Started on	Friday, 24 February 2023, 2:44 PM
State	Finished
Completed on	Friday, 24 February 2023, 4:39 PM
Time taken	1 hour 55 mins
Grade	**8.78** out of 10.00 (**87.77**%)

Question **1**

Correct

Mark 0.50 out of 0.50

What is meant by formatting a disk/partition?

a. erasing all data on the disk/partition

b. storing all the necessary programs on the disk/partition

c. creating layout of empty directory tree/graph data structure ✔

d. writing zeroes on all sectors

The correct answer is: creating layout of empty directory tree/graph data structure

Question **2**

Partially correct

Mark 0.44 out of 0.50

How does the compiler calculate addresses for the different parts of a C program, when paging is used?

Text	starting with 0	✔
Static variables	Immediately after the text, along with globals	✔
Local variables	An offset with respect to stack pointer (esp)	✔
#include files	No memory allocated, they are handled by linker	✘
#define	No memory allocated, they are handled by pre-processor	✔
typedef	No memory allocated, as they are not variables, but only conceptual definition of a type	✔
Global variables	Immediately after the text	✔
malloced memory	Heap (handled by the malloc-free library, using OS's system calls)	✔

Your answer is partially correct.

You have correctly selected 7.

The correct answer is: Text → starting with 0, Static variables → Immediately after the text, along with globals, Local variables → An offset with respect to stack pointer (esp), #include files → No memory allocated for the file, but if it contains variables, then variables may be allocated memory, #define → No memory allocated, they are handled by pre-processor, typedef → No memory allocated, as they are not variables, but only conceptual definition of a type, Global variables → Immediately after the text, malloced memory → Heap (handled by the malloc-free library, using OS's system calls)

Question **3**

Correct

Mark 0.50 out of 0.50

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:

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

Your answer is correct.

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 **4**

Partially correct

Mark 0.43 out of 0.50

You must have seen the error message "Segmentation fault, core dumped" very often.

With respect to this error message, mark the statements as True/False.

True	False			
✗	● ✓	The term "core" refers to the core code of the kernel.	✔	core means memory, all memory for the process.
✗	● ✓	On Linux, the message is printed only because the memory management scheme is segmentation	✔	No, it's just a term used, even if paging is used for memory management.
● ✓	✗	The process has definitely performed illegal memory access.	✔	
● ✓	✗	On Linux, the process was sent a SIGSEGV signal and the default handler for the signal is "Term", so the process is terminated.	✔	
● ✗	✓	The illegal memory access was detected by the kernel and the process was punished by kernel.	✖	"detection" is done by CPU, not kernel.
● ✓	✗	The core file can be analysed later using a debugger, to determine what went wrong.	✔	use gdb ./core ./executable-filename
● ✓	✗	The image of the process is stored in a file called "core", if the ulimit allows so.	✔	see ulimit -a

The term "core" refers to the core code of the kernel.: False
On Linux, the message is printed only because the memory management scheme is segmentation: False
The process has definitely performed illegal memory access.: True
On Linux, the process was sent a SIGSEGV signal and the default handler for the signal is "Term", so the process is terminated.: True
The illegal memory access was detected by the kernel and the process was punished by kernel.: False
The core file can be analysed later using a debugger, to determine what went wrong.: True
The image of the process is stored in a file called "core", if the ulimit allows so.: True

Question **5**

Correct

Mark 0.50 out of 0.50

Which of the following instructions should be privileged?

Select one or more:

- [x] a. Access memory management unit of the processor ✔
- [x] b. Turn off interrupts. ✔
- [ ] c. Read the clock.
- [x] d. Set value of timer. ✔
- [x] e. Switch from user to kernel mode. ✔ This instruction (like INT) is itself privileged - and that is why it not only changes the mode, but also ensures a jump to an ISR (kernel code)
- [ ] f. Access a general purpose register
- [ ] g. Set value of a memory location
- [x] h. Modify entries in device-status table ✔
- [x] i. Access I/O device. ✔

Your answer is correct.

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 **6**

Correct

Mark 0.50 out of 0.50

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)

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

Your answer is correct.

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

Question **7**

Partially correct

Mark 0.43 out of 0.50

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[ 2 ✔ ][2];

