

Partially correct

Mark 0.10 out of 0.25

Select the order in which the various stages of a compiler execute.

Linking	3	✗
Syntatical Analysis	2	✓
Pre-processing	1	✓
Intermediate code generation	does not exist	✗
Loading	4	✗

Your answer is partially correct.

You have correctly selected 2.
The correct answer is: Linking → 4, Syntatical Analysis → 2, Pre-processing → 1, Intermediate code generation → 3, Loading → does not exist

Question **32**

Partially correct

Mark 0.08 out of 0.50

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

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

Your answer is partially correct.

You have correctly selected 1.
The correct answer is: 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, timer interrupt occurs → 2, Process P0 is running → 1, Control is passed to P1 → 5

Question **33**

Not answered

Marked out of 1.00

Select the correct statements about interrupt handling in xv6 code

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

Your answer is incorrect.

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

◄ (Assignment) Change free list management in xv6

Jump to...

Started on	Tuesday, 22 March 2022, 1:59:34 PM
State	Finished
Completed on	Tuesday, 22 March 2022, 4:14:53 PM
Time taken	2 hours 15 mins
Grade	**24.57** out of 40.00 (**61**%)

Question **1**

Partially correct

Mark 0.10 out of 1.00

Select all the correct statements w.r.t user and kernel threads

Select one or more:

- [x] a. many-one model can be implemented even if there are no kernel threads ✔
- [ ] b. many-one model gives no speedup on multicore processors
- [x] c. all three models, that is many-one, one-one, many-many , require a user level thread library ✔
- [ ] d. A process blocks in many-one model even if a single thread makes a blocking system call
- [x] e. A process may not block in many-one model, if a thread makes a blocking system call ✖
- [x] f. one-one model increases kernel's scheduling load ✔
- [ ] g. one-one model can be implemented even if there are no kernel threads

Your answer is partially correct.

You have correctly selected 3.

The correct answers are: many-one model can be implemented even if there are no kernel threads, all three models, that is many-one, one-one, many-many , require a user level thread library, one-one model increases kernel's scheduling load, many-one model gives no speedup on multicore processors, A process blocks in many-one model even if a single thread makes a blocking system call

Question **2**

Partially correct

Mark 0.50 out of 1.00

Select all correct statements w.r.t. Major and Minor page faults on Linux

- [ ] a. Thrashing is possible only due to major page faults
- [ ] b. Minor page fault may occur because the page was freed, but still tagged and available in the free page list
- [x] c. Minor page fault may occur because of a page fault during fork(), on code of an already running process ✔
- [x] d. Major page faults are likely to occur in more numbers at the beginning of the process ✔
- [x] e. Minor page fault may occur because the page was a shared memory page ✔
- [ ] f. Minor page faults are an improvement of the page buffering techniques

The correct answers are: Minor page fault may occur because the page was a shared memory page, Minor page fault may occur because of a page fault during fork(), on code of an already running process, Minor page fault may occur because the page was freed, but still tagged and available in the free page list, Major page faults are likely to occur in more numbers at the beginning of the process, Thrashing is possible only due to major page faults, Minor page faults are an improvement of the page buffering techniques

Question **3**

Correct

Mark 2.00 out of 2.00

W.r.t. Memory management in xv6,

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

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

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

Question **4**

Correct

Mark 1.00 out of 1.00

Given that a kernel has 1000 KB of total memory, and holes of sizes (in that order) 300 KB, 200 KB, 100 KB, 250 KB. For each of the requests on the left side, match it with the chunk chosen using the specified algorithm.

Consider each request as first request.

50 KB, worst fit	300 KB	✔
100 KB, worst fit	300 KB	✔
200 KB, first fit	300 KB	✔
150 KB, best fit	200 KB	✔
220 KB, best fit	250 KB	✔
150 KB, first fit	300 KB	✔

The correct answer is: 50 KB, worst fit → 300 KB, 100 KB, worst fit → 300 KB, 200 KB, first fit → 300 KB, 150 KB, best fit → 200 KB, 220 KB, best fit → 250 KB, 150 KB, first fit → 300 KB

Question **5**

Partially correct

Mark 0.60 out of 1.00

Choice of the global or local replacement strategy is a subjective choice for kernel programmers. There are advantages and disadvantages on either side. Out of the following statements, that advocate either global or local replacement strategy, select those statements that have a logically CONSISTENT argument. (That is any statement that is logically correct about either global or local replacement)

Consistent	Inconsistent		
◉☑	○✖	Local replacement can lead to under-utilisation of memory, because a process may not use all the pages allocated to it all the time.	✔
○☑	◉✖	Global replacement may give highly variable per process completion time because number of page faults become un-predictable.	✖
◉☑	○✖	Global replacement can be preferred when greater throughput (number of processes completing per unit time) is a concern, because each process tries to complete at the expense of others, thus leading to overall more processes completing (unless thrashing occurs).	✔
◉☑	○✖	Local replacement results in more predictable per-process completion time because number of page faults can be better predicted.	✔
○☑	◉✖	Local replacement can be preferred when avoiding thrashing is a major concern because with local replacement and minimum number of frames allocated, a process is always able to progress and cascading inter-process page faults are avoided.	✖

Local replacement can lead to under-utilisation of memory, because a process may not use all the pages allocated to it all the time.: Consistent
Global replacement may give highly variable per process completion time because number of page faults become un-predictable.: Consistent
Global replacement can be preferred when greater throughput (number of processes completing per unit time) is a concern, because each process tries to complete at the expense of others, thus leading to overall more processes completing (unless thrashing occurs).: Consistent
Local replacement results in more predictable per-process completion time because number of page faults can be better predicted.: Consistent
Local replacement can be preferred when avoiding thrashing is a major concern because with local replacement and minimum number of frames allocated, a process is always able to progress and cascading inter-process page faults are avoided.: Consistent

Question **6**

Correct

Mark 1.00 out of 1.00

Map the functionality/use with function/variable in xv6 code.

Setup kernel part of a page table, and switch to that page table	kvmalloc() ✔
Setup kernel part of a page table, mapping kernel code, data, read-only data, I/O space, devices	setupkvm() ✔
return a free page, if available; 0, otherwise	kalloc() ✔
Create page table entries for a given range of virtual and physical addresses; including page directory entries if needed	mappages() ✔
Return address of page table entry in a given page directory, for a given virtual address; creates page table if necessary	walkpgdir() ✔
Array listing the kernel memory mappings, to be used by setupkvm()	kmap[] ✔

Your answer is correct.

The correct answer is: Setup kernel part of a page table, and switch to that page table → kvmalloc(), Setup kernel part of a page table, mapping kernel code, data, read-only data, I/O space, devices → setupkvm(), return a free page, if available; 0, otherwise → kalloc(), Create page table entries for a given range of virtual and physical addresses; including page directory entries if needed → mappages(), Return address of page table entry in a given page directory, for a given virtual address; creates page table if necessary → walkpgdir(), Array listing the kernel memory mappings, to be used by setupkvm() → kmap[]

Question **7**

Partially correct

Mark 0.20 out of 1.00

Mark statements True/False w.r.t. change of states of a process. Note that a statement is true only if the claim and argument both are true.

Reference: The process state diagram (and your understanding of how kernel code works). Note - the diagram does not show zombie state!

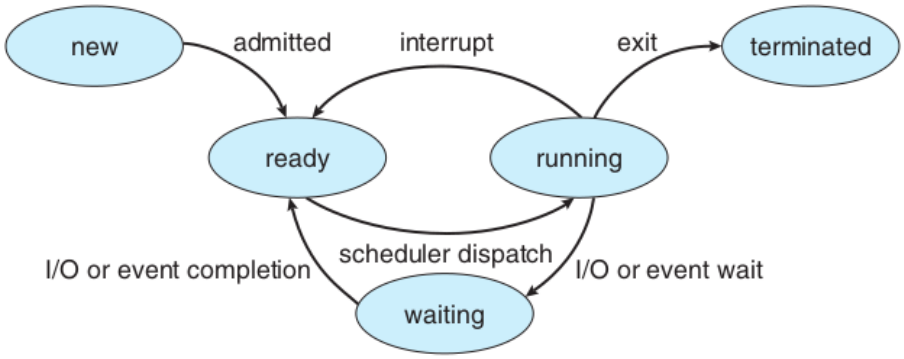


Figure 3.2 Diagram of process state.

True	False		
◉ ☑	○ ✖	Every forked process has to go through ZOMBIE state, at least for a small duration.	✔
◉ ✖	○ ☑	A process only in RUNNING state can become TERMINATED because scheduler moves it to ZOMBIE state first	✖
○ ☑	◉ ✖	A process in WAITING state can not become RUNNING because the event it's waiting for has not occurred and it has not been moved to ready queue yet	✖
○ ☑	◉ ✖	Only a process in READY state is considered by scheduler	✖
◉ ✖	○ ☑	A process in READY state can not go to WAITING state because the resource on which it will WAIT will not be in use when process is in READY state.	✖

Every forked process has to go through ZOMBIE state, at least for a small duration.: True
A process only in RUNNING state can become TERMINATED because scheduler moves it to ZOMBIE state first: False
A process in WAITING state can not become RUNNING because the event it's waiting for has not occurred and it has not been moved to ready queue yet: True
Only a process in READY state is considered by scheduler: True
A process in READY state can not go to WAITING state because the resource on which it will WAIT will not be in use when process is in READY state.: False

Question **8**

Correct

Mark 2.00 out of 2.00

Consider the reference string
6 4 2 0 1 2 6 9 2 0 5

If the number of page frames is 3, then total number of page faults (including initial), using FIFO replacement is:

Answer: 10 ✔

#6# 6,4# 6,4,2 #0,4,2# 0,1,2 #0,1,6 #9,1,6# 9,2,6# 9,2,0 #5,2,0

The correct answer is: 10

Question **9**

Incorrect

Mark 0.00 out of 1.00

Select all the correct statements about linking and loading.

Select one or more:

- [ ] a. Continuous memory management schemes can support static linking and dynamic loading. (may be inefficiently)
- [ ] b. Dynamic linking essentially results in relocatable code.
- [ ] c. Continuous memory management schemes can support static linking and static loading. (may be inefficiently)
- [ ] d. Loader is last stage of the linker program
- [ ] e. Dynamic linking and loading is not possible without demand paging or demand segmentation.
- [ ] f. Static linking leads to non-relocatable code
- [ ] g. Continuous memory management schemes can support dynamic linking and dynamic loading.
- [x] h. Dynamic linking is possible with continous memory management, but variable sized partitions only. ✖
- [x] i. Loader is part of the operating system ✔

Your answer is incorrect.

The correct answers are: Continuous memory management schemes can support static linking and static loading. (may be inefficiently), Continuous memory management schemes can support static linking and dynamic loading. (may be inefficiently), Dynamic linking essentially results in relocatable code., Loader is part of the operating system, Dynamic linking and loading is not possible without demand paging or demand segmentation.

Question **10**

Correct

Mark 1.00 out of 1.00

Consider a computer system with a 32-bit logical address and 4- KB page size. The system supports up to 512 MB of physical memory. How many entries are there in each of the following?

Write answer as a decimal number.

A conventional, single-level page table:

1048576

✔

An inverted page table:

131072

✔

Question **11**

Incorrect

Mark 0.00 out of 1.00

W.r.t. xv6 code, match the state of a process with a code that sets the state

RUNNING	Choose...
ZOMBIE	Choose...
EMBRYO	Choose...
SLEEPING	Choose...
UNUSED	Choose...
RUNNABLE	scheduler() ✖

The correct answer is: RUNNING → scheduler(), ZOMBIE → exit(), called by process itself, EMBRYO → fork()->allocproc() before setting up the UVM, SLEEPING → sleep(), called by any process blocking itself, UNUSED → wait(), called by parent process, RUNNABLE → wakeup(), called by an interrupt handler

Question **12**

Not answered

Marked out of 1.00

Select the correct statements about interrupt handling in xv6 code

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

Your answer is incorrect.

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 **13**

Correct

Mark 1.00 out of 1.00

The complete range of virtual addresses (after main() in main.c is over), from which the free pages used by kalloc() and kfree() is derived,are:

- a. end, (4MB + PHYSTOP)
- b. P2V(end), PHYSTOP
- c. end, P2V(4MB + PHYSTOP)
- d. end, PHYSTOP
- e. end, 4MB
- f. end, P2V(PHYSTOP) ✔
- g. P2V(end), P2V(PHYSTOP)

Your answer is correct.

The correct answer is: end, P2V(PHYSTOP)

Question **14**

Correct

Mark 1.00 out of 1.00

Select all the correct statements about MMU and it's functionality (on a non-demand paged system)

Select one or more:

- a. Illegal memory access is detected in hardware by MMU and a trap is raised ✔
- b. Illegal memory access is detected by operating system
- c. MMU is a separate chip outside the processor
- d. The operating system interacts with MMU for every single address translation
- e. Logical to physical address translations in MMU are done in hardware, automatically ✔
- f. Logical to physical address translations in MMU are done with specific machine instructions
- g. MMU is inside the processor ✔
- h. The Operating system sets up relevant CPU registers to enable proper MMU translations ✔

Your answer is correct.

The correct answers are: MMU is inside the processor, Logical to physical address translations in MMU are done in hardware, automatically, The Operating system sets up relevant CPU registers to enable proper MMU translations, Illegal memory access is detected in hardware by MMU and a trap is raised

Question **15**

Incorrect

Mark 0.00 out of 2.00

Order the following events, in the creation of init() process in xv6:

1. ✖ initcode is selected by scheduler for execution
2. ✖ kernel stack is allocated for initcode process
3. ✖ values are set in the trapframe of initcode
4. ✖ sys_exec runs
5. ✖ initcode process is set to be runnable
6. ✖ code is set to start in forkret() when process gets scheduled
7. ✖ Arguments on setup on process stack for /init
8. ✖ trapframe and context pointers are set to proper location
9. ✖ trap() runs
10. ✖ userinit() is called
11. ✖ the header of "/init" ELF file is ready by kernel
12. ✖ empty struct proc is obtained for initcode
13. ✖ Stack is allocated for "/init" process
14. ✖ function pointer from syscalls[] array is invoked
15. ✖ memory mappings are created for "/init" process
16. ✖ page table mappings of 'initcode' are replaced by makpings of 'init'
17. ✖ initcode calls exec system call
18. ✖ initcode process runs
19. ✖ name of process "/init" is copied in struct proc

Your answer is incorrect.

Grading type: Relative to the next item (including last)

Grade details: 0 / 19 = 0%

Here are the scores for each item in this response:

1. 0 / 1 = 0%
2. 0 / 1 = 0%
3. 0 / 1 = 0%
4. 0 / 1 = 0%
5. 0 / 1 = 0%
6. 0 / 1 = 0%
7. 0 / 1 = 0%
8. 0 / 1 = 0%
9. 0 / 1 = 0%
10. 0 / 1 = 0%
11. 0 / 1 = 0%
12. 0 / 1 = 0%
13. 0 / 1 = 0%
14. 0 / 1 = 0%
15. 0 / 1 = 0%

16. 0 / 1 = 0%
17. 0 / 1 = 0%
18. 0 / 1 = 0%
19. 0 / 1 = 0%

The correct order for these items is as follows:

1. userinit() is called
2. empty struct proc is obtained for initcode
3. kernel stack is allocated for initcode process
4. trapframe and context pointers are set to proper location
5. code is set to start in forkret() when process gets scheduled
6. kernel memory mappings are created for initcode
7. values are set in the trapframe of initcode
8. initcode process is set to be runnable
9. initcode is selected by scheduler for execution
10. initcode process runs
11. initcode calls exec system call
12. trap() runs
13. function pointer from syscalls[] array is invoked
14. sys_exec runs
15. the header of "/init" ELF file is ready by kernel
16. memory mappings are created for "/init" process
17. Stack is allocated for "/init" process
18. Arguments on setup on process stack for /init
19. name of process "/init" is copied in struct proc
20. page table mappings of 'initcode' are replaced by makpings of 'init'

Question **16**

Partially correct

Mark 0.56 out of 1.00

Mark the statements as True or False, w.r.t. mmap()

True	False		
○ ☑	◉ ✖	mmap() can be implemented on both demand paged and non-demand paged systems.	✖
◉ ✖	○ ☑	MAP_FIXED guarantees that the mapping is always done at the specified address	✖
◉ ☑	○ ✖	MAP_PRIVATE leads to a mapping that is copy-on-write	✔
◉ ☑	○ ✖	on failure mmap() returns (void *)-1	✔
○ ✖	◉ ☑	MAP_SHARED leads to a mapping that is copy-on-write	✔
◉ ✖	○ ☑	mmap() results in changes to buffer-cache of the kernel.	✖
○ ✖	◉ ☑	on failure mmap() returns NULL	✔
○ ☑	◉ ✖	mmap() results in changes to page table of a process.	✖
◉ ☑	○ ✖	mmap() is a system call	✔

mmap() can be implemented on both demand paged and non-demand paged systems.: True
MAP_FIXED guarantees that the mapping is always done at the specified address: False
MAP_PRIVATE leads to a mapping that is copy-on-write: True
on failure mmap() returns (void *)-1: True
MAP_SHARED leads to a mapping that is copy-on-write: False
mmap() results in changes to buffer-cache of the kernel.: False
on failure mmap() returns NULL: False
mmap() results in changes to page table of a process.: True
mmap() is a system call: True

Question **17**

Incorrect

Mark 0.00 out of 1.00

If one thread opens a file with read privileges then

Select one:

- a. other threads in the same process can also read from that file
- b. none of these
- c. any other thread cannot read from that file
- d. other threads in the another process can also read from that file ✖

Your answer is incorrect.

