Dashboard / My courses / Computer Engineering & IT / CEIT-Even-sem-20-21 / OS-Even-sem-2020-21 / 14 February - 20 February / Quiz-1

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.

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

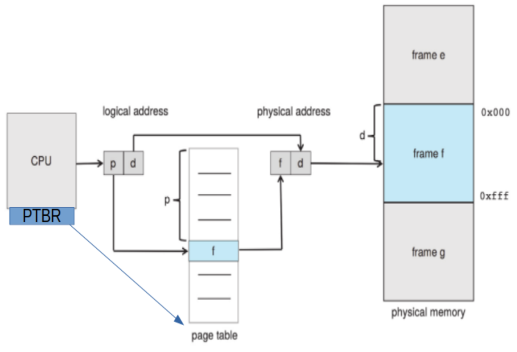

Figure 9.8 Paging hardware.

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...