   pipe( pfd[0] ✔ );
   pid1 = fork() ✔ ;
   if(pid1 != 0) {
      close(pfd[0] [0] ✔ );
      close( 1 ✔ );
      dup( pfd[0][1] ✔ );
      execl("/bin/ls", "/bin/ls", " -1 ✘ ", NULL);
   }
   pipe( pfd[1] ✔ );
   pid2 ✔ = fork();
   if(pid2 == 0) {
      close( pfd[0][1] ✘ ;
      close(0);
      dup( pfd[1][0] ✘ );
      close(pfd[1] [0] ✔ );
      close( 1 ✔ );
      dup( pfd[1][1] ✔ );
      execl("/usr/bin/head", "/usr/bin/head", " -3 ✔ ", NULL);
   } else {
      close(pfd [1][1] ✔ );
      close( 0 ✔ );
      dup( pfd[1][0] ✔ );
      close(pfd [0][0] ✔ );
```

execl("/usr/bin/tail", "/usr/bin/tail", " -1 ", NULL);

}

}

Question **8**

Partially correct

Mark 0.45 out of 0.50

Match the elements of C program to their place in memory

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

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

Question **9**

Correct

Mark 0.50 out of 0.50

Doing a lookup on the pathname /a/b/b/c/d for opening the file "d" requires reading 6 no. of inodes. Assume that there are no hard/soft links on the path.

Write the answer as a number.

The correct answer is: 6

Question **10**

Partially correct

Mark 0.42 out of 0.50

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

True	False		
✗	✓	Different devices of the same type (e.g. 2 IDE hard disks) must need different device drivers.	✗
✓	✗	It's possible that a particular hardware has multiple device drivers available for it.	✓
✓	✗	Device driver is part of OS code	✓
✓	✗	Device driver is an intermediary between the hardware controller and OS	✓
✗	✓	Device driver is part of hardware	✓
✓	✗	Writing a device driver mandatorily demands reading the technical documentation about the hardware.	✓

Different devices of the same type (e.g. 2 IDE hard disks) must need different device drivers.: False
It's possible that a particular hardware has multiple device drivers available for it.: True
Device driver is part of OS code: True
Device driver is an intermediary between the hardware controller and OS: True
Device driver is part of hardware: False
Writing a device driver mandatorily demands reading the technical documentation about the hardware.: True

Question **11**

Correct

Mark 0.50 out of 0.50

Map each signal with it's meaning

SIGSEGV	Invalid Memory Reference	✔
SIGALRM	Timer Signal from alarm()	✔
SIGPIPE	Broken Pipe	✔
SIGUSR1	User Defined Signal	✔
SIGCHLD	Child Stopped or Terminated	✔

The correct answer is: SIGSEGV → Invalid Memory Reference, SIGALRM → Timer Signal from alarm(), SIGPIPE → Broken Pipe, SIGUSR1 → User Defined Signal, SIGCHLD → Child Stopped or Terminated

Question **12**

Partially correct

Mark 0.25 out of 0.50

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"

Block bitmap(s) for all the blocks of the file	Yes	✔
Data blocks of the file	Yes	✖
One or multiple data blocks of the parent directory	Yes	✖
Superblock	Yes	✔
Possibly one block bitmap corresponding to the parent directory	Yes	✔
One or more data bitmap blocks for the parent directory	Yes	✖

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 3.
The correct answer is: Block bitmap(s) for all the blocks of the file → Yes, Data blocks of the file → No, One or multiple data blocks of the parent directory → No, Superblock → Yes, Possibly one block bitmap corresponding to the parent directory → Yes, One or more data bitmap blocks for the parent directory → No

Question **13**

Correct

Mark 0.50 out of 0.50

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	many continuous chunks of variable size ✔
Segmentation, then paging	many continuous chunks of variable size ✔
Paging	one continuous chunk ✔
Relocation + Limit	one continuous chunk ✔

Your answer is correct.

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

Question **14**

Partially correct

Mark 0.30 out of 0.50

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

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

Question **15**

Partially correct

Mark 0.29 out of 0.50

Mark the statements as True/False w.r.t. the basic concepts of memory management.

True	False		
○ ✗	◉ ✓	The kernel refers to the page table for converting each virtual address to physical address.	✓
◉ ✓	○ ✗	When a process is executing, each virtual address is converted into physical address by the CPU hardware directly.	✓
◉ ✗	○ ✓	The compiler interacts with the kernel continuously while compiling a program and obtains the correct set of memory addresses for code/stack/heap/data and then generates the machine code file.	✗
○ ✗	◉ ✓	The compiler generates the address references for code/data/stack/heap in the executable file as per the memory management schema chosen by the compiler itself, and then the kernel ensures that program is executed with this schema.	✓
○ ✓	◉ ✗	The kernel ensures that the MMU is setup before scheduling a process and then the CPU/MMU ensures that the address translation takes place.	✗
◉ ✗	○ ✓	When a process is executing, each virtual address is converted into physical address by the kernel directly.	✗
◉ ✓	○ ✗	The compiler generates address references for code/data/stack/heap in the executable file, depending on the MM architecture provided by CPU and kernel.	✓

The kernel refers to the page table for converting each virtual address to physical address.: False
When a process is executing, each virtual address is converted into physical address by the CPU hardware directly.: True
The compiler interacts with the kernel continuously while compiling a program and obtains the correct set of memory addresses for code/stack/heap/data and then generates the machine code file.: False
The compiler generates the address references for code/data/stack/heap in the executable file as per the memory management schema chosen by the compiler itself, and then the kernel ensures that program is executed with this schema.: False
The kernel ensures that the MMU is setup before scheduling a process and then the CPU/MMU ensures that the address translation takes place.: True
When a process is executing, each virtual address is converted into physical address by the kernel directly.: False
The compiler generates address references for code/data/stack/heap in the executable file, depending on the MM architecture provided by CPU and kernel.: True

Question **16**

Correct

Mark 0.50 out of 0.50

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!

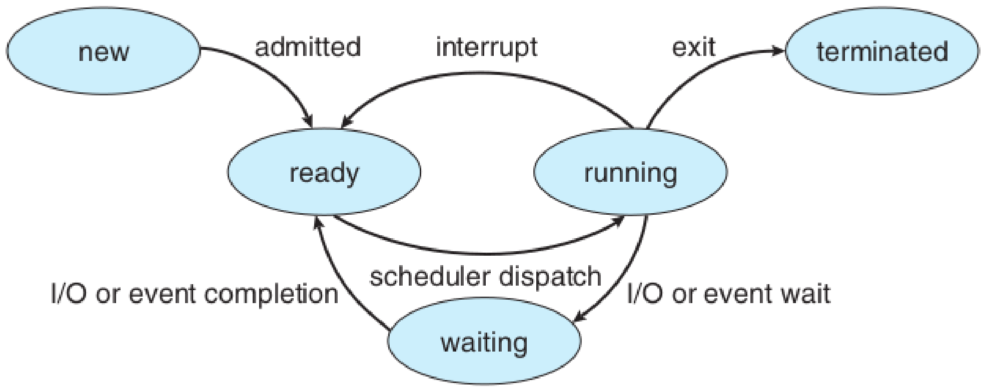


Figure 3.2 Diagram of process state.

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	✔
◉ ✓	○ ✗	Only a process in READY state is considered by scheduler	✔
◉ ✓	○ ✗	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	✔
○ ✗	◉ ✓	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
Only a process in READY state is considered by scheduler: True
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
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 **17**

Partially correct

Mark 0.45 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	✔
○ ✖	◉ ✔	Pointer to IDT	✔
○ ✖	◉ ✔	PID of Init	✔
◉ ✔	○ ✖	PID	✔
○ ✖	◉ ✔	Function pointers to all system calls	✔
○ ✔	◉ ✖	EIP at the time of context switch	✖
◉ ✔	○ ✖	Process state	✔
◉ ✔	○ ✖	List of opened files	✔
◉ ✔	○ ✖	Process context	✔
◉ ✔	○ ✖	Pointer to the parent process	✔

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

Question **18**

Correct

Mark 0.50 out of 0.50

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 **19**

Correct

Mark 0.50 out of 0.50

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

Select one:

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

Your answer is correct.

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

Question **20**

Partially correct

Mark 0.33 out of 0.50

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

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

Your answer is partially correct.

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

◄ Quiz-1 Preparation questions

Jump to...

Quiz - 2 (17 March 2023) ►

Started on	Friday, 17 March 2023, 2:33 PM
State	Finished
Completed on	Friday, 17 March 2023, 4:54 PM
Time taken	2 hours 21 mins
Grade	**6.37** out of 10.00 (**63.73**%)

Question **1**

Incorrect

Mark 0.00 out of 0.50

The first instruction that runs when you do "make qemu" is

`cli`

from bootasm.S

Why?

a. "cli" stands for clear screen and the screen should be cleared before OS boots.

b. "cli" that is Command Line Interface needs to be enabled first

c. "cli" clears all registers and makes them zero, so that processor is as good as "new"

d. "cli" clears the pipeline of the CPU so that it is as good as "fresh" CPU

e. "cli" enables interrupts, it is required because the kernel supports interrupts.

f. "cli" disables interrupts. It is required because as of now there are no interrupt handlers available ✖

g. It disables interrupts. It is required because the interrupt handlers of kernel are not yet installed.

h. "cli" enables interrupts, it is required because the kernel must handle interrupts.

Your answer is incorrect.

The correct answer is: It disables interrupts. It is required because the interrupt handlers of kernel are not yet installed.

Question **2**

Incorrect

Mark 0.00 out of 0.50

The struct buf has a sleeplock, and not a spinlock, because

- a. struct buf is used as a general purpose cache by kernel and cache operations take lot of time, so better to use sleeplock rather than spinlock
- b. sleeplock is preferable because it is used in interrupt context and spinlock can not be used in interrupt context
- c. It could be a spinlock, but xv6 has chosen sleeplock for purpose of demonstrating how to use a sleeplock.
- d. struct buf is used for disk I/o which takes lot of time, so sleeping/blocking is the only option available. ✖
- e. struct buf is used for disk I/o which takes lot of time, so sleeping/blocking is preferred to spinning/busy-wait for the desired buf.

Your answer is incorrect.

The correct answer is: struct buf is used for disk I/o which takes lot of time, so sleeping/blocking is preferred to spinning/busy-wait for the desired buf.

Question **3**

Partially correct

Mark 0.25 out of 0.50

when is each of the following stacks allocated?

kernel stack of process	during fork() in allocproc()	✔
kernel stack for scheduler, on first processor	on the arrival of a hardware interrupt	✖
user stack of process	during exec()	✖
kernel stack for the scheduler, on other processors	in main()->startothers()	✔

Your answer is partially correct.

You have correctly selected 2.
The correct answer is: kernel stack of process → during fork() in allocproc(), kernel stack for scheduler, on first processor → in entry.S, user stack of process → during fork() in copyuvm(), kernel stack for the scheduler, on other processors → in main()->startothers()

Question **4**

Partially correct

Mark 0.56 out of 0.75

Mark statements as True/False w.r.t. ptable.lock

True	False			
◉ ✘	○ ✔	A process can sleep on ptable.lock if it can't aquire it.	✘	It's a spinlock!
○ ✔	◉ ✘	It is taken by one process but released by another process, running on same processor	✘	
◉ ✔	○ ✘	ptable.lock protects the proc[] array and all struct proc in the array	✔	
○ ✘	◉ ✔	The swtch() in scheduler() is called without holding the ptable.lock when control jumps to it from sched()	✔	No. it's always held. sched() will hold the lock.
◉ ✔	○ ✘	the rule of "never block holding a spinlock" does not apply to ptable.lock in xv6	✔	sched() is called only if you hold ptable.lock
◉ ✔	○ ✘	One sequence of function calls which takes and releases the ptable.lock is this: iderw->sleep, acquire(ptable.lock)->sched->swtch()->scheduler()->swtch()->yield(), release(ptable.lock)	✔	One process slept, another was scheduled and it came out of timer interrupt.
○ ✘	◉ ✔	ptable.lock can be held by different processes on different processors at the same time	✔	No lock can be held like this!
○ ✘	◉ ✔	ptable.lock is acquired but never released	✔	how is that possible?

A process can sleep on ptable.lock if it can't aquire it.: False
It is taken by one process but released by another process, running on same processor: True
ptable.lock protects the proc[] array and all struct proc in the array: True
The swtch() in scheduler() is called without holding the ptable.lock when control jumps to it from sched(): False
the rule of "never block holding a spinlock" does not apply to ptable.lock in xv6: True
One sequence of function calls which takes and releases the ptable.lock is this:
iderw->sleep, acquire(ptable.lock)->sched->swtch()->scheduler()->swtch()->yield(), release(ptable.lock): True
ptable.lock can be held by different processes on different processors at the same time: False
ptable.lock is acquired but never released: False

Question **5**

Correct

Mark 0.50 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.

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

Your answer is correct.

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 **6**

Partially correct

Mark 0.43 out of 0.75

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

movw %ax, %gs	real mode	✖
jmp *%eax	protected mode with segmentation and 4 MB pages	✔
inb $0x64,%al	real mode	✔
ljmp $(SEG_KCODE<<3), $start32	protected mode with only segmentation	✖
readseg((uchar*)elf, 4096, 0);	protected mode with only segmentation	✔
orl $CR0_PE, %eax	real mode	✔
movl $(V2P_WO(entrypgdir)), %eax	protected mode with segmentation and 4 MB pages	✖

The correct answer is: movw %ax, %gs → protected mode with only segmentation, jmp *%eax → protected mode with segmentation and 4 MB pages, inb $0x64,%al → real mode, ljmp $(SEG_KCODE<<3), $start32 → real mode, readseg((uchar*)elf, 4096, 0); → protected mode with only segmentation, orl $CR0_PE, %eax → real mode, movl $(V2P_WO(entrypgdir)), %eax → protected mode with only segmentation

Question **7**

Correct

Mark 0.50 out of 0.50

Which of the following is DONE by allocproc() ?

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

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 **8**

Partially correct

Mark 0.25 out of 0.50

Mark statements as True/False w.r.t. the creation of free page list in xv6.

True	False			
◉	○	kmem.use_lock is set to 1 after free page list is created, so that kmem.lock is taken before accessing kmem.freelist.	✖	
◉	○	The pointers that link the pages together are in the first 4 bytes of the pages themselves	✔	

◉	○	if(kmem.use_lock)
		    acquire(&kmem.lock);
		is not done when called from kinit1() because there is no need to take the lock when kinit1() is running because interrupts are disabled and only one processor is running
			✔	

○	◉	free page list is a singly circular linked list.	✔	it's singly linked NULL terminated list.

◉	○	if(kmem.use_lock)
		    acquire(&kmem.lock);
		this "if" condition is true, when kinit2() runs because multi-processor support has been enabled by now.
			✖	No. kinit2() calls kfree() and then initializes use_lock.

○	◉	the kmem.lock is used by kfree() and kalloc() only.	✖	

kmem.use_lock is set to 1 after free page list is created, so that kmem.lock is taken before accessing kmem.freelist.: True
The pointers that link the pages together are in the first 4 bytes of the pages themselves: True
if(kmem.use_lock)
    acquire(&kmem.lock);
is not done when called from kinit1() because there is no need to take the lock when kinit1() is running because interrupts are disabled and only one processor is running: True
free page list is a singly circular linked list.: False
if(kmem.use_lock)
    acquire(&kmem.lock);
this "if" condition is true, when kinit2() runs because multi-processor support has been enabled by now.: False
the kmem.lock is used by kfree() and kalloc() only.: True

Question **9**

Partially correct

Mark 0.55 out of 1.00

Given below is code of sleeplock in xv6.

```
// Long-term locks for processes
struct sleeplock {
  uint locked;       // Is the lock held?
  struct spinlock lk; // spinlock protecting this sleep lock

  // For debugging:
  char *name;        // Name of lock.
  int pid;           // Process holding lock
};



void
acquiresleep(struct sleeplock *lk)
{
  acquire(&lk->lk);
  while (lk->locked) {
    sleep(lk, &lk->lk);
  }
  lk->locked = 1;
  lk->pid = myproc()->pid;
  release(&lk->lk);
}

void
releasesleep(struct sleeplock *lk)
{
  acquire(&lk->lk);
  lk->locked = 0;
  lk->pid = 0;
  wakeup(lk);
  release(&lk->lk);
}
```

Mark the statements as True/False w.r.t. this code.

True	False			
◉ ✖	○ ✔	Wakeup() will wakeup the first process waiting for the lock	✖	Wakeup() will wakeup all processes waiting for the lock

◉ ✖	○ ✔	acquire(&lk->lk);
		while (lk->locked) {
		sleep(lk, &lk->lk);
		}
		could also be written as
		acquire(&lk->lk);
		if (lk->locked) {
		sleep(lk, &lk->lk);
		}
			✖	loop is required because other process might have obtained the lock before this process returns from sleep().

True	False			
◉ ✗	○ ✓	A process has acquired the sleeplock when it comes out of sleep()	✗	
◉ ✗	○ ✓	sleep() is called holding a spinlock. This could be avoided by releasing the lock before calling sleep() and acquiring it again after call to sleep()	✗	
◉ ✓	○ ✗	sleep() is the function which blocks a process.	✓	
◉ ✓	○ ✗	The process which called acquiresleep() and then got blocked, is woken up by the timer interrupt	✓	it's woken up by another process which called releasesleep() and then wakeup()
◉ ✗	○ ✓	the 'spinlock lk' protects 'locked' variable, but not the 'name' nor the 'pid'	✗	
◉ ✓	○ ✗	the 'spinlock lk' is needed in a sleeplock, because access to the sleeplock for locking/unlocking itself creates a critical section	✓	
◉ ✓	○ ✗	The spinlock lk->lk is held when the process comes out of sleep()	✓	
◉ ✓	○ ✗	Sleeplock() will ensure that either the process gets the lock or the process gets blocked.	✓	
◉ ✓	○ ✗	All processes waiting for the sleeplock will have a race for aquiring lk->lk spinlock, because all are woken up	✓	wakeup() wakes up all processes, and they "thunder" to take the spinlock.

Wakeup() will wakeup the first process waiting for the lock: False