The correct answer is: other threads in the same process can also read from that file

Question **18**

Partially correct

Mark 0.60 out of 1.00

Mark the statements about named and un-named pipes as True or False

True	False		
◉☑	○✖	Named pipe exists as a file	✔
◉☑	○✖	Un-named pipes are inherited by a child process from parent.	✔
◉✖	○☑	The buffers for named-pipe are in process-memory while the buffers for the un-named pipe are in kernel memory.	✖
◉☑	○✖	Both types of pipes are an extension of the idea of "message passing".	✔
◉✖	○☑	A named pipe has a name decided by the kernel.	✖
○☑	◉✖	Un-named pipes can be used for communication between only "related" processes, if the common ancestor created it.	✖
◉☑	○✖	Both types of pipes provide FIFO communication.	✔
○✖	◉☑	Named pipes can be used for communication between only "related" processes.	✔
◉☑	○✖	Named pipes can exist beyond the life-time of processes using them.	✔
◉✖	○☑	The pipe() system call can be used to create either a named or un-named pipe.	✖

Named pipe exists as a file: True
Un-named pipes are inherited by a child process from parent.: True
The buffers for named-pipe are in process-memory while the buffers for the un-named pipe are in kernel memory.: False
Both types of pipes are an extension of the idea of "message passing".: True
A named pipe has a name decided by the kernel.: False
Un-named pipes can be used for communication between only "related" processes, if the common ancestor created it.: True
Both types of pipes provide FIFO communication.: True
Named pipes can be used for communication between only "related" processes.: False
Named pipes can exist beyond the life-time of processes using them.: True
The pipe() system call can be used to create either a named or un-named pipe.: False

Question **19**

Partially correct

Mark 0.67 out of 1.00

Select the most common causes of use of IPC by processes

- [ ] a. More modular code
- [x] b. Breaking up a large task into small tasks and speeding up computation, on multiple core machines ✔
- [ ] c. More security checks
- [x] d. Sharing of information of common interest ✔
- [ ] e. Get the kernel performance statistics

The correct answers are: Sharing of information of common interest, Breaking up a large task into small tasks and speeding up computation, on multiple core machines, More modular code

Question **20**

Correct

Mark 1.00 out of 1.00

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

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

Your answer is correct.

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

Question **21**

Partially correct

Mark 0.50 out of 1.00

Mark whether the given sequence of events is possible or not-possible. Also, select the reason for your answer.

For each sequence it's a not-possible sequence if some important event is not mentioned in the sequence.

Assume that the kernel code is non-interruptible and uniprocessor system.

```
Process P1, user code executing
Timer interrupt
Context changes to kernel context
Generic interrupt handler runs
Generic interrupt handler calls Scheduler
Scheduler selects P2 for execution
After scheduler, Process P2 user code executing
```

This sequence of events is: not-possible ✔

Because

Generic interrupt handler can not call scheduler ✖

Question **22**

Partially correct

Mark 0.63 out of 1.00

Mark the statements as True or False, w.r.t. passing of arguments to system calls in xv6 code.

True	False		
◉✖	○☑	Integer arguments are stored in eax, ebx, ecx, etc. registers	✖
◉☑	○✖	String arguments are NOT copied in kernel memory, but just pointed to by a kernel memory pointer	✔
◉☑	○✖	The functions like argint(), argstr() make the system call arguments available in the kernel.	✔
◉✖	○☑	String arguments are first copied to trapframe and then from trapframe to kernel's other variables.	✖
◉☑	○✖	The arguments to system call originally reside on process stack.	✔
◉✖	○☑	The arguments to system call are copied to kernel stack in trapasm.S	✖
◉☑	○✖	Integer arguments are copied from user memory to kernel memory using argint()	✔
◉☑	○✖	The arguments are accessed in the kernel code using esp on the trapframe.	✔

Integer arguments are stored in eax, ebx, ecx, etc. registers: False
String arguments are NOT copied in kernel memory, but just pointed to by a kernel memory pointer: True
The functions like argint(), argstr() make the system call arguments available in the kernel.: True
String arguments are first copied to trapframe and then from trapframe to kernel's other variables.: False
The arguments to system call originally reside on process stack.: True
The arguments to system call are copied to kernel stack in trapasm.S: False
Integer arguments are copied from user memory to kernel memory using argint(): True
The arguments are accessed in the kernel code using esp on the trapframe.: True

Question **23**

Not answered

Marked out of 1.00

Given below is a sequence of reference bits on pages before the second chance algorithm runs. Before the algorithm runs, the counter is at the page marked (x). Write the sequence of reference bits after the second chance algorithm has executed once. In the answer write PRECISELY one space BETWEEN each number and do not mention (x).

0 0 1(x) 1 0 1 1

Answer:

The correct answer is: 0 0 0 0 0 1 1

Question **24**

Correct

Mark 2.00 out of 2.00

For the reference string
3 4 3 5 2

using FIFO replacement policy for pages,

consider the number of page faults for 2, 3 and 4 page frames.
Select the correct statement.

Select one:

- a. Exhibit Balady's anomaly between 3 and 4 frames
- b. Do not exhibit Balady's anomaly
- c. Exhibit Balady's anomaly between 2 and 3 frames

Your answer is correct.

The correct answer is: Do not exhibit Balady's anomaly

Question **25**

Correct

Mark 1.00 out of 1.00

For the reference string
3 4 3 5 2

using LRU replacement policy for pages,

consider the number of page faults for 2, 3 and 4 page frames.
Select the most correct statement.

Select one:

- a. LRU will never exhibit Balady's anomaly ✔
- b. Exhibit Balady's anomaly between 2 and 3 frames
- c. This example does not exhibit Balady's anomaly
- d. Exhibit Balady's anomaly between 3 and 4 frames

Your answer is correct.

The correct answer is: LRU will never exhibit Balady's anomaly

Question **26**

Partially correct

Mark 0.55 out of 1.00

Select all the correct statements about process states.

Note that in this question you lose marks for every incorrect choice that you make, proportional to actual number of incorrect choices.

- [ ] a. Process state is implemented as a string
- [x] b. Process state is stored in the PCB ✔
- [ ] c. A process becomes ZOMBIE when another process bites into it's memory
- [x] d. Process state is stored in the processor ✖
- [ ] e. The scheduler can change state of a process from RUNNALBE to RUNNING and vice-versa
- [x] f. The scheduler can change state of a process from RUNNALBE to RUNNING ✔
- [x] g. A process becomes ZOMBIE when it calls exit() ✔
- [ ] h. Process state is changed only by interrupt handlers
- [ ] i. Process state can be implemented as just a number

Your answer is partially correct.

You have correctly selected 3.

The correct answers are: Process state is stored in the PCB, Process state can be implemented as just a number, The scheduler can change state of a process from RUNNALBE to RUNNING, A process becomes ZOMBIE when it calls exit()

Question **27**

Partially correct

Mark 0.38 out of 1.00

Consider a demand-paging system with the following time-measured utilizations:
CPU utilization : 20%
Paging disk: 97.7%
Other I/O devices: 5%

For each of the following, indicate whether it will (or is likely to) improve CPU utilization (even if by a small amount). Explain your answers.

a. Install a faster CPU : Yes ✖

b. Install a bigger paging disk. : Yes ✖

c. Increase the degree of multiprogramming. : Yes ✖

d. Decrease the degree of multiprogramming. : Yes ✔

e. Install more main memory.: Yes ✔

f. Install a faster hard disk or multiple controllers with multiple hard disks. : Yes ✔

g. Add prepaging to the page-fetch algorithms. :
May be ✖

h. Increase the page size. : May be ✖

Question **28**

Incorrect

Mark 0.00 out of 1.00

Suppose a kernel uses a buddy allocator. The smallest chunk that can be allocated is of size 32 bytes. One bit is used to track each such chunk, where 1 means allocated and 0 means free. The chunk looks like this as of now:

10011010

Now, there is a request for a chunk of 50 bytes.

After this allocation, the bitmap, indicating the status of the buddy allocator will be

Answer: 10110010 ✖

The correct answer is: 11111010

Question **29**

Partially correct

Mark 0.75 out of 1.00

Select the correct points of comparison between POSIX and System V shared memory.

- [x] a. POSIX shared memory is newer than System V shared memory ✔
- [x] b. POSIX shared memory is "thread safe", System V is not ✔
- [x] c. System V is more prevalent than POSIX even today ✔
- [ ] d. POSIX allows giving name to shared memory, System V does not

The correct answers are: POSIX shared memory is newer than System V shared memory, POSIX shared memory is "thread safe", System V is not, POSIX allows giving name to shared memory, System V does not, System V is more prevalent than POSIX even today

Question **30**

Partially correct

Mark 0.67 out of 1.00

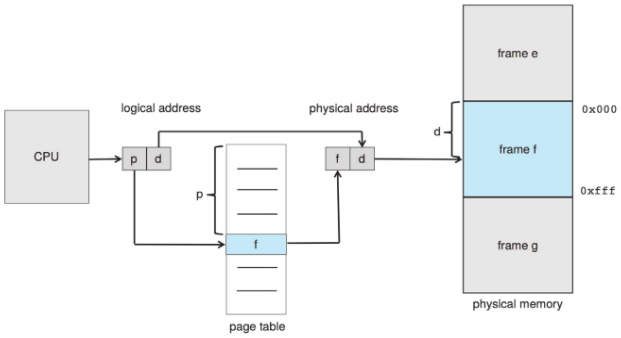


Figure 9.8 Paging hardware.

Mark the statements as True or False, w.r.t. the above diagram (note that the diagram does not cover all details of what actually happens!)

True	False		
◉	○	The combining of f and d is done by MMU	✔
○	◉	There are total 3 memory references in this diagram	✖
◉	○	The split of logical address into p and d is done by MMU	✔
◉	○	The page table is in physical memory and must be continuous	✔
○	◉	Using the offset d in the physical page-frame is done by MMU	✖
◉	○	The logical address issued by CPU is the same one generated by compiler	✔

The combining of f and d is done by MMU: True
There are total 3 memory references in this diagram: False
The split of logical address into p and d is done by MMU: True
The page table is in physical memory and must be continuous: True
Using the offset d in the physical page-frame is done by MMU: False
The logical address issued by CPU is the same one generated by compiler: True

Question **31**

Partially correct

Mark 0.50 out of 1.00

Select all the correct statements about signals

Select one or more:

- [ ] a. SIGKILL definitely kills a process because it's code runs in kernel mode of CPU
- [x] b. Signals are delivered to a process by another process ✖
- [ ] c. The signal handler code runs in kernel mode of CPU
- [x] d. SIGKILL definitely kills a process because it can't be caught or ignored, and it's default action terminates the process ✔
- [x] e. The signal handler code runs in user mode of CPU ✔
- [x] f. A signal handler can be invoked asynchronously or synchronously depending on signal type ✔
- [ ] g. Signal handlers once replaced can't be restored
- [ ] h. Signals are delivered to a process by kernel

Your answer is partially correct.

You have correctly selected 3.
The correct answers are: Signals are delivered to a process by kernel, A signal handler can be invoked asynchronously or synchronously depending on signal type, The signal handler code runs in user mode of CPU, SIGKILL definitely kills a process because it can't be caught or ignored, and it's default action terminates the process

Question **32**

Correct

Mark 1.00 out of 1.00

The data structure used in kalloc() and kfree() in xv6 is

- ( ) a. Singly linked circular list
- (•) b. Singly linked NULL terminated list ✔
- ( ) c. Double linked NULL terminated list
- ( ) d. Doubly linked circular list

Your answer is correct.

The correct answer is: Singly linked NULL terminated list

Question **33**

Partially correct

Mark 1.78 out of 2.00

Match the description of a memory management function with the name of the function that provides it, in xv6

Load contents from ELF into existing pages	loaduvm()	✔
Mark the page as in-accessible	clearpteu()	✔
setup the kernel part in the page table	setupkvm()	✔
Switch to kernel page table	switchkvm()	✔
Create a copy of the page table of a process	copyuvm()	✔
Copy the code pages of a process	No such function	✔
Setup and load the user page table for initcode process	inituvm()	✔
Switch to user page table	switchuvm()	✔
Load contents from ELF into pages after allocating the pages first	inituvm()	✖

The correct answer is: Load contents from ELF into existing pages → loaduvm(), Mark the page as in-accessible → clearpteu(), setup the kernel part in the page table → setupkvm(), Switch to kernel page table → switchkvm(), Create a copy of the page table of a process → copyuvm(), Copy the code pages of a process → No such function, Setup and load the user page table for initcode process → inituvm(), Switch to user page table → switchuvm(), Load contents from ELF into pages after allocating the pages first → No such function

Question **34**

Partially correct

Mark 0.60 out of 1.00

Mark the statements as True or False, w.r.t. thrashing

True	False		
✖	☑	Thrashing occurs because some process is doing lot of disk I/O.	✖
☑	✖	Processes keep changing their locality of reference, and a high rate of page faults occur when they are changing the locality.	✔
✖	☑	mmap() solves the problem of thrashing.	✔
☑	✖	The working set model is an attempt at approximating the locality of a process.	✔
☑	✖	Thrashing is particular to demand paging systems, and does not apply to pure paging systems.	✔
✖	☑	Processes keep changing their locality of reference, and least number of page faults occur when they are changing the locality.	✖
✖	☑	Thrashing can occur even if entire memory is not in use.	✖
☑	✖	During thrashing the CPU is under-utilised as most time is spent in I/O	✖
☑	✖	Thrashing can be limited if local replacement is used.	✔
☑	✖	Thrashing occurs when the total size of all processe's locality exceeds total memory size.	✔

Thrashing occurs because some process is doing lot of disk I/O.: False
Processes keep changing their locality of reference, and a high rate of page faults occur when they are changing the locality.: True
mmap() solves the problem of thrashing.: False
The working set model is an attempt at approximating the locality of a process.: True
Thrashing is particular to demand paging systems, and does not apply to pure paging systems.: True
Processes keep changing their locality of reference, and least number of page faults occur when they are changing the locality.: False
Thrashing can occur even if entire memory is not in use.: False
During thrashing the CPU is under-utilised as most time is spent in I/O: True
Thrashing can be limited if local replacement is used.: True
Thrashing occurs when the total size of all processe's locality exceeds total memory size.: True

Question **35**

Correct

Mark 1.00 out of 1.00

After virtual memory is implemented

(select T/F for each of the following)One Program's size can be larger than physical memory size

True	False		
◉ ☑	○ ✖	Cumulative size of all programs can be larger than physical memory size	✔
◉ ☑	○ ✖	Code need not be completely in memory	✔
◉ ☑	○ ✖	One Program's size can be larger than physical memory size	✔
○ ✖	◉ ☑	Virtual addresses become available to executing process	✔
○ ✖	◉ ☑	Virtual access to memory is granted to all processes	✔
◉ ☑	○ ✖	Relatively less I/O may be possible during process execution	✔
◉ ☑	○ ✖	Logical address space could be larger than physical address space	✔

Cumulative size of all programs can be larger than physical memory size: True
Code need not be completely in memory: True
One Program's size can be larger than physical memory size: True
Virtual addresses become available to executing process: False
Virtual access to memory is granted to all processes: False
Relatively less I/O may be possible during process execution: True
Logical address space could be larger than physical address space: True

◄ (Optional Assignment) lseek system call in xv6

Jump to...

Feedback on Quiz-2 ►

Started on	Thursday, 18 March 2021, 2:46 PM
State	Finished
Completed on	Thursday, 18 March 2021, 3:50 PM
Time taken	1 hour 4 mins
Grade	**10.36** out of 20.00 (**52**%)

Question **1**

Partially correct

Mark 0.57 out of 1.00

Mark True, the actions done as part of code of swtch() in swtch.S, in xv6

True	False		
◉☑	○✖	Restore new callee saved registers from kernel stack of new context	✔
◉☑	○✖	Save old callee saved registers on kernel stack of old context	✔
○✖	◉☑	Save old callee saved registers on user stack of old context	✔
◉✖	○☑	Switch from old process context to new process context	✖
○☑	◉✖	Switch from one stack (old) to another(new)	✖
○✖	◉☑	Restore new callee saved registers from user stack of new context	✔
◉✖	○☑	Jump to code in new context	✖

Restore new callee saved registers from kernel stack of new context: True
Save old callee saved registers on kernel stack of old context: True
Save old callee saved registers on user stack of old context: False
Switch from old process context to new process context: False
Switch from one stack (old) to another(new): True
Restore new callee saved registers from user stack of new context: False
Jump to code in new context: False

Question **2**

Partially correct

Mark 0.17 out of 0.50

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

bootmain()	gdt setup with 3 entries, right from first line of code of bootloader	✗
kvmalloc() in main()	gdt setup with 5 entries (0 to 4) on one processor	✗
after startothers() in main()	gdt setup with 5 entries (0 to 4) on all processors	✓
after seginit() in main()	gdt setup with 5 entries (0 to 4) on all processors	✗
bootasm.S	gdt setup with 3 entries, right from first line of code of bootloader	✗
entry.S	gdt setup with 3 entries, at start32 symbol of bootasm.S	✓

Your answer is partially correct.

You have correctly selected 2.

The correct answer is: bootmain() → gdt setup with 3 entries, at start32 symbol of bootasm.S, kvmalloc() in main() → 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, 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, entry.S → gdt setup with 3 entries, at start32 symbol of bootasm.S

Question **3**

Partially correct

Mark 0.38 out of 1.00

Compare paging with demand paging and select the correct statements.