```
acquire(&lk->lk);
  while (lk->locked) {
    sleep(lk, &lk->lk);
  }
```

could also be written as

```
acquire(&lk->lk);
  if (lk->locked) {
    sleep(lk, &lk->lk);
  }: False
```

A process has acquired the sleeplock when it comes out of sleep(): False
sleep() is called holding a spinlock. This could be avoided by releasing the lock before calling sleep() and acquiring it again after call to sleep(): False
sleep() is the function which blocks a process.: True
The process which called acquiresleep() and then got blocked, is woken up by the timer interrupt: True
the 'spinlock lk' protects 'locked' variable, but not the 'name' nor the 'pid': False
the 'spinlock lk' is needed in a sleeplock, because access to the sleeplock for locking/unlocking itself creates a critical section: True
The spinlock lk->lk is held when the process comes out of sleep(): True
Sleeplock() will ensure that either the process gets the lock or the process gets blocked.: True
All processes waiting for the sleeplock will have a race for aquiring lk->lk spinlock, because all are woken up: True

Question **10**

Partially correct

Mark 0.50 out of 1.00

Mark the statements as True/False w.r.t. swtch()

True	False			
◉ ✔	○ ✖	p->context used in scheduler()->swtch() was **Generally** set when the process was interrupted earlier, and came via sched()->swtch()	✔	That's the only place when p->context is changed.
◉ ✔	○ ✖	swtch() is written in assembly language, because it violates calling convention, by changing the stack itself.	✔	
◉ ✖	○ ✔	swtch() is written in assembly language because it violates the calling convention by pushing parameters on the stack on its own.	✖	any function can push anything on stack, but remove it properly, that will not affect calling convention.
◉ ✖	○ ✔	swtch() called from scheduler() changes the stack from the process's kernel stack to the scheduler's kernel stack.	✖	it does reverse!
◉ ✔	○ ✖	swtch() is called only from sched() or scheduler()	✔	
◉ ✖	○ ✔	switch stores the old context on new stack, and restores new context from old stack.	✖	old goes on old, new comes from new stack
◉ ✖	○ ✔	sched() is the only place when p->context is set	✖	no. allocproc() also sets it.
◉ ✖	○ ✔	movl %esp, (%eax) means, *(c->scheduler) = contents of esp When swtch() is called from scheduler()	✖	No. it means c->scheduler = contents of esp.
◉ ✔	○ ✖	push in swtch() happens on old stack, while pop happens from new stack	✔	
◉ ✔	○ ✖	swtch() changes the context from "old" to "new"	✔	yeah, that's the definition

p->context used in scheduler()->swtch() was **Generally** set when the process was interrupted earlier, and came via sched()->swtch(): True
swtch() is written in assembly language, because it violates calling convention, by changing the stack itself.: True
swtch() is written in assembly language because it violates the calling convention by pushing parameters on the stack on its own.: False
swtch() called from scheduler() changes the stack from the process's kernel stack to the scheduler's kernel stack.: False
swtch() is called only from sched() or scheduler(): True
switch stores the old context on new stack, and restores new context from old stack.: False
sched() is the only place when p->context is set: False
movl %esp, (%eax)
means, *(c->scheduler) = contents of esp
When swtch() is called from scheduler(): False
push in swtch() happens on old stack, while pop happens from new stack: True
swtch() changes the context from "old" to "new": True

Question **11**

Correct

Mark 0.25 out of 0.25

Select the odd one out

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

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

Question **12**

Correct

Mark 0.25 out of 0.25

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

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

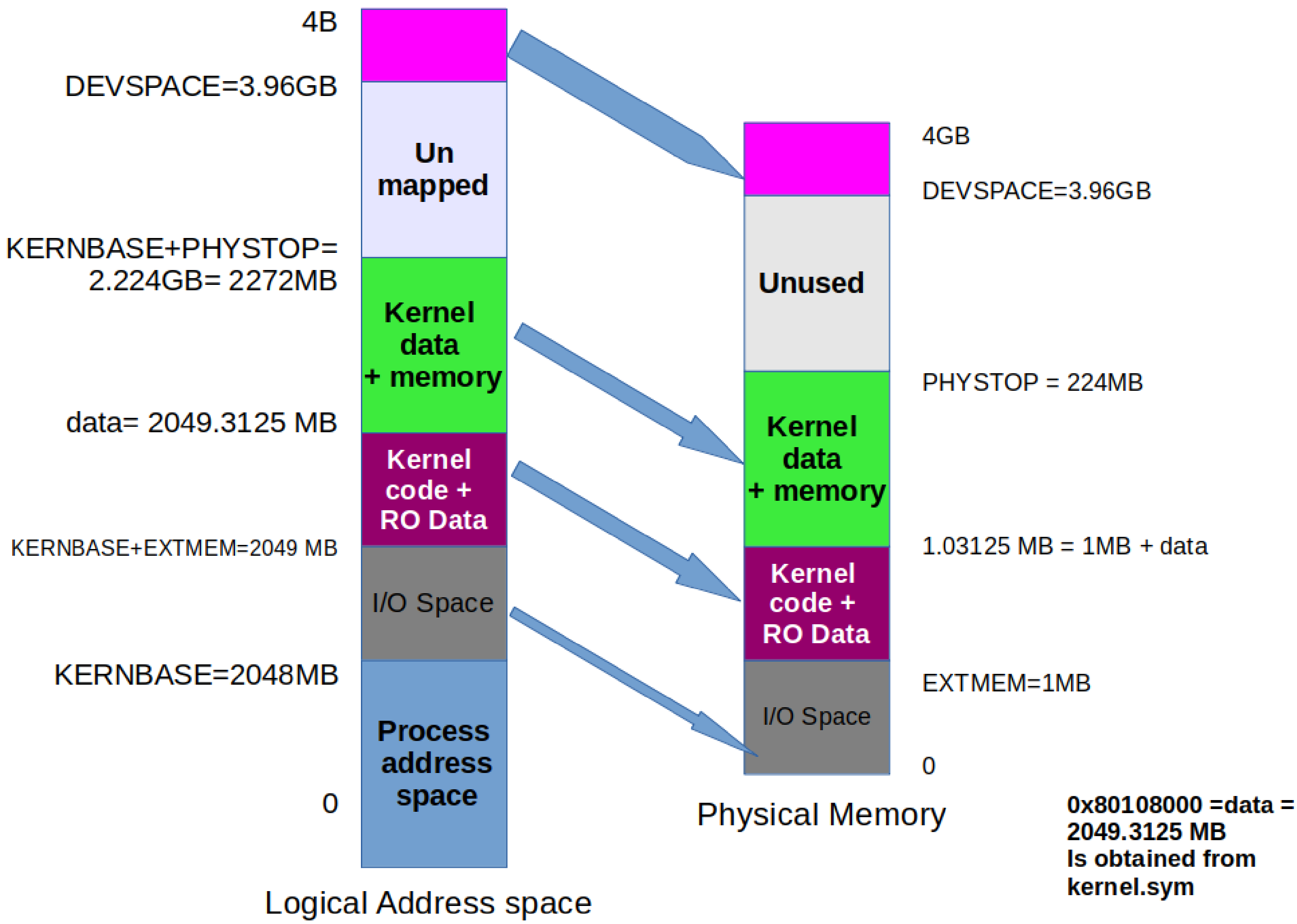
The correct answer is: 801154a8

Question **13**

Partially correct

Mark 0.36 out of 0.50

With respect to this diagram, mark statements as True/False.



True	False			
◉ ✓	○ ✗	This diagram only shows the absolutely defined virtual->physical mappings, not the mappings defined at run time by kernel.	✓	
○ ✓	◉ ✗	"Kernel data + memory" on right side, here refers to the region from which pages are allocated to the kernel and process both.	✗	"Kernel data + memory" on LEFT side, here refers to the virtual addresses of kernel used at run time.
◉ ✓	○ ✗	The kernel virtual addresses start from KERNLINK = KERNBASE + EXTMEM	✓	
◉ ✓	○ ✗	The process's pages are mapped into physical memory from 1.03125 MB to PHYSTOP.	✓	
◉ ✓	○ ✗	The kernel file, after compilation, has maximum virtual address up to "data" as shown in the diagram, which is equal to "end" variable	✓	

True	False		
◉ ✔	○ ✖	When bootloader loads the kernel, then physical memory from EXTMEM upto EXTMEM + data is occupied.	✔
○ ✔	◉ ✖	PHYSTOP can be changed , but that needs kernel recompilation and re-execution.	✖

This diagram only shows the absolutely defined virtual->physical mappings, not the mappings defined at run time by kernel.: True
"Kernel data + memory" on right side, here refers to the region from which pages are allocated to the kernel and process both.: True
The kernel virtual addresses start from KERNLINK = KERNBASE + EXTMEM: True
The process's pages are mapped into physical memory from 1.03125 MB to PHYSTOP.: True
The kernel file, after compilation, has maximum virtual address up to "data" as shown in the diagram, which is equal to "end" variable: True
When bootloader loads the kernel, then physical memory from EXTMEM upto EXTMEM + data is occupied.: True
PHYSTOP can be changed , but that needs kernel recompilation and re-execution.: True

Question **14**

Correct

Mark 0.25 out of 0.25

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

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

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

Question **15**

Correct

Mark 0.50 out of 0.50

We often use terms like "swtch() changes stack from process's kernel stack to scheduler's stack", or "the values are pushed on stack", or "the stack is initialized to the new page", etc. while discussing xv6 on x86.

Which of the following most accurately describes the meaning of "stack" in such sentences?

- a. The stack variable used in the program being discussed
- b. The region of memory where the kernel remembers all the function calls made
- c. The region of memory allocated by kernel for storing the parameters of functions
- d. The ss:esp pair
- e. The "stack" variable declared in "stack.S" in xv6
- f. the region of memory which is currently used as stack by processor
- g. The stack segment

Your answer is correct.

The correct answer is: The ss:esp pair

Question **16**

Partially correct

Mark 0.31 out of 0.50

Mark the statements as True/False, with respect to the use of the variable "chan" in struct proc.

True	False		
◉ ✔	○ ✘	The value of 'chan' is changed only in sleep()	✔
◉ ✘	○ ✔	when chan is NULL, the 'state' in proc must be RUNNABLE.	✘
◉ ✔	○ ✘	chan stores the address of the variable, representing a condition, for which the process is waiting.	✔
◉ ✔	○ ✘	'chan' is used only by the sleep() and wakeup1() functions.	✔
◉ ✘	○ ✔	chan is the head pointer to a linked list of processes, waiting for a particular event to occur	✘
◉ ✔	○ ✘	When chan is not NULL, the 'state' in struct proc must be SLEPING	✔
◉ ✔	○ ✘	in xv6, the address of an appropriate variable is used as a "condition" for a waiting process.	✔
◉ ✘	○ ✔	Changing the state of a process automatically changes the value of 'chan'	✘

The value of 'chan' is changed only in sleep(): True
when chan is NULL, the 'state' in proc must be RUNNABLE.: False
chan stores the address of the variable, representing a condition, for which the process is waiting.: True
'chan' is used only by the sleep() and wakeup1() functions.: True
chan is the head pointer to a linked list of processes, waiting for a particular event to occur: False
When chan is not NULL, the 'state' in struct proc must be SLEPING: True
in xv6, the address of an appropriate variable is used as a "condition" for a waiting process.: True
Changing the state of a process automatically changes the value of 'chan': False

Question **17**

Correct

Mark 0.25 out of 0.25

Match function with it's meaning

iderw	Issue a disk read/write for a buffer, block the issuing process
idewait	Wait for disc controller to be ready
idestart	tell disc controller to start I/O for the first buffer on idequeue
ideinit	Initialize the disc controller
ideintr	disk interrupt handler, transfer data from conroller to buffer, wake up processes waiting for this buffer, start I/O for next buffer

Your answer is correct.

The correct answer is: iderw → Issue a disk read/write for a buffer, block the issuing process, idewait → Wait for disc controller to be ready, idestart → tell disc controller to start I/O for the first buffer on idequeue, ideinit → Initialize the disc controller, ideintr → disk interrupt handler, transfer data from conroller to buffer, wake up processes waiting for this buffer, start I/O for next buffer

Question **18**

Correct

Mark 0.25 out of 0.25

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

- a. Because there is a limit on the values that the argumets to knit1() can take.
- b. call to seginit() makes it possible to actually use PHYSTOP in argument to kinit2()
- c. When kinit1() is called there is a need for few page frames, but later knit2() is called to serve need of more page frames
- 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 **19**

Correct

Mark 0.25 out of 0.25

Which of the following call sequence is impossible in xv6?

a. Process 1: write() -> sys_write()-> file_write() -- timer interrupt -> trap() -> yield() -> sched() -> switch() (jumps to)-> scheduler() ->swtch() (jumps to)-> Process 2 (return call sequence) sched() -> yield() -> trap-> user-code

b. Process 1: write() -> sys_write()-> file_write() -> writei() -> bread() -> bget() -> iderw() -> sleep() -> sched() -> switch() (jumps to)-> scheduler() ->swtch()(jumps to)-> Process 2 (return call sequence) sched() -> yield() -> trap-> user-code

c. Process 1: timer interrupt -> trap() -> yield() -> sched() -> switch() -> scheduler()-> Process 2 runs -> write -> sys_write() -> trap()-> ... ✔

Your answer is correct.

The correct answer is: Process 1: timer interrupt -> trap() -> yield() -> sched() -> switch() -> scheduler()-> Process 2 runs -> write -> sys_write() -> trap()-> ...

Question **20**

Partially correct

Mark 0.17 out of 0.50

Mark statements as True/False, w.r.t. the given diagram

Started on	Thursday, 2 February 2023, 9:01 PM
State	Finished
Completed on	Thursday, 2 February 2023, 10:55 PM
Time taken	1 hour 53 mins
Grade	**16.46** out of 20.00 (**82.28**%)

Question **1**

Complete

Mark 1.00 out of 1.00

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

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

Question **2**

Complete

Mark 1.00 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;
}
```