Select one or more:

- [x] a. The meaning of valid-invalid bit in page table is different in paging and demand-paging. ✔
- [x] b. Demand paging requires additional hardware support, compared to paging. ✔
- [ ] c. Paging requires some hardware support in CPU
- [x] d. With paging, it's possible to have user programs bigger than physical memory. ✖
- [x] e. Both demand paging and paging support shared memory pages. ✔
- [ ] f. Demand paging always increases effective memory access time.
- [x] g. With demand paging, it's possible to have user programs bigger than physical memory. ✔
- [x] h. Calculations of number of bits for page number and offset are same in paging and demand paging. ✔
- [ ] i. TLB hit ration has zero impact in effective memory access time in demand paging.
- [ ] j. Paging requires NO hardware support in CPU

Your answer is partially correct.

You have correctly selected 5.
The correct answers are: Demand paging requires additional hardware support, compared to paging., Both demand paging and paging support shared memory pages., With demand paging, it's possible to have user programs bigger than physical memory., Demand paging always increases effective memory access time., Paging requires some hardware support in CPU, Calculations of number of bits for page number and offset are same in paging and demand paging., The meaning of valid-invalid bit in page table is different in paging and demand-paging.

Question **4**

Partially correct

Mark 0.44 out of 0.50

Suppose a processor supports base(relocation register) + limit scheme of MMU.

Assuming this, mark the statements as True/False

True	False		
◉ ☑	○ ✖	The OS may terminate the process while handling the interrupt of memory violation	✔
◉ ☑	○ ✖	The hardware detects any memory access beyond the limit value and raises an interrupt	✔
◉ ✖	○ ☑	The hardware may terminate the process while handling the interrupt of memory violation	✖
◉ ☑	○ ✖	The OS sets up the relocation and limit registers when the process is scheduled	✔
◉ ☑	○ ✖	The compiler generates machine code assuming continuous memory address space for process, and calculating appropriate sizes for code, and data;	✔
○ ✖	◉ ☑	The process sets up it's own relocation and limit registers when the process is scheduled	✔
○ ✖	◉ ☑	The OS detects any memory access beyond the limit value and raises an interrupt	✔
○ ✖	◉ ☑	The compiler generates machine code assuming appropriately sized semgments for code, data and stack.	✔

The OS may terminate the process while handling the interrupt of memory violation: True
The hardware detects any memory access beyond the limit value and raises an interrupt: True
The hardware may terminate the process while handling the interrupt of memory violation: False
The OS sets up the relocation and limit registers when the process is scheduled: True
The compiler generates machine code assuming continuous memory address space for process, and calculating appropriate sizes for code, and data;: True
The process sets up it's own relocation and limit registers when the process is scheduled: False
The OS detects any memory access beyond the limit value and raises an interrupt: False
The compiler generates machine code assuming appropriately sized semgments for code, data and stack.: False

Question **5**

Correct

Mark 0.50 out of 0.50

Consider the following list of free chunks, in continuous memory management:

10k, 25k, 12k, 7k, 9k, 13k

Suppose there is a request for chunk of size 9k, then the free chunk selected under each of the following schemes will be

Best fit:

9k ✔

First fit:

10k ✔

Worst fit:

25k ✔

Question **6**

Partially correct

Mark 0.50 out of 1.00

Select all the correct statements about MMU and it's functionality

Select one or more:

- [ ] a. MMU is a separate chip outside the processor
- [x] b. MMU is inside the processor ✔
- [ ] c. Logical to physical address translations in MMU are done with specific machine instructions
- [x] d. The operating system interacts with MMU for every single address translation ✖
- [x] e. Illegal memory access is detected in hardware by MMU and a trap is raised ✔
- [ ] f. The Operating system sets up relevant CPU registers to enable proper MMU translations
- [x] g. Logical to physical address translations in MMU are done in hardware, automatically ✔
- [ ] h. Illegal memory access is detected by operating system

Your answer is partially correct.

You have correctly selected 3.
The correct answers are: MMU is inside the processor, Logical to physical address translations in MMU are done in hardware, automatically, The Operating system sets up relevant CPU registers to enable proper MMU translations, Illegal memory access is detected in hardware by MMU and a trap is raised

Question **7**

Incorrect

Mark 0.00 out of 0.50

Assuming a 8- KB page size, what is the page numbers for the address 874815 reference in decimal :

(give answer also in decimal)

Answer: 2186 ✖

The correct answer is: 107

Question **8**

Incorrect

Mark 0.00 out of 0.25

Select the compiler's view of the process's address space, for each of the following MMU schemes:
(Assume that each scheme,e.g. paging/segmentation/etc is effectively utilised)

Segmentation, then paging	Many continuous chunks each of page size ✖
Relocation + Limit	Many continuous chunks of same size ✖
Segmentation	one continuous chunk ✖
Paging	many continuous chunks of variable size ✖

Your answer is incorrect.

The correct answer is: Segmentation, then paging → many continuous chunks of variable size, Relocation + Limit → one continuous chunk, Segmentation → many continuous chunks of variable size, Paging → one continuous chunk

Question **9**

Incorrect

Mark 0.00 out of 0.50

Suppose the memory access time is 180ns and TLB hit ratio is 0.3, then effective memory access time is (in nanoseconds);

Answer: 192 ✖

The correct answer is: 306.00

Question **10**

Correct

Mark 0.50 out of 0.50

In xv6, The struct context is given as

```
struct context {
  uint edi;
  uint esi;
  uint ebx;
  uint ebp;
  uint eip;
};
```

Select all the reasons that explain why only these 5 registers are included in the struct context.

- [x] a. The segment registers are same across all contexts, hence they need not be saved ✔
- [x] b. esp is not saved in context, because context{} is on stack and it's address is always argument to swtch() ✔
- [ ] c. xv6 tries to minimize the size of context to save memory space
- [ ] d. esp is not saved in context, because it's not part of the context
- [x] e. eax, ecx, edx are caller save, hence no need to save ✔

Your answer is correct.

The correct answers are: The segment registers are same across all contexts, hence they need not be saved, eax, ecx, edx are caller save, hence no need to save, esp is not saved in context, because context{} is on stack and it's address is always argument to swtch()

Question **11**

Partially correct

Mark 0.83 out of 1.50

Arrange the following events in order, in page fault handling:

Disk interrupt wakes up the process	7 ✔
The reference bit is found to be invalid by MMU	1 ✔
OS makes available an empty frame	6 ✖
Restart the instruction that caused the page fault	9 ✔
A hardware interrupt is issued	3 ✖
OS schedules a disk read for the page (from backing store)	5 ✔
Process is kept in wait state	4 ✖
Page tables are updated for the process	8 ✔
Operating system decides that the page was not in memory	2 ✖

Your answer is partially correct.

You have correctly selected 5.

The correct answer is: Disk interrupt wakes up the process → 7, The reference bit is found to be invalid by MMU → 1, OS makes available an empty frame → 4, Restart the instruction that caused the page fault → 9, A hardware interrupt is issued → 2, OS schedules a disk read for the page (from backing store) → 5, Process is kept in wait state → 6, Page tables are updated for the process → 8, Operating system decides that the page was not in memory → 3

Question **12**

Incorrect

Mark 0.00 out of 0.50

Suppose a kernel uses a buddy allocator. The smallest chunk that can be allocated is of size 32 bytes. One bit is used to track each such chunk, where 1 means allocated and 0 means free. The chunk looks like this as of now:

00001010

Now, there is a request for a chunk of 70 bytes.

After this allocation, the bitmap, indicating the status of the buddy allocator will be

Answer: 11101010 ✖

The correct answer is: 11111010

Question **13**

Incorrect

Mark 0.00 out of 0.25

The complete range of virtual addresses (after main() in main.c is over), from which the free pages used by kalloc() and kfree() is derived,are:

- a. end, 4MB
- b. P2V(end), P2V(PHYSTOP)
- c. end, P2V(4MB + PHYSTOP)
- d. P2V(end), PHYSTOP ✖
- e. end, (4MB + PHYSTOP)
- f. end, PHYSTOP
- g. end, P2V(PHYSTOP)

Your answer is incorrect.

The correct answer is: end, P2V(PHYSTOP)

Question **14**

Partially correct

Mark 0.33 out of 0.50

Match the pair

Hashed page table	Linear search on collsion done by OS (e.g. SPARC Solaris) typically	✔
Inverted Page table	Linear/Parallel search using frame number in page table	✖
Hierarchical Paging	More memory access time per hierarchy	✔

Your answer is partially correct.

You have correctly selected 2.
The correct answer is: Hashed page table → Linear search on collsion done by OS (e.g. SPARC Solaris) typically, Inverted Page table → Linear/Parallel search using page number in page table, Hierarchical Paging → More memory access time per hierarchy

Question **15**

Partially correct

Mark 0.29 out of 0.50

After virtual memory is implemented

(select T/F for each of the following)One Program's size can be larger than physical memory size

True	False		
◉☑	○✖	Code need not be completely in memory	✔
◉☑	○✖	Cumulative size of all programs can be larger than physical memory size	✔
◉✖	○☑	Virtual access to memory is granted	✖
◉☑	○✖	Logical address space could be larger than physical address space	✔
◉✖	○☑	Virtual addresses are available	✖
○☑	◉✖	Relatively less I/O may be possible during process execution	✖
◉☑	○✖	One Program's size can be larger than physical memory size	✔

Code need not be completely in memory: True
Cumulative size of all programs can be larger than physical memory size: True
Virtual access to memory is granted: False
Logical address space could be larger than physical address space: True
Virtual addresses are available: False
Relatively less I/O may be possible during process execution: True
One Program's size can be larger than physical memory size: True

Question **16**

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

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

Question **17**

Incorrect

Mark 0.00 out of 1.50

Consider the reference string
6 4 2 0 1 2 6 9 2 0 5

If the number of page frames is 3, then total number of page faults (including initial), using LRU replacement is:

Answer: 8 ✖

#6# 6,4# 6,4,2 # 0,4,2#0,1,2#6,1,2#6,9,2#0,9,2#0,5,2

The correct answer is: 9

Question **18**

Partially correct

Mark 0.31 out of 0.50

Consider the image given below, which explains how paging works.

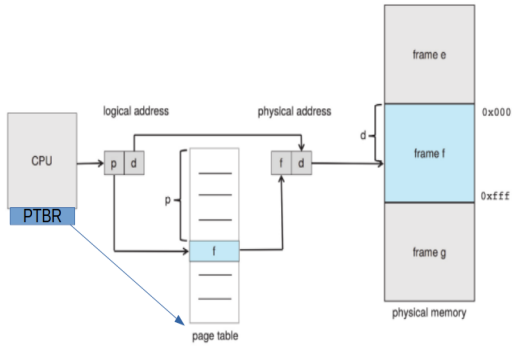


Figure 9.8 Paging hardware.

Mention whether each statement is True or False, with respect to this image.

True	False		
◉	○	The PTBR is present in the CPU as a register	✔
○	◉	The page table is indexed using frame number	✔
○	◉	The page table is indexed using page number	✖
◉	○	The locating of the page table using PTBR also involves paging translation	✖
○	◉	Size of page table is always determined by the size of RAM	✔
◉	○	The page table is itself present in Physical memory	✔
○	○	Maximum Size of page table is determined by number of bits used for page number	✖
◉	○	The physical address may not be of the same size (in bits) as the logical address	✔

The PTBR is present in the CPU as a register: True
The page table is indexed using frame number: False
The page table is indexed using page number: True
The locating of the page table using PTBR also involves paging translation: False
Size of page table is always determined by the size of RAM: False
The page table is itself present in Physical memory: True
Maximum Size of page table is determined by number of bits used for page number: True
The physical address may not be of the same size (in bits) as the logical address: True

Question **19**

Correct

Mark 2.00 out of 2.00

Given below is shared memory code with two processes sharing a memory segment.
The first process sends a user input string to second process. The second capitalizes the string. Then the first process prints the capitalized version.
Fill in the blanks to complete the code.

// First process

```
#define SHMSZ    27

int main()
{
   char c;
   int shmid;
   key_t key;
   char *shm, *s, string[128];
   key = 5679;
   if ((shmid =
```

shmget

```
   (key, SHMSZ, IPC_CREAT | 0666)) < 0) {
      perror("shmget");
      exit(1);
   }
   if ((shm =
```

shmat

```
   (shmid, NULL, 0)) == (char *) -1) {
      perror("shmat");
      exit(1);
   }
   s = shm;
   *s = '$';
   scanf("%s", string);
   strcpy(s + 1, string);
   *s = '
```

@

```
   '; //note the quotes
   while(*s != '
```

$

```
   ')
      sleep(1);
   printf("%s\n", s + 1);
   exit(0);
}
```

//Second process

```
#define SHMSZ    27
int main()
{
   int shmid;
   key_t key;
   char *shm, *s;
   int i;
   char string[128];
   key =
```

5679

✔ ;

```
if ((shmid = shmget(key, SHMSZ, 0666)) < 0) {
    perror("shmget");
    exit(1);
}
if ((shm = shmat(shmid, NULL, 0)) == (char *) -1) {
    perror("shmat");
    exit(1);
}
s =
```

shm

✔ ;

```
while(*s != '@')
    sleep(1);
for(i = 0; i < strlen(s + 1); i++)
    s[i + 1] =  toupper(s[i + 1]);
*s = '$';
exit(0);
}
```

Question **20**

Partially correct

Mark 0.25 out of 0.50

Map the functionality/use with function/variable in xv6 code.

return a free page, if available; 0, otherwise	kinit1() ✖
Create page table entries for a given range of virtual and physical addresses; including page directory entries if needed	mappages() ✔
Array listing the kernel memory mappings, to be used by setupkvm()	kmap[] ✔
Setup kernel part of a page table, mapping kernel code, data, read-only data, I/O space, devices	kvmalloc() ✖
Return address of page table entry in a given page directory, for a given virtual address; creates page table if necessary	walkpgdir() ✔
Setup kernel part of a page table, and switch to that page table	setupkvm() ✖

Your answer is partially correct.

You have correctly selected 3.

The correct answer is: return a free page, if available; 0, otherwise → kalloc(), Create page table entries for a given range of virtual and physical addresses; including page directory entries if needed → mappages(), Array listing the kernel memory mappings, to be used by setupkvm() → kmap[], Setup kernel part of a page table, mapping kernel code, data, read-only data, I/O space, devices → setupkvm(), Return address of page table entry in a given page directory, for a given virtual address; creates page table if necessary → walkpgdir(), Setup kernel part of a page table, and switch to that page table → kvmalloc()

Question **21**

Partially correct

Mark 1.53 out of 2.50

Order events in xv6 timer interrupt code

(Transition from process P1 to P2's code.)

P2 is selected and marked RUNNING	12 ✔
Change of stack from user stack to kernel stack of P1	3 ✔
Timer interrupt occurs	2 ✔
alltraps() will call iret	17 ✖
change to context of P2, P2's kernel stack in use now	13 ✔
P2's trap() will return to alltraps	16 ✖
jump in vector.S	4 ✔
P2 will return from sched() in yield()	14 ✖
yield() is called	8 ✔
trap() is called	7 ✔
Process P2 is executing	18 ✖
P1 is marked as RUNNABLE	9 ✔
P2's yield() will return in trap()	15 ✖
Process P1 is executing	1 ✔
sched() is called,	11 ✖
change to context of the scheduler, scheduler's stack in use now	10 ✖
jump to alltraps	5 ✔
Trapframe is built on kernel stack of P1	6 ✔

Your answer is partially correct.

You have correctly selected 11.

The correct answer is: P2 is selected and marked RUNNING → 12, Change of stack from user stack to kernel stack of P1 → 3, Timer interrupt occurs → 2, alltraps() will call iret → 18, change to context of P2, P2's kernel stack in use now → 13, P2's trap() will return to alltraps → 17, jump in vector.S → 4, P2 will return from sched() in yield() → 15, yield() is called → 8, trap() is called → 7, Process P2 is executing → 14, P1 is marked as RUNNABLE → 9, P2's yield() will return in trap() → 16, Process P1 is executing → 1, sched() is called, → 10, change to context of the scheduler, scheduler's stack in use now → 11, jump to alltraps → 5, Trapframe is built on kernel stack of P1 → 6

Question **22**

Incorrect

Mark 0.00 out of 1.00

Given that the memory access time is 200 ns, probability of a page fault is 0.7 and page fault handling time is 8 ms,
The effective memory access time in nanoseconds is:

Answer: 192 ✖

The correct answer is: 5600060.00

Question **23**

Correct

Mark 0.25 out of 0.25

Select the state that is not possible after the given state, for a process:

New: Running ✔

Ready : Waiting ✔

Running: : None of these ✔

Waiting: Running ✔

Question **24**

Partially correct

Mark 0.63 out of 1.00

Select the correct statements about sched() and scheduler() in xv6 code

- [x] a. scheduler() switches to the selected process's context ✔
- [x] b. When either sched() or scheduler() is called, it does not return immediately to caller ✔
- [ ] c. After call to swtch() in sched(), the control moves to code in scheduler()
- [x] d. Each call to sched() or scheduler() involves change of one stack inside swtch() ✔
- [ ] e. After call to swtch() in scheduler(), the control moves to code in sched()
- [x] f. When either sched() or scheduler() is called, it results in a context switch ✔
- [x] g. sched() switches to the scheduler's context ✔
- [ ] h. sched() and scheduler() are co-routines

Your answer is partially correct.

You have correctly selected 5.
The correct answers are: sched() and scheduler() are co-routines, When either sched() or scheduler() is called, it does not return immediately to caller, When either sched() or scheduler() is called, it results in a context switch, sched() switches to the scheduler's context, scheduler() switches to the selected process's context, After call to swtch() in scheduler(), the control moves to code in sched(), After call to swtch() in sched(), the control moves to code in scheduler(), Each call to sched() or scheduler() involves change of one stack inside swtch()

Question **25**

Correct

Mark 0.25 out of 0.25

The data structure used in kalloc() and kfree() in xv6 is

- a. Doubly linked circular list
- b. Singly linked circular list
- c. Double linked NULL terminated list
- d. Singly linked NULL terminated list ✔

Your answer is correct.

The correct answer is: Singly linked NULL terminated list

◄ (Assignment) lseek system call in xv6

Jump to...

Started on	Saturday, 22 May 2021, 8:00 AM
State	Finished
Completed on	Saturday, 22 May 2021, 9:30 AM
Time taken	1 hour 30 mins
Grade	**26.12** out of 40.00 (**65**%)

Question **1**

Incorrect

Mark 0.00 out of 1.00

A 4 GB disk with 1 KB of block size would require these many number of **blocks** for it's free block bitmap:

Answer: 4096 ✖

The correct answer is: 512

Question **2**

Correct

Mark 1.00 out of 1.00

Given that the memory access time is 110 ns, probability of a page fault is 0.5 and page fault handling time is 12 ms,
The effective memory access time in nanoseconds is:

Answer: 6000165 ✔

The correct answer is: 6000055.00

Question **3**

Incorrect

Mark 0.00 out of 1.00

The maximum size of a file in number of blocks of BSIZE in xv6 code is

(write a number only)

Answer: 268 ✖

The correct answer is: 138

Question **4**

Incorrect

Mark 0.00 out of 1.00

Calculate the average waiting time using
Round Robin scheduling with time quantum of 5 time units
for the following workload

assuming that they arrive in the order written below.

Process Burst Time
P1 5
P2 7
P3 6
P4 2

Write only a number in the answer upto two decimal points.

Answer: 40.75 ✖

The correct answer is: 10.25

Question **5**

Correct

Mark 1.00 out of 1.00

For the reference string

4 2 5 1 0 1 2 5 4 1 2

the number of page faults, including initial ones,

with FIFO replacement and 2 frames are :

Answer: 10 ✔

4 -

4 2

5 2

5 1

0 1

-

2 1

2 5

4 5

4 1

2 1

The correct answer is: 10

Question **6**

Correct

Mark 1.00 out of 1.00

Assuming a 16- KB page size, what is the page number for the address 428517 reference in decimal :

(give answer also in decimal)

Answer: 27 ✔

The correct answer is: 26

Question **7**

Correct

Mark 1.00 out of 1.00

In the code below assume that each function can be executed concurrently by many threads/processes.
Ignore syntactical issues, and focus on the semantics.

This program is an example of

```
spinlock a, b;  // assume initialized
thread1() {
    spinlock(b);
    //some code;
    spinlock(a);
    //some code;
    spinunlock(b);
    spinunlock(a);
}
thread2() {
    spinlock(a);
    //some code;
    spinlock(b);
    //some code;
    spinunlock(b);
    spinunlock(a);
}
```

- a. Deadlock ✔
- b. Self Deadlock
- c. None of these
- d. Deadlock or livelock depending on actual race
- e. Livelock

Your answer is correct.

The correct answer is: Deadlock

Question **8**

Partially correct

Mark 1.33 out of 2.00

Match the snippets of xv6 code with the core functionality they achieve, or problems they avoid.
"..." means some code.

```
static inline uint
xchg(volatile uint *addr, uint newval)
{
  uint result;

  // The + in "+m" denotes a read-modify-write operand.
  asm volatile("lock; xchgl %0, %1" :
               "+m" (*addr), "=a" (result) :
               "1" (newval) :
               "cc");
  return result;
}
```

Atomic compare and swap instruction (to be expanded inline into code) ✔

```
void
sleep(void *chan, struct spinlock *lk)
{
  ...
  if(lk != &ptable.lock){
    acquire(&ptable.lock);
    release(lk);
  }
```

If you don't do this, a process may be running on two processors parallely ✖

```
void
acquire(struct spinlock *lk)
{
...
__sync_synchronize();
```

Tell compiler not to reorder memory access beyond this line ✔

Your answer is partially correct.

You have correctly selected 2.

The correct answer is: static inline uint
xchg(volatile uint *addr, uint newval)
{
  uint result;

  // The + in "+m" denotes a read-modify-write operand.
  asm volatile("lock; xchgl %0, %1" :
               "+m" (*addr), "=a" (result) :
               "1" (newval) :
               "cc");
  return result;
} → Atomic compare and swap instruction (to be expanded inline into code), void
sleep(void *chan, struct spinlock *lk)
{
  ...
  if(lk != &ptable.lock){
    acquire(&ptable.lock);
    release(lk);
  } → Avoid a self-deadlock, void
acquire(struct spinlock *lk)
{
...
__sync_synchronize(); → Tell compiler not to reorder memory access beyond this line

Question **9**

Correct

Mark 1.00 out of 1.00

Predict the output of the program given here.

Assume that all the path names for the programs are correct. For example "/usr/bin/echo" will actually run echo command.

Assume that there is no mixing of printf output on screen if two of them run concurrently.

In the answer replace a new line by a single space.

For example::

`good`

`output`

should be written as `good output`

`--`

```
main() {
    int i;
    i = fork();
    if(i == 0)
        execl("/usr/bin/echo", "/usr/bin/echo", "hi", 0);
    else
        wait(0);
    fork();
    execl("/usr/bin/echo", "/usr/bin/echo", "one", 0);
}
```

Answer: hi one one ✔

The correct answer is: hi one one

Question **10**

Partially correct

Mark 1.67 out of 2.00

Select all the blocks that may need to be written back to disk (if updated, of-course), as "Yes", when an operation of deleting a file is carried out on ext2 file system.

An option has to be correct entirely to be marked "Yes"

Superblock	Yes ✔
One or multiple data blocks of the parent directory	No ✔
One or more data bitmap blocks for the parent directory	No ✔
Block bitmap(s) for all the blocks of the file	No ✖
Possibly one block bitmap corresponding to the parent directory	Yes ✔
Data blocks of the file	No ✔

Your answer is partially correct.

only one data block of parent directory. multiple blocks not possible. an entry is always contained within one single block

You have correctly selected 5.

The correct answer is: Superblock → Yes, One or multiple data blocks of the parent directory → No, One or more data bitmap blocks for the parent directory → No, Block bitmap(s) for all the blocks of the file → Yes, Possibly one block bitmap corresponding to the parent directory → Yes, Data blocks of the file → No

Question **11**

Correct

Mark 1.00 out of 1.00

Select all the correct statements about bootloader.

Every wrong selection will deduct marks proportional to 1/n where n is total wrong choices in the question.

You will get minimum a zero.

- [x] a. Modern Bootloaders often allow configuring the way an OS boots ✔
- [x] b. Bootloaders allow selection of OS to boot from ✔
- [ ] c. Bootloader must be one sector in length
- [ ] d. The bootloader loads the BIOS
- [x] e. LILO is a bootloader ✔

Your answer is correct.

The correct answers are: LILO is a bootloader, Modern Bootloaders often allow configuring the way an OS boots, Bootloaders allow selection of OS to boot from

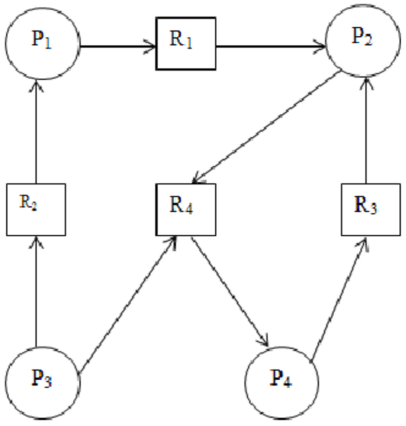
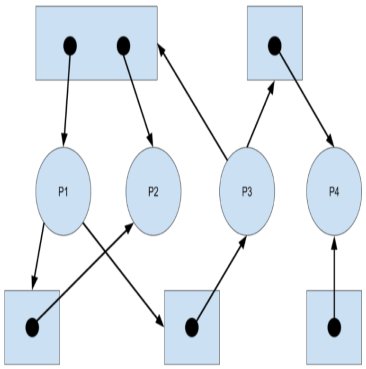
Question **12**

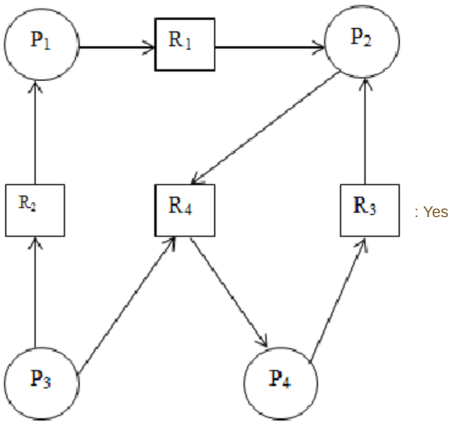
Incorrect

Mark 0.00 out of 1.00

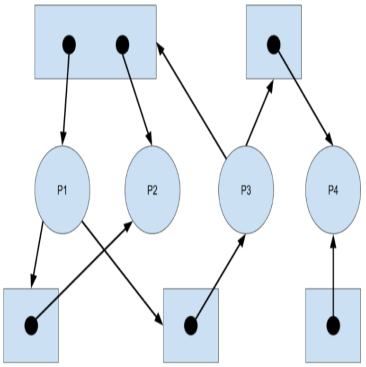
For each of the resource allocation diagram shown,

infer whether the graph contains at least one deadlock or not.

Yes	No	
		
		



: Yes



: No

Question **13**

Partially correct

Mark 0.71 out of 1.00

Mark the statements about device drivers by marking as True or False.

True	False		
○	●	It's possible that a particular hardware has multiple device drivers available for it.	✖
●	○	xv6 has device drivers for IDE disk and console.	✔
●	○	A disk driver converts OS's logical view of disk into physical locations on disk.	✔
●	○	A device driver code is specific to a hardware device	✔
●	○	All devices of the same type (e.g. 2 hard disks) can typically use the same device driver	✔
○	●	Writing a device driver mandatorily demands reading the technical documentation about the hardware.	✖
○	●	Device driver is an intermediary between the end-user and OS	✔

It's possible that a particular hardware has multiple device drivers available for it.: True
xv6 has device drivers for IDE disk and console.: True
A disk driver converts OS's logical view of disk into physical locations on disk.: True
A device driver code is specific to a hardware device: True
All devices of the same type (e.g. 2 hard disks) can typically use the same device driver: True
Writing a device driver mandatorily demands reading the technical documentation about the hardware.: True
Device driver is an intermediary between the end-user and OS: False

Question **14**

Partially correct

Mark 0.33 out of 1.00

Consider this program.
Some statements are identified using the // comment at the end.

Assume that = is an atomic operation.

```
#include <stdio.h>
#include <pthread.h>
long c = 0, c1 = 0, c2 = 0, run = 1;
void *thread1(void *arg) {
    while(run == 1) {//E
        c = 10; //A
        c1 = c2 + 5; //B
    }
}
void *thread2(void *arg) {
    while(run == 1) {//F
        c = 20;//C
        c2 = c1 + 3;//D
    }
}
int main() {
    pthread_t th1, th2;
    pthread_create(&th1, NULL, thread1, NULL);
    pthread_create(&th2, NULL, thread2, NULL);
    sleep(2);
    run = 0;
    fprintf(stdout, "c = %ld c1+c2 = %ld c1 = %ld c2 = %ld \n", c, c1+c2, c1, c2);
    fflush(stdout);
}
```

Which statements are part of the critical Section?

Yes	No		
○	✓	F	✗
✓	○	D	✓
○	✓	C	✗
○	✓	A	✗
✓	○	B	✓
○	✓	E	✗

F: No
D: Yes
C: No
A: No
B: Yes
E: No

Question **15**

Partially correct

Mark 1.43 out of 2.00

Mark statements as T/F

All statements are in the context of preventing deadlocks.

True	False		
◉	○	A process holding one resources and waiting for just one more resource can also be involved in a deadlock.	✔
◉	○	If a resource allocation graph contains a cycle then there is a guarantee of a deadlock	✖
◉	○	The lock ordering to be followed to avoid circular wait is a code in OS that checks for compliance with decided order	✖
◉	○	Circular wait is avoided by enforcing a lock ordering	✔
◉	○	Hold and wait means a thread/process holding some locks and waiting for acquiring some.	✔
◉	○	Deadlock is possible if all the conditions are met at the same time: Mutual exclusion, hold and wait, no pre-emption, circular wait.	✔
◉	○	Mutual exclusion is a necessary condition for deadlock because it brings in locks on which deadlock happens	✔

A process holding one resources and waiting for just one more resource can also be involved in a deadlock.: True
If a resource allocation graph contains a cycle then there is a guarantee of a deadlock: False
The lock ordering to be followed to avoid circular wait is a code in OS that checks for compliance with decided order: False
Circular wait is avoided by enforcing a lock ordering: True
Hold and wait means a thread/process holding some locks and waiting for acquiring some.: True
Deadlock is possible if all the conditions are met at the same time: Mutual exclusion, hold and wait, no pre-emption, circular wait.: True
Mutual exclusion is a necessary condition for deadlock because it brings in locks on which deadlock happens: True

Question **16**

Correct

Mark 1.00 out of 1.00

Match the left side use(or non-use) of a synchronization primitive with the best option on the right side.

This is the smallest primitive made available in software, using the hardware provided atomic instructions	spinlock ✔
This tool is useful for event-wait scenarios	semaphore ✔
This tool is more useful on multiprocessor systems	spinlock ✔
This tool is quite attractive in solving the main bounded buffer problem	semaphore ✔
This tool is very useful for waiting for 'something'	condition variables ✔

Your answer is correct.

The correct answer is: This is the smallest primitive made available in software, using the hardware provided atomic instructions → spinlock, This tool is useful for event-wait scenarios → semaphore, This tool is more useful on multiprocessor systems → spinlock, This tool is quite attractive in solving the main bounded buffer problem → semaphore, This tool is very useful for waiting for 'something' → condition variables

Question **17**

Correct

Mark 1.00 out of 1.00

The permissions -rwx--x--x on a file mean

- [x] a. The file can be read only by the owner ✔
- [ ] b. 'cat' on the file by owner will not work
- [ ] c. 'cat' on the file by any user will work
- [x] d. 'rm' on the file by any user will work ✔
- [x] e. The file can be executed by anyone ✔
- [x] f. The file can be written only by the owner ✔

Your answer is correct.

The correct answers are: The file can be executed by anyone, The file can be read only by the owner, The file can be written only by the owner, 'rm' on the file by any user will work

Question **18**

Incorrect

Mark 0.00 out of 1.00

Note: for this question you get full marks if you select all and only correct options, you get ZERO if at least one option is wrong or not selected.

Select all the correct statements about log structured file systems.

- [x] a. a transaction is said to be committed when all operations are written to file system ✖
- [x] b. log may be kept on same block device or another block device ✔
- [ ] c. file system recovery may end up losing data
- [x] d. even if file systems followed immediate writes (i.e. non-delayed writes), it could still require recovery ✔
- [x] e. file system recovery recovers all the lost data ✖

Your answer is incorrect.

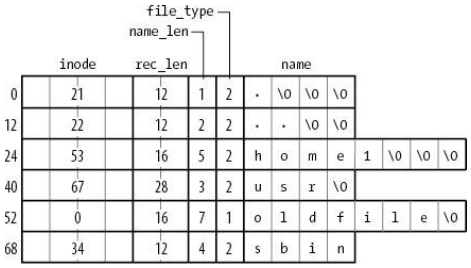
The correct answers are: file system recovery may end up losing data, log may be kept on same block device or another block device, even if file systems followed immediate writes (i.e. non-delayed writes), it could still require recovery

Question **19**

Incorrect

Mark 0.00 out of 1.00

Consider the structure of directory entry in ext2, as shown in this diagram.



Select the correct statements about the directory entry in ext2 file system.

The correct formula for rec_len is (when entries are continuously stored)

- a. rec_len = sizeof(inode entry) + sizeof(name len entry) + sizeof(file type entry) + (strlen(name) + (-1) * (strlen(name)%4)
- b. rec_len = sizeof(inode entry) + sizeof(name len entry) + sizeof(file type entry) + (strlen(name) + (strlen(name)- 4) %4
- c. rec_len = sizeof(inode entry) + sizeof(name len entry) + sizeof(file type entry) + (strlen(name) + 4 - (strlen(name)%4) ✖
- d. rec_len = sizeof(inode entry) + sizeof(name len entry) + sizeof(file type entry) + (strlen(name) + (-1) * (strlen(name)- 4)
- e. rec_len = sizeof(inode entry) + sizeof(name len entry) + sizeof(file type entry) + (strlen(name) %4
- f. rec_len = sizeof(inode entry) + sizeof(name len entry) + sizeof(file type entry) + strlen(name)

Your answer is incorrect.

The correct answer is: rec_len = sizeof(inode entry) + sizeof(name len entry) + sizeof(file type entry) + (strlen(name) + (-1) * (strlen(name)- 4)

Question **20**

Partially correct

Mark 0.50 out of 1.00

Mark whether the given sequence of events is possible or not-possible. Also, select the reason for your answer.

For each sequence it's a not-possible sequence if some important event is not mentioned in the sequence.

Assume that the kernel code is non-interruptible and uniprocessor system.

```
Process P1 executing a system call
Timer interrupt
Generic interrupt handler runs
Scheduler runs
Scheduler selects P2 for execution
P2 returns from timer interrupt handler
Process p2, user code executing
```

This sequence of events is: not-possible ✔

Because

Interrupts are not possible when kernel code is running ✖

Question **21**

Incorrect

Mark 0.00 out of 1.00

The given semaphore implementation faces which problem?