fd4	/tmp/2
fd2	/tmp/2
1	/tmp/3
fd1	/tmp/1
fd3	closed
0	/tmp/2
2	stderr

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

Question **3**

Complete

Mark 0.75 out of 1.00

Select the sequence of events that are NOT possible, assuming an interruptible kernel code

Select one or more:

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

- f. P1 running
  P1 makes system call
  timer interrupt
  Scheduler
  P2 running
  timer interrupt
  Scheuler
  P1 running
  P1's system call return

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

Question **4**

Complete

Mark 0.00 out of 1.00

Select all the correct statements about named pipes and ordinary(unnamed) pipe

Select one or more:

- [x] a. named pipes can be used between multiple processes but ordinary pipes can not be used
- [x] b. named pipe can be used between any processes
- [ ] c. both named and unnamed pipes require some kind of agreed protocol to be effectively used among multiple processes
- [x] d. named pipes are more efficient than ordinary pipes
- [x] e. ordinary pipe can only be used between related processes
- [x] f. a named pipe exists as a file on the file system
- [ ] g. named pipe exists even if the processes using it do exit()

The correct answers are: ordinary pipe can only be used between related processes, named pipe can be used between any processes, a named pipe exists as a file on the file system, named pipe exists even if the processes using it do exit(), both named and unnamed pipes require some kind of agreed protocol to be effectively used among multiple processes

Question **5**

Complete

Mark 4.50 out of 5.00

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[ 2 ][2];

    pipe( pfd[0] );
    pid1 = fork() ;
    if(pid1 != 0) {
        close(pfd[0] [0] );
        close( 1 );
        dup( pfd[0][1] );
        execl("/bin/ls", "/bin/ls", " -l ", NULL);
    }
    pipe( pfd[1] );
    pid2 = fork();
    if(pid2 == 0) {
        close( pfd[0][1] ;
        close(0);
        dup( pfd[1][0] );
        close(pfd[1] [0] );
        close( 1 );
        dup( pfd[1][1] );
        execl("/usr/bin/head", "/usr/bin/head", " -3 ", NULL);
    } else {
        close(pfd [1][1] );
        close( 0 );
        dup( pfd[1][0] );
        close(pfd [0][0] );
```

```
        execl("/usr/bin/tail", "/usr/bin/tail", "-1", NULL);
    }
}
```

Question **6**

Complete

Mark 1.00 out of 1.00

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)

Paging	one continuous chunk
Segmentation, then paging	many continuous chunks of variable size
Segmentation	many continuous chunks of variable size
Relocation + Limit	one continuous chunk

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

Question **7**

Complete

Mark 1.40 out of 2.00

Match the elements of C program to their place in memory

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

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

Question **8**

Complete

Mark 0.63 out of 1.00

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

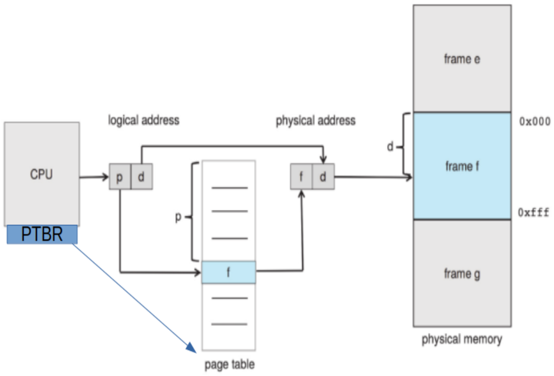


Figure 9.8 Paging hardware.

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

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

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

Question **9**

Complete

Mark 0.67 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 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
- [ ] b. P1 running
  P1 makes system call
  system call returns
  P1 running
  timer interrupt
  Scheduler running
  P2 running
- [x] c. P1 running
  P1 makes sytem call and blocks
  Scheduler
  P2 running
  P2 makes sytem call and blocks
  Scheduler
  P1 running again
- [ ] 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
- [x] e. P1 running
  P1 makes system call
  timer interrupt
  Scheduler

P2 running
timer interrupt
Scheuler
P1 running
P1's system call return

- [ ] f.

P1 running
P1 makes sytem call
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 **10**

Complete

Mark 1.00 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 is correct for > /tmp/ddd but not for 2>&1
- [ ] c. Program 1 is correct for > /tmp/ddd but not for 2>&1
- [ ] d. Program 2 ensures 2>&1 and does not ensure > /tmp/ddd
- [ ] e. Program 2 makes sure that there is one file offset used for '2' and '1'
- [ ] f. Program 1 ensures 2>&1 and does not ensure > /tmp/ddd
- [ ] g. Only Program 2 is correct
- [ ] h. Program 1 does 1>&2
- [x] i. Program 1 makes sure that there is one file offset used for '2' and '1'
- [ ] j. Program 2 does 1>&2
- [ ] k. Both program 1 and 2 are incorrect
- [x] l. Only Program 1 is correct

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 **11**

Complete

Mark 0.71 out of 1.00

Order the events that occur on a timer interrupt:

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

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

Question **12**

Complete

Mark 0.80 out of 1.00

Select all the correct statements about zombie processes

Select one or more:

- [ ] a. A process becomes zombie when it's parent finishes
- [x] b. A process becomes zombie when it finishes, and remains zombie until parent calls wait() on it
- [x] c. If the parent of a process finishes, before the process itself, then after finishing the process is typically attached to 'init' as parent
- [ ] d. Zombie processes are harmless even if OS is up for long time
- [x] e. A process can become zombie if it finishes, but the parent has finished before it
- [ ] f. A zombie process occupies space in OS data structures
- [x] g. init() typically keeps calling wait() for zombie processes to get cleaned up
- [ ] 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

Question **13**

Complete

Mark 1.00 out of 1.00

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 **14**

Complete

Mark 1.00 out of 1.00

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

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

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

Question **15**

Complete

Mark 1.00 out of 1.00

A process blocks itself means

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

◄ Random Quiz - 2: bootloader, system calls, fork-exec, open-read-write, linux-basics, processes

Started on	Thursday, 16 February 2023, 9:00 PM
State	Finished
Completed on	Thursday, 16 February 2023, 10:43 PM
Time taken	1 hour 43 mins
Grade	**13.00** out of 15.00 (**86.68**%)

Question **1**

Correct

Mark 1.00 out of 1.00

Which of the following statements is false ?

Select one:

- a. A process scheduling algorithm is preemptive if the CPU can be forcibly removed from a process.
- b. Time sharing systems generally use preemptive CPU scheduling.
- c. Real time systems generally use non preemptive CPU scheduling.
- d. Response time is more predictable in preemptive systems than in non preemptive systems.

Your answer is correct.

The correct answer is: Real time systems generally use non preemptive CPU scheduling.

Question **2**

Correct

Mark 1.00 out of 1.00

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

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

Your answer is correct.

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

Question **3**

Correct

Mark 1.00 out of 1.00

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)

Paging	one continuous chunk ✔
Relocation + Limit	one continuous chunk ✔
Segmentation	many continuous chunks of variable size ✔
Segmentation, then paging	many continuous chunks of variable size ✔

Your answer is correct.

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

Question **4**

Correct

Mark 1.00 out of 1.00

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 **5**

Correct

Mark 1.00 out of 1.00

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 **6**

Partially correct

Mark 0.47 out of 1.00

Select all the correct statements about zombie processes

Select one or more:

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

Your answer is partially correct.

You have correctly selected 4.
The correct answers are: A process becomes zombie when it finishes, and remains zombie until parent calls wait() on it, A process can become zombie if it finishes, but the parent has finished before it, A zombie process occupies space in OS data structures, If the parent of a process finishes, before the process itself, then after finishing the process is typically attached to 'init' as parent, init() typically keeps calling wait() for zombie processes to get cleaned up

Question 7

Partially correct

Mark 0.86 out of 1.00

Mark True/False

Statements about scheduling and scheduling algorithms

True	False			
◉ ✔	○ ✘	xv6 code does not care about Processor Affinity	✔	
○ ✔	◉ ✘	Statistical observations tell us that most processes have large number of small CPU bursts and relatively smaller numbers of large CPU bursts.	✘	
◉ ✔	○ ✘	Response time will be quite poor on non-interruptible kernels	✔	
◉ ✔	○ ✘	Generally the voluntary context switches are much more than non-voluntary context switches on a Linux system.	✔	
◉ ✔	○ ✘	Processor Affinity refers to memory accesses of a process being stored on cache of that processor	✔	
◉ ✔	○ ✘	A scheduling algorithm is non-premptive if it does context switch only if a process voluntarily relinquishes CPU or it terminates.	✔	
○ ✘	◉ ✔	On Linuxes the CPU utilisation is measured as the time spent in scheduling the idle thread	✔	It's the negation of this. Time NOT spent in idle thread.

xv6 code does not care about Processor Affinity: True
Statistical observations tell us that most processes have large number of small CPU bursts and relatively smaller numbers of large CPU bursts.: True
Response time will be quite poor on non-interruptible kernels: True
Generally the voluntary context switches are much more than non-voluntary context switches on a Linux system.: True
Processor Affinity refers to memory accesses of a process being stored on cache of that processor: True
A scheduling algorithm is non-premptive if it does context switch only if a process voluntarily relinquishes CPU or it terminates.: True
On Linuxes the CPU utilisation is measured as the time spent in scheduling the idle thread: False

Question **8**

Correct

Mark 1.00 out of 1.00

Mark whether the concept is related to scheduling or not.

Yes	No		
✗	✓ (selected)	file-table	✓
✓ (selected)	✗	context-switch	✓
✓ (selected)	✗	timer interrupt	✓
✓ (selected)	✗	ready-queue	✓
✓ (selected)	✗	runnable process	✓

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

Question **9**

Partially correct

Mark 0.50 out of 1.00

Select all the correct statements about signals

Select one or more:

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

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 **10**

Correct

Mark 1.00 out of 1.00

Map each signal with it's meaning

SIGPIPE	Broken Pipe	✔
SIGCHLD	Child Stopped or Terminated	✔
SIGUSR1	User Defined Signal	✔
SIGALRM	Timer Signal from alarm()	✔
SIGSEGV	Invalid Memory Reference	✔

The correct answer is: SIGPIPE → Broken Pipe, SIGCHLD → Child Stopped or Terminated, SIGUSR1 → User Defined Signal, SIGALRM → Timer Signal from alarm(), SIGSEGV → Invalid Memory Reference

Question **11**

Partially correct

Mark 0.80 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!

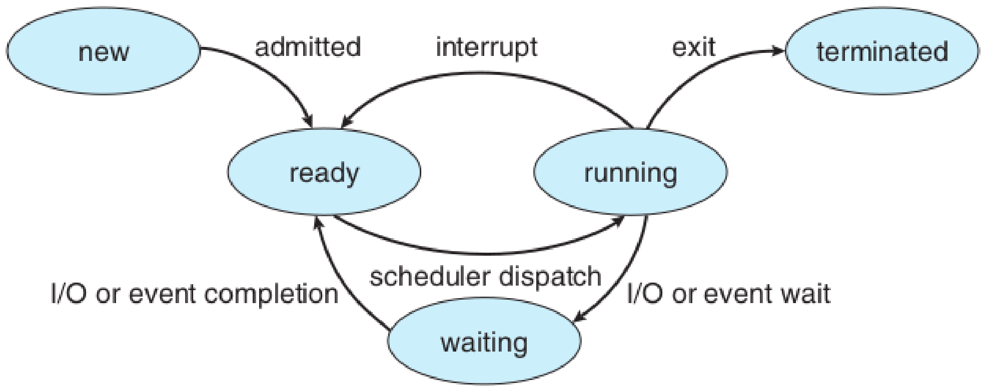


Figure 3.2 Diagram of process state.

True	False		
◉ ✓	○ ✗	Every forked process has to go through ZOMBIE state, at least for a small duration.	✔
◉ ✓	○ ✗	Only a process in READY state is considered by scheduler	✔
◉ ✓	○ ✗	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	✔
◉ ✗	○ ✓	A process only in RUNNING state can become TERMINATED because scheduler moves it to ZOMBIE state first	✖
○ ✗	◉ ✓	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
Only a process in READY state is considered by scheduler: True
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
A process only in RUNNING state can become TERMINATED because scheduler moves it to ZOMBIE state first: 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

Question **12**

Correct

Mark 1.00 out of 1.00

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

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

Your answer is correct.

The correct answers are: #directives, typedefs, global variables

Question **13**

Partially correct

Mark 1.38 out of 2.00

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

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

Your answer is partially correct.

You have correctly selected 8.
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 **14**

Correct

Mark 1.00 out of 1.00

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

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

Your answer is correct.

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

◄ Random Quiz - 3 (processes, memory management, event driven kernel), compilation-linking-loading, ipc-pipes

Jump to...

Random Quiz - 5: xv6 make, bootloader, interrupt handling, memory management ►

Started on	Thursday, 9 March 2023, 6:20 PM
State	Finished
Completed on	Thursday, 9 March 2023, 7:30 PM
Time taken	1 hour 10 mins
Overdue	14 mins
Grade	**5.55** out of 10.00 (**55.53**%)

Question **1**

Partially correct

Mark 0.18 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		
○ ✔	○ ✖	xv6 uses physical memory upto 224 MB only	✖
○ ✔	○ ✖	The stack allocated in entry.S is used as stack for scheduler's context for first processor	✖
○ ✔	○ ✖	The free page-frame are created out of nearly 222 MB	✖
○ ✔	○ ✖	The switchkvm() call in scheduler() is invoked after control comes to it from sched(), thus demanding execution in kernel's context	✖
○ ✔	○ ✖	PHYSTOP can be increased to some extent, simply by editing memlayout.h	✖
○ ✖	◉ ✔	The switchkvm() call in scheduler() changes CR3 to use page directory of new process	✔
○ ✔	○ ✖	The 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 kernel code and data take up less than 2 MB space	✖
○ ✖	◉ ✔	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	✖

xv6 uses physical memory upto 224 MB only: True
The stack allocated in entry.S is used as stack for scheduler's context for first processor: True
The free page-frame are created out of nearly 222 MB: True
The switchkvm() call in scheduler() is invoked after control comes to it from sched(), thus demanding execution in kernel's context: True
PHYSTOP can be increased to some extent, simply by editing memlayout.h: True
The switchkvm() call in scheduler() changes CR3 to use page directory of new process: False
The 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
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 swtch() scheduler(), thus demanding execution in new process's context: False
The switchkvm() call in scheduler() changes CR3 to use page directory kpgdir: True

Question **2**

Correct

Mark 1.00 out of 1.00

What's the trapframe in xv6?

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

Your answer is correct.

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

Question **3**

Partially correct

Mark 0.75 out of 1.00

Select the correct statements about interrupt handling in xv6 code

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

Your answer is partially correct.

You have correctly selected 6.
The correct answers are: All the 256 entries in the IDT are filled in xv6 code, Each entry in IDT essentially gives the values of CS and EIP to be used in handling that interrupt, xv6 uses the 64th entry in IDT for system calls, On any interrupt/syscall/exception the control first jumps in vectors.S, Before going to alltraps, the kernel stack contains upto 5 entries., The trapframe pointer in struct proc, points to a location on process's kernel stack, The function trap() is the called even if any of the hardware interrupt/system-call/exception occurs, The CS and EIP are changed only after pushing user code's SS,ESP on stack

Question **4**

Partially correct

Mark 0.55 out of 1.00

Consider the following command and it's output:

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

Following code in bootmain()

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

and following selected lines from Makefile

```
xv6.img: bootblock kernel
    dd if=/dev/zero of=xv6.img count=10000
    dd if=bootblock of=xv6.img conv=notrunc
    dd if=kernel of=xv6.img seek=1 conv=notrunc


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

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

- [ ] a. 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 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] c. 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(). ✓
- [ ] d. The kernel.asm file is the final kernel file
- [ ] e. Althought the size of the xv6.img file is ~5MB, only some part out of it is the bootloader+kernel code and remaining part is all zeroes.
- [x] f. The kernel is compiled by linking multiple .o files created from .c files; and the entry.o, initcode, entryother files ✓
- [x] g. The kernel.ld file contains instructions to the linker to link the kernel properly ✓
- [x] h. readseg() reads first 4k bytes of kernel in memory ✓
- [ ] i. The bootmain() code does not read the kernel completely in memory

Your answer is partially correct.

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

Question **5**

Correct

Mark 1.00 out of 1.00

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

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

Your answer is correct.

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

Question **6**

Correct

Mark 1.00 out of 1.00

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

- [ ] a. The code in assembly is required for transition to protected mode, from real mode; but calling convention was applicable all the time
- [x] 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
- [x] 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 correct.

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

Question **7**

Incorrect

Mark 0.00 out of 1.00

xv6.img: bootblock kernel

```
dd if=/dev/zero of=xv6.img count=10000
dd if=bootblock of=xv6.img conv=notrunc
dd if=kernel of=xv6.img seek=1 conv=notrunc
```

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

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

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 **8**

Partially correct

Mark 0.07 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 has only two program headers ✔
- [x] b. The kernel file gets loaded at the Physical address 0x10000 in memory. ✔
- [x] c. The elf->entry is set by the linker in the kernel file and it's 0x80000000 ✖
- [ ] d. The readseg finally invokes the disk I/O code using assembly instructions
- [ ] e. The elf->entry is set by the linker in the kernel file and it's 8010000c
- [ ] f. The kernel ELF file contains actual physical address where particular sections of 'kernel' file should be loaded
- [x] g. The kernel file in memory is not necessarily a continuously filled in chunk, it may have holes in it. ✔
- [x] h. The elf->entry is set by the linker in the kernel file and it's 0x80000000 ✖
- [ ] i. The kernel file gets loaded at the Physical address 0x10000 +0x80000000 in memory.