Assume any suitable code for signal()

Note: blocks means waits in a wait queue.

```
struct semaphore {
        int val;
        spinlock lk;
};
sem_init(semaphore *s, int initval) {
        s->val = initval;
        s->sl = 0;
}
wait(semaphore *s) {
        spinlock(&(s->sl));
        while(s->val <=0)
        ;
        (s->val)--;
        spinunlock(&(s->sl));
}
```

a. blocks holding a spinlock

b. deadlock

c. too much spinning, bounded wait not guaranteed ✖

d. not holding lock after unblock

Your answer is incorrect.

The correct answer is: deadlock

Question **22**

Partially correct

Mark 0.80 out of 1.00

Mark statements True/False w.r.t. change of states of a process.

Reference: The process state diagram (and your understanding of how kernel code works)

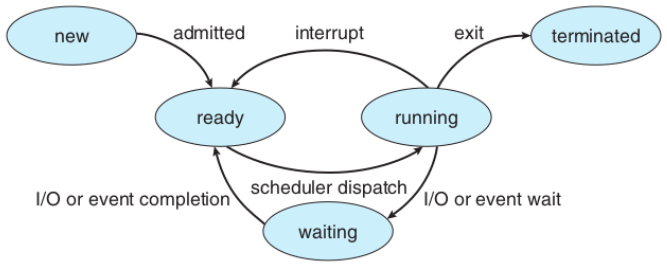


Figure 3.2 Diagram of process state.

True	False		
○	●	A process in RUNNING state only can become TERMINATED because scheduler moves it to ZOMBIE state	✔
●	○	A process in READY state can not go to WAITING state because the resource on which it will WAIT will not be in use when process is in READY state.	✖
●	○	A process in WAITING state can not become RUNNING because the event it's waiting for has not occurred	✔
●	○	Every process has to go through ZOMBIE state, at least for a small duration.	✔
●	○	Only a process in READY state is considered by scheduler	✔

A process in RUNNING state only can become TERMINATED because scheduler moves it to ZOMBIE state: False
A process in READY state can not go to WAITING state because the resource on which it will WAIT will not be in use when process is in READY state.: False
A process in WAITING state can not become RUNNING because the event it's waiting for has not occurred: True
Every process has to go through ZOMBIE state, at least for a small duration.: True
Only a process in READY state is considered by scheduler: True

Question **23**

Correct

Mark 1.00 out of 1.00

Select T/F for statements about Volume Managers.
Do pay attention to the use of the words physical partition and physical volume.

True	False			
◉	○	The volume manager can create further internal sub-divisions of a physical partition for efficiency or features.	✔	
◉	○	A logical volume can be extended in size but upto the size of volume group	✔	
◉	○	A logical volume may span across multiple physical volumes	✔	
◉	○	The volume manager stores additional metadata on the physical disk partitions	✔	
◉	○	A physical partition should be initialized as a physial volume, before it can be used by volume manager.	✔	
◉	○	A volume group consists of multiple physical volumes	✔	
◉	○	A logical volume may span across multiple physical partitions	✔	since a physical volume is made up of physical partitions, and a volume can span across multiple PVs, it can also span across multiple PP

The volume manager can create further internal sub-divisions of a physical partition for efficiency or features.: True
A logical volume can be extended in size but upto the size of volume group: True
A logical volume may span across multiple physical volumes: True
The volume manager stores additional metadata on the physical disk partitions: True
A physical partition should be initialized as a physial volume, before it can be used by volume manager.: True
A volume group consists of multiple physical volumes: True
A logical volume may span across multiple physical partitions: True

Question **24**

Correct

Mark 1.00 out of 1.00

Map the block allocation scheme with the problem it suffers from

(Match pairs 1-1, match a scheme with the problem that it suffers from relatively the most, compared to others)

Continuous allocation	need for compaction	✔
Linked allocation	Too many seeks	✔
Indexed Allocation	Overhead of reading metadata blocks	✔

Your answer is correct.

The correct answer is: Continuous allocation → need for compaction, Linked allocation → Too many seeks, Indexed Allocation → Overhead of reading metadata blocks

Question **25**

Correct

Mark 1.00 out of 1.00

This one is not a system call:

- ○ a. open
- ○ b. read
- ○ c. write
- ◉ d. scheduler ✔

Your answer is correct.

The correct answer is: scheduler

Question **26**

Correct

Mark 1.00 out of 1.00

Match the pairs.

This question is based on your general knowledge about operating systems/related concepts and their features.

Java thereads	monitors,re-entrant locks, semaphores ✔
Linux threads	atomic-instructions, spinlocks, etc. ✔
POSIX threads	semaphore, mutex, condition variables ✔

Your answer is correct.

The correct answer is: Java thereads → monitors,re-entrant locks, semaphores, Linux threads → atomic-instructions, spinlocks, etc., POSIX threads → semaphore, mutex, condition variables

Question **27**

Correct

Mark 1.00 out of 1.00

Consider the following list of free chunks, in continuous memory management:

7k, 15k, 21k, 14k, 19k, 6k

Suppose there is a request for chunk of size 5k, then the free chunk selected under each of the following schemes will be

Best fit: 6k ✔

First fit: 7k ✔

Worst fit: 21k ✔

Question **28**

Correct

Mark 1.00 out of 1.00

This one is not a scheduling algorithm

- a. Round Robin
- b. SJF
- c. Mergesort ✔
- d. FCFS

Your answer is correct.

The correct answer is: Mergesort

Question **29**

Correct

Mark 1.00 out of 1.00

Mark whether the concept is related to scheduling or not.

Yes	No		
◉	○	timer interrupt	✔
◉	○	context-switch	✔
◉	○	ready-queue	✔
○	◉	file-table	✔
◉	○	runnable process	✔

timer interrupt: Yes
context-switch: Yes
ready-queue: Yes
file-table: No
runnable process: Yes

Question **30**

Partially correct

Mark 1.00 out of 2.00

Map ext2 data structure features with their purpose

Many copies of Superblock	Choose...
Free blocks count in superblock and group descriptor ✖	Redundancy to ensure the most crucial data structure is not lost
Used directories count in group descriptor ✖	is redundant and helps do calculations of directory entries faster
Combining file type and access rights in one variable ✔	saves 1 byte of space
rec_len field in directory entry ✖	Try to keep all the data of a directory and it's file close together in a group
File Name is padded ✔	aligns all memory accesses on word boundary, improving performance
Inode bitmap is one block ✔	limits total number of files that can belong to a group
Block bitmap is one block ✔	Limits the size of a block group, thus improvising on purpose of a group
Mount count in superblock ✔	to enforce file check after certain amount of mounts at boot time
Inode table location in Group Descriptor ✖	is redundant and helps do calculations of directory entries faster
Inode table ✔	All inodes are kept together so that one disk read leads to reading many inodes together, effectively doing a buffering of subsequent inode reads, and to save space on disk
A group ✖	Redundancy to ensure the most crucial data structure is not lost

Your answer is partially correct.

You have correctly selected 6.

The correct answer is: Many copies of Superblock → Redundancy to ensure the most crucial data structure is not lost, Free blocks count in superblock and group descriptor → Redundancy to help fsck restore consistency, Used directories count in group descriptor → attempt is made to evenly spread the first-level directories, this count is used there, Combining file type and access rights in one variable → saves 1 byte of space, rec_len field in directory entry → allows holes and linking of entries in directory, File Name is padded → aligns all memory accesses on word boundary, improving performance, Inode bitmap is one block → limits total number of files that can belong to a group, Block bitmap is one block → Limits the size of a block group, thus improvising on purpose of a group, Mount count in superblock → to enforce file check after certain amount of mounts at boot time, Inode table location in Group Descriptor → Obvious, as it's per group and not per file-system, Inode table → All inodes are kept together so that one disk read leads to reading many inodes together, effectively doing a buffering of subsequent inode reads, and to save space on disk, A group → Try to keep all the data of a directory and it's file close together in a group

Question **31**

Partially correct

Mark 1.85 out of 2.00

Mark True/False

Statements about scheduling and scheduling algorithms

True	False		
◉	○	The nice() system call is used to set priorities for processes	✔
◉	○	Aging is used to ensure that low-priority processes do not starve in priority scheduling.	✔
○	◉	In non-pre-emptive priority scheduling, the highest priority process is scheduled and runs until it gives up CPU.	✖
◉	○	xv6 code does not care about Processor Affinity	✔
◉	○	In pre-emptive priority scheduling, priority is implemented by assigning more time quantum to higher priority process.	✔
◉	○	A scheduling algorithm is non-premptive if it does context switch only if a process voluntarily relinquishes CPU or it terminates.	✔
◉	○	Processor Affinity refers to memory accesses of a process being stored on cache of that processor	✔
◉	○	Response time will be quite poor on non-interruptible kernels	✔
◉	○	Shortest Remaining Time First algorithm is nothing but pre-emptive Shortest Job First algorithm	✔
◉	○	On Linuxes the CPU utilisation is measured as the time spent in scheduling the idle thread	✔
◉	○	Generally the voluntary context switches are much more than non-voluntary context switches on a Linux system.	✔
◉	○	Pre-emptive scheduling leads to many race conditions in kernel code.	✔
◉	○	Statistical observations tell us that most processes have large number of small CPU bursts and relatively smaller numbers of large CPU bursts.	✔

The nice() system call is used to set priorities for processes: True
Aging is used to ensure that low-priority processes do not starve in priority scheduling.: True
In non-pre-emptive priority scheduling, the highest priority process is scheduled and runs until it gives up CPU.: True
xv6 code does not care about Processor Affinity: True
In pre-emptive priority scheduling, priority is implemented by assigning more time quantum to higher priority process.: True
A scheduling algorithm is non-premptive if it does context switch only if a process voluntarily relinquishes CPU or it terminates.: True
Processor Affinity refers to memory accesses of a process being stored on cache of that processor: True
Response time will be quite poor on non-interruptible kernels: True
Shortest Remaining Time First algorithm is nothing but pre-emptive Shortest Job First algorithm: True
On Linuxes the CPU utilisation is measured as the time spent in scheduling the idle thread: True
Generally the voluntary context switches are much more than non-voluntary context switches on a Linux system.: True
Pre-emptive scheduling leads to many race conditions in kernel code.: True
Statistical observations tell us that most processes have large number of small CPU bursts and relatively smaller numbers of large CPU bursts.: True

Question **32**

Partially correct

Mark 1.17 out of 2.00

The unix file semantics demand that changes to any open file are visible immediately to any other processes accessing that file at that point in time.

Select the data-structure/programmatic features that ensure the implementation of unix semantics. (Assume there is no mmap())

Yes	No		
○ ✗	◉ ☑	All processes accessing the same file share the file descriptor among themselves	✔
○ ✗	◉ ☑	The pointer entry in the file descriptor array entry points to the data of the file directly	✔
◉ ✗	○ ☑	There is only one global file structure per on-disk file.	✖
○ ✗	◉ ☑	All file accesses are made using only global variables	✔
◉ ✗	○ ☑	The 'file offset' is shared among all the processes that access the file.	✖
○ ✗	◉ ☑	No synchronization is implemented so that changes are made available immediately.	✔
◉ ✗	○ ☑	A single spinlock is to be used to protect the unique global 'file structure' representing the file, thus synchronizing access, and making other processes wait for earlier process to finish writing so that writes get visible immediately.	✖
◉ ☑	○ ✗	There is only one in-memory copy of the on disk file's contents in kernel memory/buffers	✔
◉ ✗	○ ☑	The file descriptors in every PCB are pointers to the same global file structure.	✖
○ ✗	◉ ☑	The file descriptor array is external to PCB and all processes that share a file, have pointers to same file-descriptors' array	✔
◉ ☑	○ ✗	All file structures representing any open file, give access to the same in-memory copy of the file's contents	✔
◉ ✗	○ ☑	The 'file offset' index is stored outside the file-structure to which file-descriptor array points	✖

All processes accessing the same file share the file descriptor among themselves: No
The pointer entry in the file descriptor array entry points to the data of the file directly: No
There is only one global file structure per on-disk file.: No
All file accesses are made using only global variables: No
The 'file offset' is shared among all the processes that access the file.: No
No synchronization is implemented so that changes are made available immediately.: No
A single spinlock is to be used to protect the unique global 'file structure' representing the file, thus synchronizing access, and making other processes wait for earlier process to finish writing so that writes get visible immediately.: No
There is only one in-memory copy of the on disk file's contents in kernel memory/buffers: Yes
The file descriptors in every PCB are pointers to the same global file structure.: No
The file descriptor array is external to PCB and all processes that share a file, have pointers to same file-descriptors' array: No
All file structures representing any open file, give access to the same in-memory copy of the file's contents: Yes
The 'file offset' index is stored outside the file-structure to which file-descriptor array points: No

Question **33**

Partially correct

Mark 0.33 out of 2.00

Map the function in xv6's file system code, to it's perceived logical layer.

namei	inode	✗
filestat()	Choose...	
dirlookup	directory	✓
ialloc	file descriptor	✗
stati	Choose...	
ideintr	buffer cache	✗
bread	Choose...	
balloc	file descriptor	✗
sys_chdir()	system call	✓
skipelem	system call	✗
commit	system call	✗
bmap	system call	✗

Your answer is partially correct.

You have correctly selected 2.
The correct answer is: namei → pathname lookup, filestat() → file descriptor, dirlookup → directory, ialloc → inode, stati → inode, ideintr → disk driver, bread → buffer cache, balloc → block allocation on disk, sys_chdir() → system call, skipelem → pathname lookup, commit → logging, bmap → inode

◄ Course Exit Feedback

Jump to...

xv6-public-master ►

Started on	Monday, 24 January 2022, 7:07:42 PM
State	Finished
Completed on	Monday, 24 January 2022, 8:08:11 PM
Time taken	1 hour
Grade	**8.90** out of 20.00 (**45**%)

Question **1**

Complete

Mark 0.80 out of 1.00

Match the register with the segment used with it.

ebp	ss
eip	cs
edi	ds
esp	ss
esi	ds

The correct answer is: ebp → ss, eip → cs, edi → es, esp → ss, esi → ds

Question **2**

Complete

Mark 1.00 out of 1.00

```
int value = 5;
int main()
{
   pid_t pid;
   pid = fork();
   if (pid == 0) { /* child process */
      value += 15;
      return 0;
   }
   else if (pid > 0) { /* parent process */
      wait(NULL);
      printf("%d", value); /* LINE A */
   }
   return 0;
}
```

What's the value printed here at LINE A?

Answer: 5

The correct answer is: 5

Question **3**

Complete

Mark 0.50 out of 0.50

Is the command "cat README > done &" possible on xv6? (Note the & in the end)

- a. no
- b. yes

The correct answer is: yes

Question **4**

Complete

Mark 0.00 out of 2.00

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?

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

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)

Question **5**

Complete

Mark 0.43 out of 1.00

Rank the following storage systems from slowest (first) to fastest(last)

You can drag and drop the items below/above each other.

- Registers
- Cache
- Main memory
- Nonvolatile memory
- Magnetic tapes
- Optical disk
- Hard-disk drives

The correct order for these items is as follows:

1. Magnetic tapes
2. Optical disk
3. Hard-disk drives
4. Nonvolatile memory
5. Main memory
6. Cache
7. Registers

Question **6**

Complete

Mark 1.00 out of 1.00

How does the distinction between kernel mode and user mode function as a rudimentary form of protection (security) ?

Select one:

- a. It disallows hardware interrupts when a process is running
- b. It prohibits one process from accessing other process's memory
- c. It prohibits a user mode process from running privileged instructions
- d. It prohibits invocation of kernel code completely, if a user program is running

The correct answer is: It prohibits a user mode process from running privileged instructions

Question **7**

Complete

Mark 0.00 out of 2.00

Which of the following are NOT a part of job of a typical compiler?

- [ ] a. Suggest alternative pieces of code that can be written
- [x] b. Check the program for syntactical errors
- [ ] c. Check the program for logical errors
- [x] d. Convert high level langauge code to machine code
- [x] e. Process the # directives in a C program
- [ ] f. Invoke the linker to link the function calls with their code, extern globals with their declaration

The correct answers are: Check the program for logical errors, Suggest alternative pieces of code that can be written

Question **8**

Complete

Mark 0.00 out of 2.00

Match the program with it's output (ignore newlines in the output. Just focus on the count of the number of 'hi')

`main() { execl("/usr/bin/echo", "/usr/bin/echo", "hi\n", NULL); }`	hi hi
`main() { int i = fork(); if(i == 0) execl("/usr/bin/echo", "/usr/bin/echo", "hi\n", NULL); }`	hi hi hi hi
`main() { int i = NULL; fork(); printf("hi\n"); }`	No output
`main() { fork(); execl("/usr/bin/echo", "/usr/bin/echo", "hi\n", NULL); }`	hi

The correct answer is: `main() { execl("/usr/bin/echo", "/usr/bin/echo", "hi\n", NULL); }` → hi, `main() { int i = fork(); if(i == 0) execl("/usr/bin/echo", "/usr/bin/echo", "hi\n", NULL); }` → hi, `main() { int i = NULL; fork(); printf("hi\n"); }` → hi hi, `main() { fork(); execl("/usr/bin/echo", "/usr/bin/echo", "hi\n", NULL); }` → hi hi

Question **9**

Complete

Mark 0.83 out of 2.00

Select all statements that correctly explain the use/purpose of system calls.

Select one or more:

- [x] a. Provide an environment for process creation
- [ ] b. Switch from user mode to kernel mode
- [ ] c. Handle ALL types of interrupts
- [ ] d. Handle exceptions like division by zero
- [x] e. Run each instruction of an application program
- [x] f. Allow I/O device access to user processes
- [x] g. Provide services for accessing files

The correct answers are: Switch from user mode to kernel mode, Provide services for accessing files, Allow I/O device access to user processes, Provide an environment for process creation

Question **10**

Complete

Mark 0.50 out of 0.50

Compare multiprogramming with multitasking

- (•) a. A multiprogramming system is not necessarily multitasking
- ( ) b. A multitasking system is not necessarily multiprogramming

The correct answer is: A multiprogramming system is not necessarily multitasking

Question **11**

Complete

Mark 0.60 out of 1.00

Select all the correct statements about two modes of CPU operation

Select one or more:

- [x] a. The two modes are essential for a multitasking system
- [ ] b. Some instructions are allowed to run only in user mode, while all instructions can run in kernel mode
- [x] c. The software interrupt instructions change the mode from user mode to kernel mode and jumps to predefined location simultaneously
- [ ] d. The two modes are essential for a multiprogramming system
- [x] e. There is an instruction like 'iret' to return from kernel mode to user mode

The correct answers are: The two modes are essential for a multiprogramming system, The two modes are essential for a multitasking system, There is an instruction like 'iret' to return from kernel mode to user mode, The software interrupt instructions change the mode from user mode to kernel mode and jumps to predefined location simultaneously, Some instructions are allowed to run only in user mode, while all instructions can run in kernel mode

Question **12**

Complete

Mark 0.50 out of 0.50

Is the terminal a part of the kernel on GNU/Linux systems?

- (•) a. no
- ( ) b. yes

The correct answer is: no

Question **13**

Complete

Mark 1.00 out of 1.00

Why should a program exist in memory before it starts executing ?

- ( ) a. Because the hard disk is a slow medium
- (•) b. Becase the processor can run instructions and access data only from memory
- ( ) c. Because the variables of the program are stored in memory
- ( ) d. Because the memory is volatile

The correct answer is: Becase the processor can run instructions and access data only from memory

Question **14**

Complete

Mark 1.33 out of 2.00

Which of the following instructions should be privileged?

Select one or more:

- [ ] a. Read the clock.
- [x] b. Access memory management unit of the processor
- [ ] c. Access I/O device.
- [x] d. Turn off interrupts.
- [ ] e. Set value of a memory location
- [x] f. Set value of timer.
- [ ] g. Access a general purpose register
- [x] h. Modify entries in device-status table
- [ ] i. Switch from user to kernel mode.

The correct answers are: Set value of timer., Access memory management unit of the processor, Turn off interrupts., Modify entries in device-status table, Access I/O device., Switch from user to kernel mode.

Question **15**

Complete

Mark 0.07 out of 2.00

Select all the correct statements about calling convention on x86 32-bit.

- [ ] a. The ebp pointers saved on the stack constitute a chain of activation records
- [x] b. The return value is either stored on the stack or returned in the eax register
- [ ] c. The two lines in the beginning of each function, "push %ebp; mov %esp, %ebp", create space for local variables
- [x] d. Parameters may be passed in registers or on stack
- [ ] e. Return address is one location above the ebp
- [ ] f. Parameters may be passed in registers or on stack
- [x] g. Compiler may allocate more memory on stack than needed
- [ ] h. Space for local variables is allocated by substracting the stack pointer inside the code of the called function
- [ ] i. Paramters are pushed on the stack in left-right order
- [ ] j. during execution of a function, ebp is pointing to the old ebp
- [ ] k. Space for local variables is allocated by substracting the stack pointer inside the code of the caller function

The correct answers are: Compiler may allocate more memory on stack than needed, Parameters may be passed in registers or on stack, Parameters may be passed in registers or on stack, Return address is one location above the ebp, during execution of a function, ebp is pointing to the old ebp, Space for local variables is allocated by substracting the stack pointer inside the code of the called function, The ebp pointers saved on the stack constitute a chain of activation records

Question **16**

Complete

Mark 0.33 out of 0.50

Order the following events in boot process (from 1 onwards)

Shell	3
BIOS	1
Init	4
OS	6
Login interface	5
Boot loader	2

The correct answer is: Shell → 6, BIOS → 1, Init → 4, OS → 3, Login interface → 5, Boot loader → 2

◄ (Task) Compulsory xv6 task

Jump to...

(Optional Assignment) Shell Programming(Conformance tests) ►

Started on	Wednesday, 9 February 2022, 7:00:12 PM
State	Finished
Completed on	Wednesday, 9 February 2022, 7:46:38 PM
Time taken	46 mins 26 secs
Grade	**3.00** out of 11.00 (**27**%)

Question **1**

Complete

Mark 0.00 out of 1.00

The number of GDT entries setup during boot process of xv6 is

- a. 2
- b. 3
- c. 0
- d. 256
- e. 4
- f. 255

The correct answer is: 3

Question **2**

Complete

Mark 0.00 out of 1.00

x86 provides which of the following type of memory management options?

- a. segmentation and one level paging
- b. segmentation or one or two level paging
- c. segmentation and two level paging
- d. segmentation or paging
- e. segmentation and one or two level paging
- f. segmentation only

The correct answer is: segmentation and one or two level paging

Question **3**

Complete

Mark 0.00 out of 1.00

which of the following is not a difference between real mode and protected mode

- a. in real mode general purpose registers are 16 bit, in protected mode they are 32 bit
- b. in real mode the addressable memory is more than in protected mode
- c. in real mode the addressable memory is less than in protected mode
- d. in real mode the segment is multiplied by 16, in protected mode segment is used as index in GDT
- e. processor starts in real mode

The correct answer is: in real mode the addressable memory is more than in protected mode

Question **4**

Complete

Mark 0.00 out of 1.00

The kernel ELF file contains how many Program headers?

- a. 4
- b. 2
- c. 3
- d. 9
- e. 10

The correct answer is: 3

Question **5**

Not answered

Marked out of 0.50

code line, MMU setting: Match the line of xv6 code with the MMU setup employed

Answer:

The correct answer is: inb $0x64,%al

Question **6**

Complete

Mark 1.00 out of 1.00

The kernel is loaded at Physical Address

- a. 0x00100000
- b. 0x0010000
- c. 0x80100000
- d. 0x80000000

The correct answer is: 0x00100000

Question **7**

Complete

Mark 0.00 out of 1.00

Why is the code of entry() in Assembly and not in C?

- a. Because the symbol entry() is inside the ELF file
- b. Because the kernel code must begin in assembly
- c. Because it needs to setup paging
- d. There is no particular reason, it could also be in C

The correct answer is: Because it needs to setup paging

Question **8**

Complete

Mark 1.00 out of 1.00

The ljmp instruction in general does

- a. change the CS and EIP to 32 bit mode, and jumps to next line of code
- b. change the CS and EIP to 32 bit mode, and jumps to new value of EIP
- c. change the CS and EIP to 32 bit mode
- d. change the CS and EIP to 32 bit mode, and jumps to kernel code

The correct answer is: change the CS and EIP to 32 bit mode, and jumps to new value of EIP

Question **9**

Complete

Mark 0.00 out of 1.00

The variable $stack in entry.S is

- a. located at the value given by %esp as setup by bootmain()
- b. located at 0x7c00
- c. a memory region allocated as a part of entry.S
- d. located at less than 0x7c00
- e. located at 0

The correct answer is: a memory region allocated as a part of entry.S

Question **10**

Not answered

Marked out of 0.50

Match the pairs of which action is taken by whom

Answer:

The correct answer is: kernel

Question **11**

Complete

Mark 0.00 out of 1.00

ELF Magic number is

- a. 0xFFFFFFFFF
- b. 0
- c. 0xELFELFELF
- d. 0xELF
- e. 0x0x464CELF
- f. 0x464C457FL
- g. 0x464C457FU

The correct answer is: 0x464C457FU

Question **12**

Complete

Mark 1.00 out of 1.00

The right side of line of code "entry = (void(*)(void))(elf->entry)" means

- a. Convert the "entry" in ELF structure into void
- b. Get the "entry" in ELF structure and convert it into a function pointer accepting no arguments and returning nothing
- c. Get the "entry" in ELF structure and convert it into a function void pointer
- d. Get the "entry" in ELF structure and convert it into a void pointer

The correct answer is: Get the "entry" in ELF structure and convert it into a function pointer accepting no arguments and returning nothing

◄ Homework questions: Basics of MM, xv6 booting

Jump to...

(Code) Files, redirection, dup, (IPC)pipe ►

Started on	Monday, 21 February 2022, 7:00:28 PM
State	Finished
Completed on	Monday, 21 February 2022, 7:49:24 PM
Time taken	48 mins 56 secs
Grade	**8.30** out of 10.00 (**83**%)

Question **1**

Complete

Mark 0.80 out of 1.00

Match the elements of C program to their place in memory

Local Static variables	Data
Global variables	Data
Code of main()	Code
Function code	Code
Arguments	Stack
Malloced Memory	Heap
#include files	No Memory needed
#define MACROS	No memory needed
Local Variables	Stack
Global Static variables	Data

The correct answer is: Local Static variables → Data, Global variables → Data, Code of main() → Code, Function code → Code, Arguments → Stack, Malloced Memory → Heap, #include files → No memory needed, #define MACROS → No Memory needed, Local Variables → Stack, Global Static variables → Data

Question **2**

Complete

Mark 1.00 out of 1.00

What will be the output of this program

```
int main() {
  int fd;
  printf("%d ",   open("/etc/passwd", O_RDONLY));
  close(1);
  fd = printf("%d ", open("/etc/passwd", O_RDONLY));
  close(fd);
   fd = printf("%d ", open("/etc/passwd", O_RDONLY));
}
```

- a. 3 1 2
- b. 3 3 3
- c. 3 1 1
- d. 1 1 1
- e. 3 4 5
- f. 2 2 2

The correct answer is: 3 1 1

Question **3**

Complete

Mark 1.00 out of 1.00

Arrange in correct order, the files involved in execution of system call

vectors.S	2
trap.c	4
usys.S	1
trapasm.S	3

The correct answer is: vectors.S → 2, trap.c → 4, usys.S → 1, trapasm.S → 3

Question **4**

Complete

Mark 1.00 out of 1.00

The "push 0" in vectors.S is

- a. A placeholder to match the size of struct trapframe
- b. Place for the error number value
- c. To indicate that it's a system call and not a hardware interrupt
- d. To be filled in as the return value of the system call

The correct answer is: Place for the error number value

Question **5**

Complete

Mark 0.00 out of 1.00

A process blocks itself means

- a. The kernel code of an interrupt handler, moves the process to a waiting queue and calls scheduler
- b. The application code calls the scheduler
- c. The kernel code of system call, called by the process, moves the process to a waiting queue and calls scheduler
- d. The kernel code of system call calls scheduler

The correct answer is: The kernel code of system call, called by the process, moves the process to a waiting queue and calls scheduler

Question **6**

Complete

Mark 1.00 out of 1.00

Select the odd one out

- a. Kernel stack of new process to Process stack of new process
- b. Process stack of running process to kernel stack of running process
- c. Kernel stack of running process to kernel stack of scheduler
- d. Kernel stack of scheduler to kernel stack of new process
- e. Kernel stack of new process to kernel stack of scheduler

The correct answer is: Kernel stack of new process to kernel stack of scheduler

Question **7**

Complete

Mark 0.50 out of 0.50

Match the File descriptors to their meaning

0	Standard Input
2	Standard error
1	Standard output

The correct answer is: 0 → Standard Input, 2 → Standard error, 1 → Standard output

Question **8**

Complete

Mark 1.00 out of 1.00

Which of the following is not a task of the code of swtch() function

- [ ] a. Switch stacks
- [x] b. Change the kernel stack location
- [ ] c. Load the new context
- [x] d. Jump to next context EIP
- [x] e. Save the return value of the old context code
- [ ] f. Save the old context

The correct answers are: Save the return value of the old context code, Change the kernel stack location

Question **9**

Complete

Mark 0.50 out of 0.50

Match the names of PCB structures with kernel

linux	struct task_struct
xv6	struct proc

The correct answer is: linux → struct task_struct, xv6 → struct proc

Question **10**

Complete

Mark 0.50 out of 0.50

Match the MACRO with it's meaning

KERNBASE	2 GB
KERNLINK	2.224 GB
PHYSTOP	224 MB

The correct answer is: KERNBASE → 2 GB, KERNLINK → 2.224 GB, PHYSTOP → 224 MB

Question **11**

Complete

Mark 1.00 out of 1.00

The trapframe, in xv6, is built by the

- a. hardware, trapasm.S
- b. hardware, vectors.S
- c. hardware, vectors.S, trapasm.S, trap()
- d. hardware, vectors.S, trapasm.S
- e. vectors.S, trapasm.S

The correct answer is: hardware, vectors.S, trapasm.S

Question **12**

Complete

Mark 0.00 out of 0.50

Which of the following state transitions are not possible?

- a. Running -> Waiting
- b. Ready -> Waiting
- c. Waiting -> Terminated
- d. Ready -> Terminated

The correct answers are: Ready -> Terminated, Waiting -> Terminated, Ready -> Waiting

◄ Description of some possible course mini projects

Jump to...

(Code) mmap related programs ►

Started on	Saturday, 26 February 2022, 5:18:30 PM
State	Finished
Completed on	Saturday, 26 February 2022, 6:30:44 PM
Time taken	1 hour 12 mins
Grade	**8.55** out of 15.00 (**57**%)

Question **1**

Complete

Mark 0.50 out of 0.50

Map the technique with it's feature/problem

static linking	large executable file
dynamic loading	allocate memory only if needed
dynamic linking	small executable file
static loading	wastage of physical memory

The correct answer is: static linking → large executable file, dynamic loading → allocate memory only if needed, dynamic linking → small executable file, static loading → wastage of physical memory

Question **2**

Complete

Mark 0.00 out of 1.00

Calculate the EAT in NANO-seconds (upto 2 decimal points) w.r.t. a page fault, given

Memory access time = 299 ns

Average page fault service time = 6 ms

Page fault rate = 0.8

Answer: 4.80

The correct answer is: 4800059.80

Question **3**

Complete

Mark 1.00 out of 1.00

Given six memory partitions of 300 KB , 600 KB , 350 KB , 200 KB , 750 KB , and 125 KB (in order), how would the first-fit, best-fit, and worst-fit algorithms place processes of size 115 KB and 500 KB (in order)?

best fit 115 KB	125 KB
best fit 500 KB	600 KB
worst fit 500 KB	635 KB
worst fit 115 KB	750 KB
first fit 500 KB	600 KB
first fit 115 KB	300 KB

The correct answer is: best fit 115 KB → 125 KB, best fit 500 KB → 600 KB, worst fit 500 KB → 635 KB, worst fit 115 KB → 750 KB, first fit 500 KB → 600 KB, first fit 115 KB → 300 KB

Question **4**

Complete

Mark 0.29 out of 1.00

Compare paging with demand paging and select the correct statements.

Select one or more:

- [ ] a. TLB hit ration has zero impact in effective memory access time in demand paging.
- [x] b. Both demand paging and paging support shared memory pages.
- [ ] c. Calculations of number of bits for page number and offset are same in paging and demand paging.
- [ ] d. Demand paging requires additional hardware support, compared to paging.
- [ ] e. The meaning of valid-invalid bit in page table is different in paging and demand-paging.
- [ ] f. Paging requires NO hardware support in CPU
- [ ] g. With paging, it's possible to have user programs bigger than physical memory.
- [ ] h. With demand paging, it's possible to have user programs bigger than physical memory.
- [ ] i. Paging requires some hardware support in CPU
- [x] j. Demand paging always increases effective memory access time.

The correct answers are: Demand paging requires additional hardware support, compared to paging., Both demand paging and paging support shared memory pages., With demand paging, it's possible to have user programs bigger than physical memory., Demand paging always increases effective memory access time., Paging requires some hardware support in CPU, Calculations of number of bits for page number and offset are same in paging and demand paging., The meaning of valid-invalid bit in page table is different in paging and demand-paging.

Question **5**

Complete

Mark 0.36 out of 0.50

Map the parts of a C code to the memory regions they are related to

local variables	stack
global un-initialized variables	bss
static variables	data
global initialized variables	data
function arguments	stack
malloced memory	stack
functions	stack

The correct answer is: local variables → stack, global un-initialized variables → bss, static variables → data, global initialized variables → data, function arguments → stack, malloced memory → heap, functions → code

Question **6**

Complete

Mark 0.75 out of 1.00

which of the following, do you think, are valid concerns for making the kernel pageable?

- [x] a. The kernel must have some dedicated frames for it's own work
- [ ] b. No data structure of kernel should be pageable
- [x] c. The kernel's own page tables should not be pageable
- [ ] d. The disk driver and disk interrupt handler should not be pageable
- [ ] e. No part of kernel code should be pageable.
- [x] f. The page fault handler should not be pageable

The correct answers are: The kernel's own page tables should not be pageable, The page fault handler should not be pageable, The kernel must have some dedicated frames for it's own work, The disk driver and disk interrupt handler should not be pageable

Question **7**

Complete

Mark 0.75 out of 1.00

W.r.t the figure given below, mark the given statements as True or False.