- [ ] 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 partially correct.

You have correctly selected 4.
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 **9**

Correct

Mark 1.00 out of 1.00

In bootasm.S, on the line

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

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

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

Your answer is correct.

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

Question **10**

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

ljmp $(SEG_KCODE<<3), $start32 in bootasm.S	0x7c00 to 0	✖
readseg((uchar*)elf, 4096, 0); in bootmain.c	Immaterial as the stack is not used here	✖
jmp *%eax in entry.S	0x7c00 to 0	✖
cli in bootasm.S	Choose...	
call bootmain in bootasm.S	Immaterial as the stack is not used here	✖

Your answer is incorrect.

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

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

Jump to...

Random Quiz - 6 (xv6 file system) ►

Started on	Friday, 31 March 2023, 6:18 PM
State	Finished
Completed on	Friday, 31 March 2023, 7:00 PM
Time taken	41 mins 52 secs
Grade	**4.92** out of 15.00 (**32.83**%)

Question **1**

Partially correct

Mark 0.67 out of 1.00

Select all the actions taken by iget()

- [x] a. Returns an inode with given dev+inode-number from cache, if it exists in cache✔
- [x] b. Returns the inode with reference count incremented✔
- [ ] c. Returns a free-inode , with dev+inode-number set, if not found in cache
- [ ] d. Panics if inode does not exist in cache
- [ ] e. Returns a valid inode if not found in cache
- [ ] f. Returns the inode with inode-cache lock held
- [ ] g. Returns the inode locked

Your answer is partially correct.

You have correctly selected 2.
The correct answers are: Returns an inode with given dev+inode-number from cache, if it exists in cache, Returns the inode with reference count incremented, Returns a free-inode , with dev+inode-number set, if not found in cache

Question **2**

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. ext4 is a log structured file system ✖ it's a journaled file system, not log structured
- [x] b. file system recovery recovers all the lost data ✖
- [x] c. log structured file systems considerably improve the recovery time ✔
- [ ] d. xv6 has a log structured file system
- [x] e. ext2 is by default a log structured file system ✖

Your answer is incorrect.

The correct answers are: xv6 has a log structured file system, log structured file systems considerably improve the recovery time

Question **3**

Partially correct

Mark 0.86 out of 2.00

Select T/F w.r.t physical disk handling in xv6 code

True	False		
◉ ✗	○ ☑	device files are not supported	✗
○ ☑	◉ ✗	The code supports IDE, and not SATA/SCSI	✗
◉ ☑	○ ✗	log is kept on the same device as the file system	✓
◉ ☑	○ ✗	disk driver handles only one buffer at a time	✓
◉ ☑	○ ✗	the superblock does not contain number of free blocks	✓
◉ ✗	○ ☑	only direct blocks are supported	✗
○ ☑	◉ ✗	only 2 disks are handled by default	✗

device files are not supported: False
The code supports IDE, and not SATA/SCSI: True
log is kept on the same device as the file system: True
disk driver handles only one buffer at a time: True
the superblock does not contain number of free blocks: True
only direct blocks are supported: False
only 2 disks are handled by default: True

Question **4**

Partially correct

Mark 1.00 out of 2.00

Marks the statements as True/False w.r.t. "struct buf"

True	False			
	● ✓	B_VALID means the buffer is empty and can be reused	✓	No. it means it contains data, same as the data on disk
✓	●	Lock on a buffer is acquired in bget, and released in brelse	✗	
● ✓		The reference count (refcnt) in struct buf is = number of processes accessing the buffer	✓	
●	✓	A buffer can have both B_VALID and B_DIRTY flags set	✗	only one will be set
●	✓	The "next" pointer chain gives the buffers in LRU order	✗	No. MRU order.
✓	●	A buffer can be both on the MRU/LRU list and also on idequeue list.	✗	
● ✓		The buffers are maintained in LRU order, in the function brelse	✓	
● ✓		B_DIRTY flag means the buffer contains modified data	✓	

B_VALID means the buffer is empty and can be reused: False
Lock on a buffer is acquired in bget, and released in brelse: True
The reference count (refcnt) in struct buf is = number of processes accessing the buffer: True
A buffer can have both B_VALID and B_DIRTY flags set: False
The "next" pointer chain gives the buffers in LRU order: False
A buffer can be both on the MRU/LRU list and also on idequeue list.: True
The buffers are maintained in LRU order, in the function brelse: True
B_DIRTY flag means the buffer contains modified data: True

Question **5**

Partially correct

Mark 0.40 out of 1.00

Arrange the following in their typical order of use in xv6.

1. ✔ iget
2. ✖ ilock
3. ✔ iunlock
4. ✖ iput
5. ✖ use inode

Your answer is partially correct.

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

Grade details: 2 / 5 = 40%

Here are the scores for each item in this response:

1. 1 / 1 = 100%
2. 0 / 1 = 0%
3. 1 / 1 = 100%
4. 0 / 1 = 0%
5. 0 / 1 = 0%

The correct order for these items is as follows:

1. iget
2. ilock
3. use inode
4. iunlock
5. iput

Question **6**

Partially correct

Mark 0.50 out of 1.00

Compare XV6 and EXT2 file systems.

Select True/False for each point.

True	False		
◉ ✗	○ ☑	In both ext2 and xv6, the superblock gives location of first inode block	✗
○ ☑	◉ ✗	xv6 contains journal, ext2 does not	✗
○ ✗	◉ ☑	Ext2 contains superblock but xv6 does not.	✓
○ ✗	◉ ☑	xv6 contains inode bitmap, but ext2 does not	✓
◉ ✗	○ ☑	Both xv6 and ext2 contain magic number	✗
◉ ☑	○ ✗	Ext2 contains group descriptors but xv6 does not	✓

In both ext2 and xv6, the superblock gives location of first inode block: False
xv6 contains journal, ext2 does not: True
Ext2 contains superblock but xv6 does not.: False
xv6 contains inode bitmap, but ext2 does not: False
Both xv6 and ext2 contain magic number: False
Ext2 contains group descriptors but xv6 does not: True

Question **7**

Incorrect

Mark 0.00 out of 1.00

Maximum size of a file on xv6 in **bytes** is

(just write a numeric answer)

Answer: 512 ✗

The correct answer is: 71680

Question **8**

Partially correct

Mark 0.17 out of 1.00

Select all the actions taken by ilock()

- [ ] a. Mark the in-memory inode as valid, if needed
- [x] b. Read the inode from disk, if needed ✔
- [ ] c. Take the sleeplock on the inode, always
- [ ] d. Get the inode from the inode-cache
- [x] e. Copy the on-disk inode into in-memory inode, if neeed ✔
- [x] f. Take the sleeplock on the inode, optionally ✖
- [ ] g. Lock all the buffers of the file in memory

Your answer is partially correct.

You have correctly selected 2.
The correct answers are: Read the inode from disk, if needed, Copy the on-disk inode into in-memory inode, if neeed, Take the sleeplock on the inode, always, Mark the in-memory inode as valid, if needed

Question **9**

Incorrect

Mark 0.00 out of 1.00

The lines

```
if(ip->type != T_DIR){
  iunlockput(ip);
  return 0;
}
```

in namex() function

mean

- [ ] a. The last path component (which is a file, and not a directory) has been resolved, so release the lock (using iunlockput) and return
- [x] b. One of the sub-components on the given path name, was a directory, but it was not supposed to be a directory, hence an error ✖
- [ ] c. There was a syntax error in the pathname specified
- [ ] d. One of the sub-components on the given path name, was not a directory, hence it's an error
- [x] e. One of the sub-components on the given path name, did not exist, hence it's an error ✖
- [x] f. ilock is held on the inode, and hence it's an error if it is a directory ✖
- [ ] g. No directory entry was found for the file to be opened, hence an error

Your answer is incorrect.

The correct answer is: One of the sub-components on the given path name, was not a directory, hence it's an error

Question **10**

Partially correct

Mark 0.50 out of 1.00

Suppose an application on xv6 does the following:

int main() {

char arr[128];

int fd = open("README, O_RDONLY);

read(fd, arrr, 100);

}

Assume that the code works.

Which of the following things are true about xv6 kernel code, w.r.t. the above C program.

True	False			
◉ ✗	○ ✓	The "memmove(dst, bp->data + off%BSIZE, m);" in readi() will copy the data from the disk to the kernel buffers	✗	It will transfer from kernel buffer to user memory (arr)
◉ ✓	○ ✗	The process will be made to sleep only once	✓	Yes. Reading 100 bytes means reading only one disk block. So bread()->iderw() is called only once.
◉ ✗	○ ✓	The data is transferred from disk to kernel buffers first, and then address of arr is maped to the kernel buffers	✗	No. data is copied into arr.
◉ ✓	○ ✗	The ONLY function that gets called on return devsw[ip->major].read(ip, dst, n); is consoleread	✓	There is no other device file in xv6
○ ✗	○ ✓	The loop in readi() will always read a different block using bread()	✗	
◉ ✓	○ ✗	value of fd will be 3	✓	

The "memmove(dst, bp->data + off%BSIZE, m);" in readi() will copy the data from the disk to the kernel buffers: False
The process will be made to sleep only once: True
The data is transferred from disk to kernel buffers first, and then address of arr is maped to the kernel buffers: False
The ONLY function that gets called on return devsw[ip->major].read(ip, dst, n); is consoleread: True
The loop in readi() will always read a different block using bread(): False
value of fd will be 3: True

Question **11**

Partially correct

Mark 0.83 out of 1.00

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

ideintr	disk driver	✔
filestat()	file descriptor	✔
ialloc	inode	✔
bread	buffer cache	✔
balloc	system call	✖
skipelem	pathname lookup	✔
bmap	pathname lookup	✖
sys_chdir()	system call	✔
stati	inode	✔
commit	logging	✔
namei	pathname lookup	✔
dirlookup	directory	✔

Your answer is partially correct.

You have correctly selected 10.

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

Question **12**

Incorrect

Mark 0.00 out of 1.00

Match function with it's functionality

nameiparent	Write a new entry in a given directory	✖
dirlink	Link a directory with another directory	✖
namex	Search a given name in a given directory	✖
dirlookup	Lookup (search) for a given directory	✖

Your answer is incorrect.

The correct answer is: nameiparent → return in-memory inode for parent directory of a given pathname, dirlink → Write a new entry in a given directory, namex → return in-memory inode for a given pathname, dirlookup → Search a given name in a given directory

Question **13**

Incorrect

Mark 0.00 out of 1.00

An inode is read from disk as a part of this function

- a. ilock
- b. sys_read ✖
- c. iget
- d. readi
- e. iread

Your answer is incorrect.

The correct answer is: ilock

◄ Random Quiz - 5: xv6 make, bootloader, interrupt handling, memory management

Jump to...

(Random Quiz - 7) Pre-Endsem Quiz ►

Started on	Wednesday, 19 April 2023, 6:16 PM
State	Finished
Completed on	Wednesday, 19 April 2023, 8:31 PM
Time taken	2 hours 14 mins
Overdue	14 mins 47 secs
Grade	**14.08** out of 30.00 (**46.92**%)

Question **1**

Incorrect

Mark 0.00 out of 1.00

Select all correct statements about file system recovery (without journaling) programs e.g. fsck

Select one or more:

- [x] a. Recovery programs are needed only if the file system has a delayed-write policy.✔
- [ ] b. Recovery is possible due to redundancy in file system data structures
- [x] c. It is possible to lose data as part of recovery✔
- [x] d. A recovery program, most typically, builds the file system data structure and checks for inconsistencies✔
- [x] e. Recovery programs recalculate most of the metadata summaries (e.g. free inode count)✔
- [x] f. They are used to recover deleted files✖
- [x] g. They may take very long time to execute✔
- [ ] h. Even with a write-through policy, it is possible to need a recovery program.
- [ ] i. They can make changes to the on-disk file system

Your answer is incorrect.

The correct answers are: Recovery is possible due to redundancy in file system data structures, A recovery program, most typically, builds the file system data structure and checks for inconsistencies, It is possible to lose data as part of recovery, They may take very long time to execute, They can make changes to the on-disk file system, Recovery programs recalculate most of the metadata summaries (e.g. free inode count), Recovery programs are needed only if the file system has a delayed-write policy., Even with a write-through policy, it is possible to need a recovery program.

Question **2**

Partially correct

Mark 0.70 out of 1.00

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

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

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

Question **3**

Partially correct

Mark 0.25 out of 1.00

Match each suggested semaphore implementation (discussed in class)

with the problems that it faces

```
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) {
                spinunlock(&(s->sl));
                spinlock(&(s->sl));
        }
        (s->val)--;
        spinunlock(&(s->sl));
}
```

too much spinning, bounded wait not guaranteed ✔

```
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));
}
```

blocks holding a spinlock ✘

```
struct semaphore {
        int val;
        spinlock lk;
        list l;
};
sem_init(semaphore *s, int initval) {
        s->val = initval;
        s->sl = 0;
}
block(semaphore *s) {
        listappend(s->l, current);
        spinunlock(&(s->sl));
        schedule();
}
wait(semaphore *s) {
        spinlock(&(s->sl));
        while(s->val <=0) {
                block(s);
        }
        (s->val)--;
        spinunlock(&(s->sl));
}
signal(seamphore *s) {
        spinlock(*(s->sl));
        (s->val)++;
        x = dequeue(s->sl) and enqueue(readyq, x);
        spinunlock(*(s->sl));
}
```

too much spinning, bounded wait not guaranteed ✖

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

deadlock ✖

Your answer is partially correct.

You have correctly selected 1.
The correct answer is:

```
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) {
                spinunlock(&(s->sl));
                spinlock(&(s->sl));
        }
        (s->val)--;
        spinunlock(&(s->sl));
}
```

→ too much spinning, bounded wait not guaranteed,

```
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));
}
```

→ deadlock,

```
struct semaphore {
        int val;
        spinlock lk;
        list l;
};
sem_init(semaphore *s, int initval) {
        s->val = initval;
        s->sl = 0;
}
block(semaphore *s) {
        listappend(s->l, current);
        spinunlock(&(s->sl));
        schedule();
}
wait(semaphore *s) {
        spinlock(&(s->sl));
        while(s->val <=0) {
                block(s);
        }
        (s->val)--;
        spinunlock(&(s->sl));
}
signal(seamphore *s) {
        spinlock(*(s->sl));
        (s->val)++;
        x = dequeue(s->sl) and enqueue(readyq, x);
        spinunlock(*(s->sl));
}
```

→ not holding lock after unblock,

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

→ blocks holding a spinlock

Question **4**

Partially correct

Mark 0.88 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		
◉ ✔	○ ✖	The arguments to system call originally reside on process stack.	✔
◉ ✔	○ ✖	String arguments are NOT copied in kernel memory, but just pointed to by a kernel memory pointer	✔
◉ ✔	○ ✖	The arguments are accessed in the kernel code using esp on the trapframe.	✔
○ ✖	◉ ✔	Integer arguments are stored in eax, ebx, ecx, etc. registers	✔
◉ ✔	○ ✖	Integer arguments are copied from user memory to kernel memory using argint()	✔
○ ✖	◉ ✔	String arguments are first copied to trapframe and then from trapframe to kernel's other variables.	✔
◉ ✔	○ ✖	The functions like argint(), argstr() make the system call arguments available in the kernel.	✔
◉ ✖	○ ✔	The arguments to system call are copied to kernel stack in trapasm.S	✖

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

Question **5**

Correct

Mark 1.00 out of 1.00

Map the technique with it's feature/problem

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

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

Question **6**

Partially correct

Mark 0.10 out of 2.00

Compare paging with demand paging and select the correct statements.

Select one or more:

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

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 **7**

Correct

Mark 1.00 out of 1.00

Given that the memory access time is 150 ns, probability of a page fault is 0.9 and page fault handling time is 6 ms,
The effective memory access time in nanoseconds is:

Answer: 5400015

The correct answer is: 5400015.00

Question **8**

Correct

Mark 1.00 out of 1.00

Assuming a 8- KB page size, what is the page numbers for the address 803160 reference in decimal :

(give answer also in decimal)

Answer: 98

The correct answer is: 98

Question **9**

Partially correct

Mark 1.50 out of 2.00

For Virtual File System to work, which of the following changes are required to be done to an existing OS code (e.g. xv6)?

- [x] a. Each file-system writer needs to provide the set of function pointers for VFS, and these function pointers need to be setup in generic inode of "/" of that file system during mount() ✔
- [x] b. A mount() system call should be provided to mount a partition onto some directory in existing namespace rooted at "/" ✔
- [ ] c. The operating system in-memory inode needs to be a generic-inode representing "inode" like data structure across multiple file systems.
- [x] d. The generic inode needs to have a field representing if this inode is a mount point and also to refer/point to the root of the mounted file system's inode. ✔
- [x] e. The filesystem related system calls (e.g. read, write) need to invoke the file system specific functions (e.g. ext2_read, ext2_write, ntfs_read, ntfs_write) using function pointers. ✔
- [ ] f. The file system specific function pointers, for file system system-calls, need to be setup in the generic inode during lookup.
- [x] g. Each open() needs to copy the function pointers from the inode of the parent directory into the inode of the child (if not already done), unless it's traversing a mount point. (This may be done as part of lookup() which is called by open()) ✔
- [x] h. The lookup() operation needs to check if it's crossing a mount point and call FS specific operations to read inodes/directories ✔

The correct answers are: A mount() system call should be provided to mount a partition onto some directory in existing namespace rooted at "/", The filesystem related system calls (e.g. read, write) need to invoke the file system specific functions (e.g. ext2_read, ext2_write, ntfs_read, ntfs_write) using function pointers., The file system specific function pointers, for file system system-calls, need to be setup in the generic inode during lookup., The operating system in-memory inode needs to be a generic-inode representing "inode" like data structure across multiple file systems., The generic inode needs to have a field representing if this inode is a mount point and also to refer/point to the root of the mounted file system's inode., The lookup() operation needs to check if it's crossing a mount point and call FS specific operations to read inodes/directories, Each file-system writer needs to provide the set of function pointers for VFS, and these function pointers need to be setup in generic inode of "/" of that file system during mount(), Each open() needs to copy the function pointers from the inode of the parent directory into the inode of the child (if not already done), unless it's traversing a mount point. (This may be done as part of lookup() which is called by open())

Question **10**

Partially correct

Mark 0.33 out of 1.00

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

```
void
yield(void)
{
...
release(&ptable.lock);
}
```

Release the lock held by some another process ✔

```
void
panic(char *s)
{
...
  panicked = 1;
```

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

```
void
acquire(struct spinlock *lk)
{
...
  getcallerpcs(&lk, lk->pcs);
```

Disable interrupts to avoid another process's pointer being returned ✖

Your answer is partially correct.

You have correctly selected 1.
The correct answer is: void
yield(void)
{
...
release(&ptable.lock);
} → Release the lock held by some another process, void
panic(char *s)
{
...
  panicked = 1; → Ensure that no printing happens on other processors, void
acquire(struct spinlock *lk)
{
...
  getcallerpcs(&lk, lk->pcs); → Traverse ebp chain to get sequence of instructions followed in functions calls

Question **11**

Partially correct

Mark 0.75 out of 1.00

Select all correct statements about journalling (logging) in file systems like ext3

Select one or more:

- [ ] a. A different device driver is always needed to access the journal
- [x] b. The purpose of journal is to speed up file system recovery ✔
- [x] c. Most typically a transaction in journal is recorded atomically (full or none) ✔
- [ ] d. Journals are often stored circularly
- [ ] e. Journals must be maintained on the same device that hosts the file system
- [x] f. the journal contains a summary of all changes made as part of a single transaction ✔
- [ ] g. Journal is hosted in the same device that hosts the swap space

Your answer is partially correct.

You have correctly selected 3.
The correct answers are: The purpose of journal is to speed up file system recovery, the journal contains a summary of all changes made as part of a single transaction, Most typically a transaction in journal is recorded atomically (full or none), Journals are often stored circularly

Question **12**

Partially correct

Mark 1.00 out of 2.00

Select all the correct statements about synchronization primitives.

Select one or more:

- [ ] a. Thread that is going to block should not be holding any spinlock
- [x] b. Semaphores can be used for synchronization scenarioes like ordered execution ✔
- [x] c. Mutexes can be implemented using spinlock ✔
- [ ] d. Blocking means one process passing over control to another process
- [ ] e. Semaphores are always a good substitute for spinlocks
- [ ] f. Blocking means moving the process to a wait queue and spinning
- [ ] g. Blocking means moving the process to a wait queue and calling scheduler
- [ ] h. All synchronization primitives are implemented essentially with some hardware assistance.
- [ ] i. Spinlocks are good for multiprocessor scenarios, for small critical sections
- [ ] j. Mutexes can be implemented without any hardware assistance
- [x] k. Spinlocks consume CPU time ✔
- [x] l. Mutexes can be implemented using blocking and wakeup ✔

Your answer is partially correct.

You have correctly selected 4.
The correct answers are: Spinlocks are good for multiprocessor scenarios, for small critical sections, Spinlocks consume CPU time, Semaphores can be used for synchronization scenarioes like ordered execution, Mutexes can be implemented using spinlock, Mutexes can be implemented using blocking and wakeup, Thread that is going to block should not be holding any spinlock, Blocking means moving the process to a wait queue and calling scheduler, All synchronization primitives are implemented essentially with some hardware assistance.