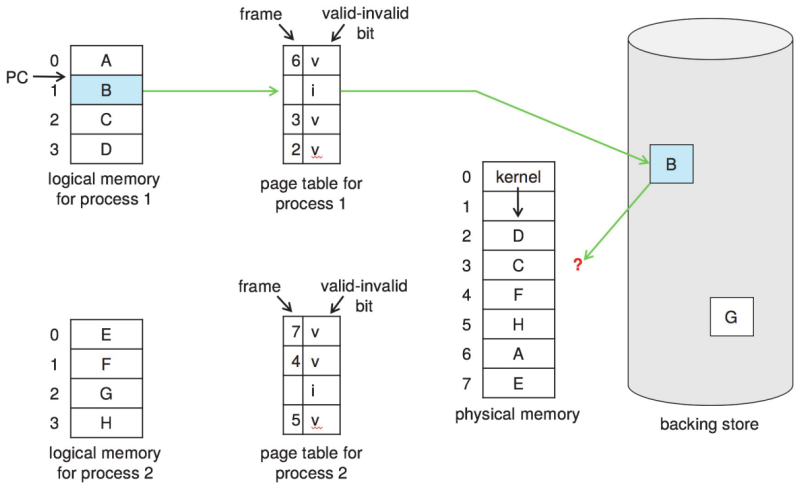


Figure 10.9 Need for page replacement.

True	False	
◉	○	Kernel occupies two page frames
◉	○	Handling this scenario demands two disk I/Os
◉	○	Local replacement means chose any of the frames 2, 3, 6
○	◉	Local replacement means chose any of the frame from 2 to 7
◉	○	Global replacement means chose any of the frame from 0 to 7
○	◉	Global replacement means chose any of the frame from 2 to 7
◉	○	The kernel's pages can not used for replacement if kernel is not pageable.
◉	○	Page 1 of process 1 needs a replacement

Kernel occupies two page frames: True
Handling this scenario demands two disk I/Os: True
Local replacement means chose any of the frames 2, 3, 6: True
Local replacement means chose any of the frame from 2 to 7: False
Global replacement means chose any of the frame from 0 to 7: False
Global replacement means chose any of the frame from 2 to 7: True
The kernel's pages can not used for replacement if kernel is not pageable.: True
Page 1 of process 1 needs a replacement: True

Question **8**

Complete

Mark 0.67 out of 1.00

Shared memory is possible with which of the following memory management schemes ?

Select one or more:

- [x] a. paging
- [x] b. segmentation
- [ ] c. continuous memory management
- [ ] d. demand paging

The correct answers are: paging, segmentation, demand paging

Question **9**

Complete

Mark 0.60 out of 1.00

Given below is the "maps" file for a particular instance of "vim.basic" process.

Mark the given statements as True or False, w.r.t. the contents of the map file.

```
55a43501b000-55a435049000 r--p 00000000 103:05 917529                    /usr/bin/vim.basic
55a435049000-55a435248000 r-xp 0002e000 103:05 917529                    /usr/bin/vim.basic
55a435248000-55a4352b6000 r--p 0022d000 103:05 917529                    /usr/bin/vim.basic
55a4352b7000-55a4352c5000 r--p 0029b000 103:05 917529                    /usr/bin/vim.basic
55a4352c5000-55a4352e2000 rw-p 002a9000 103:05 917529                    /usr/bin/vim.basic
55a4352e2000-55a4352f0000 rw-p 00000000 00:00 0
55a436bc9000-55a436e5b000 rw-p 00000000 00:00 0                          [heap]
7f275b0a3000-7f275b0a6000 r--p 00000000 103:05 917901                    /usr/lib/x86_64-linux-
gnu/libnss_files-2.31.so
7f275b0a6000-7f275b0ad000 r-xp 00003000 103:05 917901                    /usr/lib/x86_64-linux-
gnu/libnss_files-2.31.so
7f275b0ad000-7f275b0af000 r--p 0000a000 103:05 917901                    /usr/lib/x86_64-linux-
gnu/libnss_files-2.31.so
7f275b0af000-7f275b0b0000 r--p 0000b000 103:05 917901                    /usr/lib/x86_64-linux-
gnu/libnss_files-2.31.so
7f275b0b0000-7f275b0b1000 rw-p 0000c000 103:05 917901                    /usr/lib/x86_64-linux-
gnu/libnss_files-2.31.so
7f275b0b1000-7f275b0b7000 rw-p 00000000 00:00 0
7f275b0b7000-7f275b8f5000 r--p 00000000 103:05 925247                    /usr/lib/locale/locale-archive
7f275b8f5000-7f275b8fa000 rw-p 00000000 00:00 0
7f275b8fa000-7f275b8fc000 r--p 00000000 103:05 924216                    /usr/lib/x86_64-linux-
gnu/libogg.so.0.8.4
7f275b8fc000-7f275b901000 r-xp 00002000 103:05 924216                    /usr/lib/x86_64-linux-
gnu/libogg.so.0.8.4
7f275b901000-7f275b904000 r--p 00007000 103:05 924216                    /usr/lib/x86_64-linux-
gnu/libogg.so.0.8.4
7f275b904000-7f275b905000 ---p 0000a000 103:05 924216                    /usr/lib/x86_64-linux-
gnu/libogg.so.0.8.4
7f275b905000-7f275b906000 r--p 0000a000 103:05 924216                    /usr/lib/x86_64-linux-
gnu/libogg.so.0.8.4
7f275b906000-7f275b907000 rw-p 0000b000 103:05 924216                    /usr/lib/x86_64-linux-
gnu/libogg.so.0.8.4
7f275b907000-7f275b90a000 r--p 00000000 103:05 924627                    /usr/lib/x86_64-linux-
gnu/libvorbis.so.0.4.8
7f275b90a000-7f275b921000 r-xp 00003000 103:05 924627                    /usr/lib/x86_64-linux-
gnu/libvorbis.so.0.4.8
7f275b921000-7f275b932000 r--p 0001a000 103:05 924627                    /usr/lib/x86_64-linux-
gnu/libvorbis.so.0.4.8
7f275b932000-7f275b933000 ---p 0002b000 103:05 924627                    /usr/lib/x86_64-linux-
gnu/libvorbis.so.0.4.8
7f275b933000-7f275b934000 r--p 0002b000 103:05 924627                    /usr/lib/x86_64-linux-
gnu/libvorbis.so.0.4.8
7f275b934000-7f275b935000 rw-p 0002c000 103:05 924627                    /usr/lib/x86_64-linux-
gnu/libvorbis.so.0.4.8
7f275b935000-7f275b937000 rw-p 00000000 00:00 0
7f275b937000-7f275b938000 r--p 00000000 103:05 917914                    /usr/lib/x86_64-linux-
gnu/libutil-2.31.so
7f275b938000-7f275b939000 r-xp 00001000 103:05 917914                    /usr/lib/x86_64-linux-
gnu/libutil-2.31.so
7f275b939000-7f275b93a000 r--p 00002000 103:05 917914                    /usr/lib/x86_64-linux-
gnu/libutil-2.31.so
7f275b93a000-7f275b93b000 r--p 00002000 103:05 917914                    /usr/lib/x86_64-linux-
gnu/libutil-2.31.so
7f275b93b000-7f275b93c000 rw-p 00003000 103:05 917914                    /usr/lib/x86_64-linux-
```

```
gnu/libutil-2.31.so
7f275b93c000-7f275b93e000 r--p 00000000 103:05 915906                    /usr/lib/x86_64-linux-
gnu/libz.so.1.2.11
7f275b93e000-7f275b94f000 r-xp 00002000 103:05 915906                    /usr/lib/x86_64-linux-
gnu/libz.so.1.2.11
7f275b94f000-7f275b955000 r--p 00013000 103:05 915906                    /usr/lib/x86_64-linux-
gnu/libz.so.1.2.11
7f275b955000-7f275b956000 ---p 00019000 103:05 915906                    /usr/lib/x86_64-linux-
gnu/libz.so.1.2.11
7f275b956000-7f275b957000 r--p 00019000 103:05 915906                    /usr/lib/x86_64-linux-
gnu/libz.so.1.2.11
7f275b957000-7f275b958000 rw-p 0001a000 103:05 915906                    /usr/lib/x86_64-linux-
gnu/libz.so.1.2.11
7f275b958000-7f275b95c000 r--p 00000000 103:05 923645                    /usr/lib/x86_64-linux-
gnu/libexpat.so.1.6.11
7f275b95c000-7f275b978000 r-xp 00004000 103:05 923645                    /usr/lib/x86_64-linux-
gnu/libexpat.so.1.6.11
7f275b978000-7f275b982000 r--p 00020000 103:05 923645                    /usr/lib/x86_64-linux-
gnu/libexpat.so.1.6.11
7f275b982000-7f275b983000 ---p 0002a000 103:05 923645                    /usr/lib/x86_64-linux-
gnu/libexpat.so.1.6.11
7f275b983000-7f275b985000 r--p 0002a000 103:05 923645                    /usr/lib/x86_64-linux-
gnu/libexpat.so.1.6.11
7f275b985000-7f275b986000 rw-p 0002c000 103:05 923645                    /usr/lib/x86_64-linux-
gnu/libexpat.so.1.6.11
7f275b986000-7f275b988000 r--p 00000000 103:05 924057                    /usr/lib/x86_64-linux-
gnu/libltdl.so.7.3.1
7f275b988000-7f275b98d000 r-xp 00002000 103:05 924057                    /usr/lib/x86_64-linux-
gnu/libltdl.so.7.3.1
7f275b98d000-7f275b98f000 r--p 00007000 103:05 924057                    /usr/lib/x86_64-linux-
gnu/libltdl.so.7.3.1
7f275b98f000-7f275b990000 r--p 00008000 103:05 924057                    /usr/lib/x86_64-linux-
gnu/libltdl.so.7.3.1
7f275b990000-7f275b991000 rw-p 00009000 103:05 924057                    /usr/lib/x86_64-linux-
gnu/libltdl.so.7.3.1
7f275b991000-7f275b995000 r--p 00000000 103:05 921934                    /usr/lib/x86_64-linux-
gnu/libtdb.so.1.4.3
7f275b995000-7f275b9a3000 r-xp 00004000 103:05 921934                    /usr/lib/x86_64-linux-
gnu/libtdb.so.1.4.3
7f275b9a3000-7f275b9a9000 r--p 00012000 103:05 921934                    /usr/lib/x86_64-linux-
gnu/libtdb.so.1.4.3
7f275b9a9000-7f275b9aa000 r--p 00017000 103:05 921934                    /usr/lib/x86_64-linux-
gnu/libtdb.so.1.4.3
7f275b9aa000-7f275b9ab000 rw-p 00018000 103:05 921934                    /usr/lib/x86_64-linux-
gnu/libtdb.so.1.4.3
7f275b9ab000-7f275b9ad000 rw-p 00000000 00:00 0
7f275b9ad000-7f275b9af000 r--p 00000000 103:05 924631                    /usr/lib/x86_64-linux-
gnu/libvorbisfile.so.3.3.7
7f275b9af000-7f275b9b4000 r-xp 00002000 103:05 924631                    /usr/lib/x86_64-linux-
gnu/libvorbisfile.so.3.3.7
7f275b9b4000-7f275b9b5000 r--p 00007000 103:05 924631                    /usr/lib/x86_64-linux-
gnu/libvorbisfile.so.3.3.7
7f275b9b5000-7f275b9b6000 ---p 00008000 103:05 924631                    /usr/lib/x86_64-linux-
gnu/libvorbisfile.so.3.3.7
7f275b9b6000-7f275b9b7000 r--p 00008000 103:05 924631                    /usr/lib/x86_64-linux-
gnu/libvorbisfile.so.3.3.7
7f275b9b7000-7f275b9b8000 rw-p 00009000 103:05 924631                    /usr/lib/x86_64-linux-
gnu/libvorbisfile.so.3.3.7
7f275b9b8000-7f275b9ba000 r--p 00000000 103:05 924277                    /usr/lib/x86_64-linux-
gnu/libpcre2-8.so.0.9.0
7f275b9ba000-7f275ba1e000 r-xp 00002000 103:05 924277                    /usr/lib/x86_64-linux-
gnu/libpcre2-8.so.0.9.0
```

```
7f275ba1e000-7f275ba46000 r--p 00066000 103:05 924277                    /usr/lib/x86_64-linux-
gnu/libpcre2-8.so.0.9.0
7f275ba46000-7f275ba47000 r--p 0008d000 103:05 924277                    /usr/lib/x86_64-linux-
gnu/libpcre2-8.so.0.9.0
7f275ba47000-7f275ba48000 rw-p 0008e000 103:05 924277                    /usr/lib/x86_64-linux-
gnu/libpcre2-8.so.0.9.0
7f275ba48000-7f275ba6d000 r--p 00000000 103:05 917893                    /usr/lib/x86_64-linux-
gnu/libc-2.31.so
7f275ba6d000-7f275bbe5000 r-xp 00025000 103:05 917893                    /usr/lib/x86_64-linux-
gnu/libc-2.31.so
7f275bbe5000-7f275bc2f000 r--p 0019d000 103:05 917893                    /usr/lib/x86_64-linux-
gnu/libc-2.31.so
7f275bc2f000-7f275bc30000 ---p 001e7000 103:05 917893                    /usr/lib/x86_64-linux-
gnu/libc-2.31.so
7f275bc30000-7f275bc33000 r--p 001e7000 103:05 917893                    /usr/lib/x86_64-linux-
gnu/libc-2.31.so
7f275bc33000-7f275bc36000 rw-p 001ea000 103:05 917893                    /usr/lib/x86_64-linux-
gnu/libc-2.31.so
7f275bc36000-7f275bc3a000 rw-p 00000000 00:00 0
7f275bc3a000-7f275bc41000 r--p 00000000 103:05 917906                    /usr/lib/x86_64-linux-
gnu/libpthread-2.31.so
7f275bc41000-7f275bc52000 r-xp 00007000 103:05 917906                    /usr/lib/x86_64-linux-
gnu/libpthread-2.31.so
7f275bc52000-7f275bc57000 r--p 00018000 103:05 917906                    /usr/lib/x86_64-linux-
gnu/libpthread-2.31.so
7f275bc57000-7f275bc58000 r--p 0001c000 103:05 917906                    /usr/lib/x86_64-linux-
gnu/libpthread-2.31.so
7f275bc58000-7f275bc59000 rw-p 0001d000 103:05 917906                    /usr/lib/x86_64-linux-
gnu/libpthread-2.31.so
7f275bc59000-7f275bc5d000 rw-p 00000000 00:00 0
7f275bc5d000-7f275bcce000 r--p 00000000 103:05 917016                    /usr/lib/x86_64-linux-
gnu/libpython3.8.so.1.0
7f275bcce000-7f275bf29000 r-xp 00071000 103:05 917016                    /usr/lib/x86_64-linux-
gnu/libpython3.8.so.1.0
7f275bf29000-7f275c142000 r--p 002cc000 103:05 917016                    /usr/lib/x86_64-linux-
gnu/libpython3.8.so.1.0
7f275c142000-7f275c143000 ---p 004e5000 103:05 917016                    /usr/lib/x86_64-linux-
gnu/libpython3.8.so.1.0
7f275c143000-7f275c149000 r--p 004e5000 103:05 917016                    /usr/lib/x86_64-linux-
gnu/libpython3.8.so.1.0
7f275c149000-7f275c190000 rw-p 004eb000 103:05 917016                    /usr/lib/x86_64-linux-
gnu/libpython3.8.so.1.0
7f275c190000-7f275c1b3000 rw-p 00000000 00:00 0
7f275c1b3000-7f275c1b4000 r--p 00000000 103:05 917894                    /usr/lib/x86_64-linux-
gnu/libdl-2.31.so
7f275c1b4000-7f275c1b6000 r-xp 00001000 103:05 917894                    /usr/lib/x86_64-linux-
gnu/libdl-2.31.so
7f275c1b6000-7f275c1b7000 r--p 00003000 103:05 917894                    /usr/lib/x86_64-linux-
gnu/libdl-2.31.so
7f275c1b7000-7f275c1b8000 r--p 00003000 103:05 917894                    /usr/lib/x86_64-linux-
gnu/libdl-2.31.so
7f275c1b8000-7f275c1b9000 rw-p 00004000 103:05 917894                    /usr/lib/x86_64-linux-
gnu/libdl-2.31.so
7f275c1b9000-7f275c1bb000 rw-p 00000000 00:00 0
7f275c1bb000-7f275c1c0000 r-xp 00000000 103:05 923815                    /usr/lib/x86_64-linux-
gnu/libgpm.so.2
7f275c1c0000-7f275c3bf000 ---p 00005000 103:05 923815                    /usr/lib/x86_64-linux-
gnu/libgpm.so.2
7f275c3bf000-7f275c3c0000 r--p 00004000 103:05 923815                    /usr/lib/x86_64-linux-
gnu/libgpm.so.2
7f275c3c0000-7f275c3c1000 rw-p 00005000 103:05 923815                    /usr/lib/x86_64-linux-
gnu/libgpm.so.2
```

```
7f275c3c1000-7f275c3c3000 r--p 00000000 103:05 923315                    /usr/lib/x86_64-linux-
gnu/libacl.so.1.1.2253
7f275c3c3000-7f275c3c8000 r-xp 00002000 103:05 923315                    /usr/lib/x86_64-linux-
gnu/libacl.so.1.1.2253
7f275c3c8000-7f275c3ca000 r--p 00007000 103:05 923315                    /usr/lib/x86_64-linux-
gnu/libacl.so.1.1.2253
7f275c3ca000-7f275c3cb000 r--p 00008000 103:05 923315                    /usr/lib/x86_64-linux-
gnu/libacl.so.1.1.2253
7f275c3cb000-7f275c3cc000 rw-p 00009000 103:05 923315                    /usr/lib/x86_64-linux-
gnu/libacl.so.1.1.2253
7f275c3cc000-7f275c3cf000 r--p 00000000 103:05 923446                    /usr/lib/x86_64-linux-
gnu/libcanberra.so.0.2.5
7f275c3cf000-7f275c3d9000 r-xp 00003000 103:05 923446                    /usr/lib/x86_64-linux-
gnu/libcanberra.so.0.2.5
7f275c3d9000-7f275c3dd000 r--p 0000d000 103:05 923446                    /usr/lib/x86_64-linux-
gnu/libcanberra.so.0.2.5
7f275c3dd000-7f275c3de000 r--p 00010000 103:05 923446                    /usr/lib/x86_64-linux-
gnu/libcanberra.so.0.2.5
7f275c3de000-7f275c3df000 rw-p 00011000 103:05 923446                    /usr/lib/x86_64-linux-
gnu/libcanberra.so.0.2.5
7f275c3df000-7f275c3e5000 r--p 00000000 103:05 924431                    /usr/lib/x86_64-linux-
gnu/libselinux.so.1
7f275c3e5000-7f275c3fe000 r-xp 00006000 103:05 924431                    /usr/lib/x86_64-linux-
gnu/libselinux.so.1
7f275c3fe000-7f275c405000 r--p 0001f000 103:05 924431                    /usr/lib/x86_64-linux-
gnu/libselinux.so.1
7f275c405000-7f275c406000 ---p 00026000 103:05 924431                    /usr/lib/x86_64-linux-
gnu/libselinux.so.1
7f275c406000-7f275c407000 r--p 00026000 103:05 924431                    /usr/lib/x86_64-linux-
gnu/libselinux.so.1
7f275c407000-7f275c408000 rw-p 00027000 103:05 924431                    /usr/lib/x86_64-linux-
gnu/libselinux.so.1
7f275c408000-7f275c40a000 rw-p 00000000 00:00 0
7f275c40a000-7f275c418000 r--p 00000000 103:05 924540                    /usr/lib/x86_64-linux-
gnu/libtinfo.so.6.2
7f275c418000-7f275c427000 r-xp 0000e000 103:05 924540                    /usr/lib/x86_64-linux-
gnu/libtinfo.so.6.2
7f275c427000-7f275c435000 r--p 0001d000 103:05 924540                    /usr/lib/x86_64-linux-
gnu/libtinfo.so.6.2
7f275c435000-7f275c439000 r--p 0002a000 103:05 924540                    /usr/lib/x86_64-linux-
gnu/libtinfo.so.6.2
7f275c439000-7f275c43a000 rw-p 0002e000 103:05 924540                    /usr/lib/x86_64-linux-
gnu/libtinfo.so.6.2
7f275c43a000-7f275c449000 r--p 00000000 103:05 917895                    /usr/lib/x86_64-linux-
gnu/libm-2.31.so
7f275c449000-7f275c4f0000 r-xp 0000f000 103:05 917895                    /usr/lib/x86_64-linux-
gnu/libm-2.31.so
7f275c4f0000-7f275c587000 r--p 000b6000 103:05 917895                    /usr/lib/x86_64-linux-
gnu/libm-2.31.so
7f275c587000-7f275c588000 r--p 0014c000 103:05 917895                    /usr/lib/x86_64-linux-
gnu/libm-2.31.so
7f275c588000-7f275c589000 rw-p 0014d000 103:05 917895                    /usr/lib/x86_64-linux-
gnu/libm-2.31.so
7f275c589000-7f275c58b000 rw-p 00000000 00:00 0
7f275c5ae000-7f275c5af000 r--p 00000000 103:05 917889                    /usr/lib/x86_64-linux-gnu/ld-
2.31.so
7f275c5af000-7f275c5d2000 r-xp 00001000 103:05 917889                    /usr/lib/x86_64-linux-gnu/ld-
2.31.so
7f275c5d2000-7f275c5da000 r--p 00024000 103:05 917889                    /usr/lib/x86_64-linux-gnu/ld-
2.31.so
7f275c5db000-7f275c5dc000 r--p 0002c000 103:05 917889                    /usr/lib/x86_64-linux-gnu/ld-
2.31.so
```

```
7f275c5dc000-7f275c5dd000 rw-p 0002d000 103:05 917889                     /usr/lib/x86_64-linux-gnu/ld-
2.31.so
7f275c5dd000-7f275c5de000 rw-p 00000000 00:00 0
7ffd22d2f000-7ffd22d50000 rw-p 00000000 00:00 0                           [stack]
7ffd22db0000-7ffd22db4000 r--p 00000000 00:00 0                           [vvar]
7ffd22db4000-7ffd22db6000 r-xp 00000000 00:00 0                           [vdso]
ffffffffff600000-ffffffffff601000 --xp 00000000 00:00 0                   [vsyscall]
```

True	False	
○	◉	The size of the heap is one page
◉	○	This is a virtual memory map (not physical memory map)
○	◉	vim.basic uses the math library
◉	○	The 5th entry 55a4352c5000-55a4352e2000 **may** correspond to "data" of the vim.basic
◉	○	The size of the stack is one page

The size of the heap is one page: False
This is a virtual memory map (not physical memory map): True
vim.basic uses the math library: True
The 5th entry 55a4352c5000-55a4352e2000 **may** correspond to "data" of the vim.basic: True
The size of the stack is one page: False

Question **10**

Complete

Mark 0.00 out of 1.00

Select all the correct statements, w.r.t. Copy on Write

- ☐ a. Fork() used COW technique to improve performance of new process creation.
- ☑ b. Vfork() assumes that there will be no write, but rather exec()
- ☑ c. COW helps us save memory
- ☐ d. If either parent or child modifies a COW-page, then a copy of the page is made and page table entry is updated
- ☐ e. use of COW during fork() is useless if child called exit()
- ☑ f. use of COW during fork() is useless if exec() is called by the child

The correct answers are: Fork() used COW technique to improve performance of new process creation., If either parent or child modifies a COW-page, then a copy of the page is made and page table entry is updated, COW helps us save memory, Vfork() assumes that there will be no write, but rather exec()

Question **11**

Complete

Mark 1.00 out of 1.00

Assuming a 8- KB page size, what is the page numbers for the address 1093943 reference in decimal :

(give answer also in decimal)

Answer: 134

The correct answer is: 134

Question **12**

Complete

Mark 0.00 out of 1.00

Order the following events, related to page fault handling, in correct order

1. Page fault handler detects that it's a page fault and not illegal memory access
2. Disk interrupt handler runs
3. MMU detects that a page table entry is marked "invalid"
4. Page faulted process is moved to ready-queue
5. Empty frame is found
6. Page fault interrupt is generated
7. Other processes scheduled by scheduler
8. Page table of page faulted process is updated
9. Disk Interrupt occurs
10. Disk read is issued
11. Page fault handler in kernel starts executing
12. Page faulting process is made to wait in a queue

The correct order for these items is as follows:

1. MMU detects that a page table entry is marked "invalid"
2. Page fault interrupt is generated
3. Page fault handler in kernel starts executing
4. Page fault handler detects that it's a page fault and not illegal memory access
5. Empty frame is found
6. Disk read is issued
7. Page faulting process is made to wait in a queue
8. Other processes scheduled by scheduler
9. Disk Interrupt occurs
10. Disk interrupt handler runs
11. Page table of page faulted process is updated
12. Page faulted process is moved to ready-queue

Question **13**

Complete

Mark 1.00 out of 1.00

Page sizes are a power of 2 because

Select one:

- a. Certain bits are reserved for offset in logical address. Hence page size = 2^(no.of offset bits)
- b. Power of 2 calculations are highly efficient
- c. Certain bits are reserved for offset in logical address. Hence page size = 2^(32 - no.of offset bits)
- d. operating system calculations happen using power of 2
- e. MMU only understands numbers that are power of 2

The correct answer is: Certain bits are reserved for offset in logical address. Hence page size = 2^(no.of offset bits)

Question **14**

Complete

Mark 1.00 out of 1.00

Given below is the output of the command "ps -eo min_flt,maj_flt,cmd" on a Linux Desktop system. Select the statements that are consistent with the output

```
626729 482768 /usr/lib/firefox/firefox -contentproc -parentBuildID 20220202182137 -prefsLen 9256 -
prefMapSize 264738 -appDir /usr/lib/firefox/browser 6094 true rdd
2167 687 /usr/sbin/apache2 -k start
1265185 222 /usr/bin/gnome-shell
102648 111 /usr/sbin/mysqld
9813 0 bash
15497 370 /usr/bin/gedit --gapplication-service
```

- a. All of the processes here exihibit some good locality of reference
- b. Firefox has likely been running for a large amount of time
- c. The bash shell is mostly busy doing work within a particular locality
- d. Apache web-server has not been doing much work

The correct answers are: Firefox has likely been running for a large amount of time, Apache web-server has not been doing much work, The bash shell is mostly busy doing work within a particular locality, All of the processes here exihibit some good locality of reference

Question **15**

Complete

Mark 0.14 out of 1.00

Suppose two processes share a library between them. The library consists of 5 pages, and these 5 pages are mapped to frames 9, 15, 23, 4, 7 respectively. Process P1 has got 6 pages, first 3 of which consist of process's own code/data and 3 correspond to library's pages 0, 2, 4. Process P2 has got 7 pages, first 3 of which consist of processe's own code/data and remaining 4 correspond to library's pages 0, 1, 3, 4. Fill in the blanks for page table entries of P1 and P2.

Page table of P1, Page 5	23
Page table of P1, Page 3	9
Page table of P2, Page 0	6
Page table of P2, Page 3	7
Page table of P2, Page 4	9
Page table of P2, Page 1	5
Page table of P1, Page 4	9

The correct answer is: Page table of P1, Page 5 → 7, Page table of P1, Page 3 → 9, Page table of P2, Page 0 → 9, Page table of P2, Page 3 → 4, Page table of P2, Page 4 → 7, Page table of P2, Page 1 → 15, Page table of P1, Page 4 → 23

Question **16**

Complete

Mark 0.50 out of 1.00

For the reference string
3 4 3 5 2

the number of page faults (including initial ones) using

FIFO replacement and 2 page frames is :

4

FIFO replacement and 3 page frames is :

3

◄ (Code) mmap related programs

Jump to...

Points from Mid-term feedback ►

Started on	Monday, 7 March 2022, 7:00:12 PM
State	Finished
Completed on	Monday, 7 March 2022, 8:00:04 PM
Time taken	59 mins 52 secs
Grade	**9.78** out of 15.00 (**65**%)

Question **1**

Complete

Mark 1.00 out of 1.00

Why is there a call to kinit2? Why is it not merged with knit1?

- a. call to seginit() makes it possible to actually use PHYSTOP in argument to kinit2()
- b. When kinit1() is called there is a need for few page frames, but later knit2() is called to serve need of more page frames
- c. Because there is a limit on the values that the argumets to knit1() can take.
- d. knit2 refers to virtual addresses beyond 4MB, which are not mapped before kvalloc() is called

The correct answer is: knit2 refers to virtual addresses beyond 4MB, which are not mapped before kvalloc() is called

Question **2**

Complete

Mark 1.50 out of 1.50

Arrange the following in the correct order of execution (w.r.t. 'init')

initcode() returns in forkret()	6
'initcode' process is marked RUNNABLE	3
'initcode' struct proc is created	2
userinit() is called	1
mpmain() calls scheduler()	4
initcode() calls exec("/init", ...)	8
initcode() returns from trapret()	7
scheduler() schedules initcode() process	5

The correct answer is: initcode() returns in forkret() → 6, 'initcode' process is marked RUNNABLE → 3, 'initcode' struct proc is created → 2, userinit() is called → 1, mpmain() calls scheduler() → 4, initcode() calls exec("/init", ...) → 8, initcode() returns from trapret() → 7, scheduler() schedules initcode() process → 5

Question **3**

Complete

Mark 0.00 out of 2.00

exec() does this: curproc->tf->eip = elf.entry, but userinit() does this: p->tf->eip = 0; Select all the statements from below, that collectively explain this

- [x] a. the 'entry' in initcode is anyways 0
- [x] b. exec() loads from ELF file and the address of first instruction to be executed is given by 'entry'
- [ ] c. the initcode is created using objcopy, which discards all relocation information and symbols (like entry)
- [x] d. elf.entry is anyways 0, so both statements mean the same
- [x] e. In userinit() the function inituvm() has mapped the code of 'initcode' to be starting at virtual address 0
- [ ] f. the code of 'initcode' is loaded at physical address 0

The correct answers are: exec() loads from ELF file and the address of first instruction to be executed is given by 'entry', In userinit() the function inituvm() has mapped the code of 'initcode' to be starting at virtual address 0, the initcode is created using objcopy, which discards all relocation information and symbols (like entry)

Question **4**

Complete

Mark 1.00 out of 1.00

What does userinit() do ?

- ( ) a. initializes the users
- ( ) b. sets up the 'initcode' process to start execution in forkret ()
- (•) c. sets up the 'initcode' process to start execution in forkret()
- ( ) d. sets up the 'initcode' process to start execution in trapret()
- ( ) e. sets up the 'init' process to start execution in forkret()
- ( ) f. initializes the process 'init' and starts executing it

The correct answer is: sets up the 'initcode' process to start execution in forkret()

Question **5**

Complete

Mark 0.67 out of 1.00

Which of the following is done by mappages()?

- [ ] a. allocate page directory if required
- [ ] b. allocate page frame if required
- [x] c. allocate page table if required
- [x] d. create page table mappings for the range given by "va" and "va + size"
- [ ] e. create page table mappings to the range given by "pa" and "pa + size"

The correct answers are: create page table mappings for the range given by "va" and "va + size", allocate page table if required, create page table mappings to the range given by "pa" and "pa + size"

Question **6**

Complete

Mark 0.00 out of 1.00

Select the statement that most correctly describes what setupkvm() does

- a. creates a 2-level page table setup with virtual->physical mappings specified in the kmap[] global arrray
- b. creates a 2-level page table setup with virtual->physical mappings specified in the kmap[] global arrray and makes kpgdir point to it
- c. creates a 1-level page table for the use by the kernel, as specified in kmap[] global array
- d. creates a 2-level page table for the use of the kernel, as specified in gdtdesc

The correct answer is: creates a 2-level page table setup with virtual->physical mappings specified in the kmap[] global arrray

Question **7**

Complete

Mark 0.00 out of 1.00

The approximate number of page frames created by kinit1 is

- a. 4
- b. 16
- c. 4000
- d. 3000
- e. 2000
- f. 1000
- g. 10

The correct answer is: 3000

Question **8**

Complete

Mark 1.20 out of 1.50

Which of the following is DONE by allocproc() ?

- [ ] a. setup kernel memory mappings for the process
- [x] b. Select an UNUSED struct proc for use
- [ ] c. ensure that the process starts in trapret()
- [x] d. allocate kernel stack for the process
- [ ] e. allocate PID to the process
- [x] f. ensure that the process starts in forkret()
- [x] g. setup the trapframe and context pointers appropriately
- [ ] h. setup the contents of the trapframe of the process properly

The correct answers are: Select an UNUSED struct proc for use, allocate PID to the process, allocate kernel stack for the process, setup the trapframe and context pointers appropriately, ensure that the process starts in forkret()

Question **9**

Complete

Mark 1.00 out of 1.00

Map the virtual address to physical address in xv6

KERNLINK	0x100000
0xFE000000	0xFE000000
80108000	0x108000
KERNBASE	0

The correct answer is: KERNLINK → 0x100000, 0xFE000000 → 0xFE000000, 80108000 → 0x108000, KERNBASE → 0

Question **10**

Complete

Mark 0.42 out of 1.00

Select all the correct statements about initcode

- [ ] a. code of initcode is loaded at virtual address 0
- [x] b. The data and stack of initcode is mapped to one single page in userinit()
- [ ] c. code of 'initcode' is loaded along with the kernel during booting
- [x] d. the size of 'initcode' is 2c
- [ ] e. code of initcode is loaded in memory by the kernel during userinit()
- [x] f. initcode is the 'init' process
- [x] g. initcode essentially calls exec("/init",...)

The correct answers are: code of 'initcode' is loaded along with the kernel during booting, the size of 'initcode' is 2c, The data and stack of initcode is mapped to one single page in userinit(), initcode essentially calls exec("/init",...)

Question **11**

Complete

Mark 1.00 out of 1.00

The variable 'end' used as argument to kinit1 has the value

- (•) a. 801154a8
- ( ) b. 81000000
- ( ) c. 80110000
- ( ) d. 80102da0
- ( ) e. 8010a48c
- ( ) f. 80000000

The correct answer is: 801154a8

Question **12**

Complete

Mark 1.00 out of 1.00

What does seginit() do?

- ( ) a. Nothing significant, just repetition of earlier GDT setup but with 2-level paging setup done
- ( ) b. Nothing significant, just repetition of earlier GDT setup but with free frames list created now
- ( ) c. Adds two additional entries to GDT corresponding to Code and Data segments, but to be used in privilege level 0
- (•) d. Adds two additional entries to GDT corresponding to Code and Data segments, but to be used in privilege level 3
- ( ) e. Nothing significant, just repetition of earlier GDT setup but with kernel page table allocated now

The correct answer is: Adds two additional entries to GDT corresponding to Code and Data segments, but to be used in privilege level 3

Question **13**

Complete

Mark 1.00 out of 1.00

Does exec() code around clearptau() lead to wastage of one page frame?

- a. no
- b. yes

The correct answer is: yes

◄ Questions for test on kalloc/kfree/kvmalloc, etc.

Jump to...

(Optional Assignment) Slab allocator in xv6 ►