Question **13**

Partially correct

Mark 0.50 out of 2.00

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

"..." means some code.

```
void
yield(void)
{
...
release(&ptable.lock);
}
```

Ensure that no printing happens on other processors

✖

```
struct proc*
myproc(void) {
...
  pushcli();
  c = mycpu();
  p = c->proc;
  popcli();
...
}
```

Disable interrupts to avoid another process's pointer being returned

✔

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

Avoid a self-deadlock

✖

```
void
acquire(struct spinlock *lk)
{
...
 getcallerpcs(&lk, lk->pcs);
```

Disable interrupts to avoid deadlocks

✖

```
void
acquire(struct spinlock *lk)
{
 pushcli();
```

Traverse ebp chain to get sequence of instructions followed in functions calls

✖

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

Release the lock held by some another process

✖

```
void
panic(char *s)
{
...
 panicked = 1;
```

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

✖

Your answer is partially correct.

You have correctly selected 2.

The correct answer is: void
yield(void)
{
...
release(&ptable.lock);
}

→ Release the lock held by some another process, struct proc*
myproc(void) {
...
 pushcli();
 c = mycpu();
 p = c->proc;
 popcli();
...
}

→ Disable interrupts to avoid another process's pointer being returned, 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
acquire(struct spinlock *lk)
{
...
__sync_synchronize();
```

→ Tell compiler not to reorder memory access beyond this line,

```
void
acquire(struct spinlock *lk)
{
...
  getcallerpcs(&lk, lk->pcs);
```

→ Traverse ebp chain to get sequence of instructions followed in functions calls,

```
void
acquire(struct spinlock *lk)
{
  pushcli();
```

→ Disable interrupts to avoid deadlocks,

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

→ Avoid a self-deadlock,

```
void
panic(char *s)
{
...
  panicked = 1;
```

→ Ensure that no printing happens on other processors

Question **14**

Partially correct

Mark 0.86 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			
◉ ✔	○ ✖	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
○ ✔	○ ✖	A logical volume may span across multiple physical volumes	✖	
◉ ✔	○ ✖	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	✔	
◉ ✔	○ ✖	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	✔	
◉ ✔	○ ✖	The volume manager stores additional metadata on the physical disk partitions	✔	

A logical volume may span across multiple physical partitions: True
A logical volume may span across multiple physical volumes: 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
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
The volume manager stores additional metadata on the physical disk partitions: True

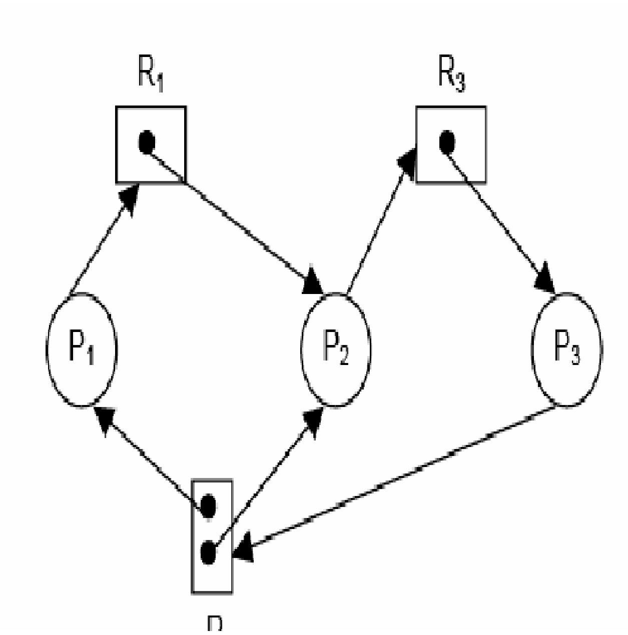
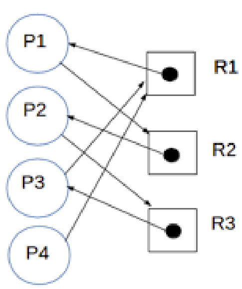
Question **15**

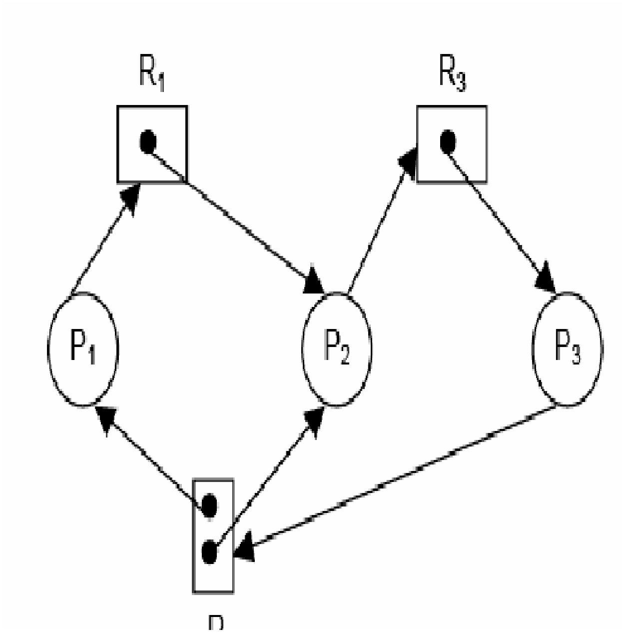
Partially correct

Mark 0.50 out of 1.00

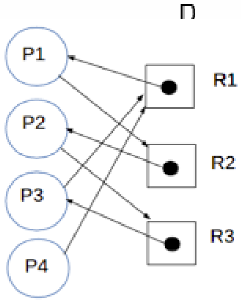
For each of the resource allocation diagram shown,

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

Yes	No		
			
			



: Yes



: Yes

Question **16**

Partially correct

Mark 0.80 out of 1.00

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

Select one or more:

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

Your answer is partially correct.

You have correctly selected 4.
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 **17**

Incorrect

Mark 0.00 out of 1.00

Match the code with it's functionality

S1 = 0; S2 = 0;
P2:
Statement1;
Signal(S2);

P1:
Wait(S2);
Statemetn2;
Signal(S1);

P3:
Wait(S1);
Statement S3;

S1 = 0; S2 = 0; P2: Statement1; Signal(S2); P1: Wait(S2); Statemetn2; Signal(S1); P3: Wait(S1); Statement S3;	Execution order P1, P2, P3 ✖
S = 0 P1: Statement1; Signal(S) P2: Wait(S) Statment2;	Execution order P3, P2, P1 ✖
S = 5 Wait(S) Critical Section Signal(S)	Execution order P2, then P1 ✖
S = 1 Wait(S) Critical Section Signal(S);	Counting semaphore ✖

Your answer is incorrect.

The correct answer is: S1 = 0; S2 = 0;
P2:
Statement1;
Signal(S2);

P1:
Wait(S2);
Statemetn2;
Signal(S1);

P3:
Wait(S1);
Statement S3; → Execution order P2, P1, P3, S = 0
P1:
Statement1;

Signal(S)

P2:
Wait(S)
Statment2; → Execution order P1, then P2, S = 5
Wait(S)
Critical Section
Signal(S) → Counting semaphore, S = 1
Wait(S)
Critical Section
Signal(S); → Binary Semaphore for mutual exclusion

Question **18**

Not answered

Marked out of 2.00

Write all changes required to xv6 to add a buddy allocator.

Every change should be mentioned in terms of either of the following:

(a) pseudo-code of new function to be added

(b) prototype of any new function or new system call to be added

(c) pseudo-code of changes to an existing function, describing lines to be removed, and lines to be added

(d) **precise** declaration of new data structures to be added in C, or changes to the existing data structure

(e) Name and a one-line description of new userland functionality to be added

(f) Changes to Makefile

(g) Any other change in a maximum of 20 words per change.

Question **19**

Correct

Mark 1.00 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.

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

Your answer is correct.

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 **20**

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:

11010010

Now, there is a request for a chunk of 45 bytes.

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

Answer: 11001110 ✖

The correct answer is: 11011110

Question **21**

Complete

Mark 0.25 out of 3.00

List down all changes required to xv6 code, in order to add the system call chown().

Every change should be mentioned in terms of either of the following:

(a) pseudo-code of new function to be added

(b) prototype of any new function or new system call to be added

(c) pseudo-code of changes to an existing function, describing lines to be removed, and lines to be added

(d) **precise** declaration of new data structures to be added in C, or changes to the existing data structure

(e) Name and a one-line description of new userland functionality to be added

(f) Changes to Makefile

(g) Any other change in a maximum of 20 words per change.

(a)int chown(char* path, char *owner_name);

d) there is no need to change the data structure and no need to add new data structure

(e)allow user to change the owner of the file

(f)_chown\

Comment:

Question **22**

Correct

Mark 1.00 out of 1.00

Calculate the average waiting time using
FCFS scheduling
for the following workload

assuming that they arrive in this order during the first time unit:

Process	Burst Time
P1	2
P2	6
P3	2
P4	3

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

Answer: 5 ✔

P2 waits for 2 units

P3 waits for 2+6 units

P4 waits for 2 + 6 +2 units of time

Total waiting = 2 + 2 + 6 + 2 + 6 + 2 = 20 units

Average waiting time = 20/4 = 5

The correct answer is: 5

Question **23**

Partially correct

Mark 0.67 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.

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

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

Started on	Wednesday, 19 April 2023, 8:46 PM
State	Finished
Completed on	Wednesday, 19 April 2023, 9:26 PM
Time taken	39 mins 31 secs
Grade	**15.60** out of 30.00 (**51.99**%)

Question **1**

Incorrect

Mark 0.00 out of 2.00

Select all the correct statements about synchronization primitives.

Select one or more:

- [ ] a. Blocking means moving the process to a wait queue and calling scheduler
- [x] b. Blocking means moving the process to a wait queue and spinning ✖
- [x] c. Semaphores are always a good substitute for spinlocks ✖
- [ ] d. Spinlocks are good for multiprocessor scenarios, for small critical sections
- [ ] e. All synchronization primitives are implemented essentially with some hardware assistance.
- [x] f. Blocking means one process passing over control to another process ✖
- [x] g. Semaphores can be used for synchronization scenarioes like ordered execution ✔
- [ ] h. Mutexes can be implemented without any hardware assistance
- [ ] i. Spinlocks consume CPU time
- [ ] j. Mutexes can be implemented using spinlock
- [ ] k. Thread that is going to block should not be holding any spinlock
- [ ] l. Mutexes can be implemented using blocking and wakeup

Your answer is incorrect.

The correct answers are: Spinlocks are good for multiprocessor scenarios, for small critical sections, Spinlocks consume CPU time, Semaphores can be used for synchronization scenarioes like ordered execution, Mutexes can be implemented using spinlock, Mutexes can be implemented using blocking and wakeup, Thread that is going to block should not be holding any spinlock, Blocking means moving the process to a wait queue and calling scheduler, All synchronization primitives are implemented essentially with some hardware assistance.

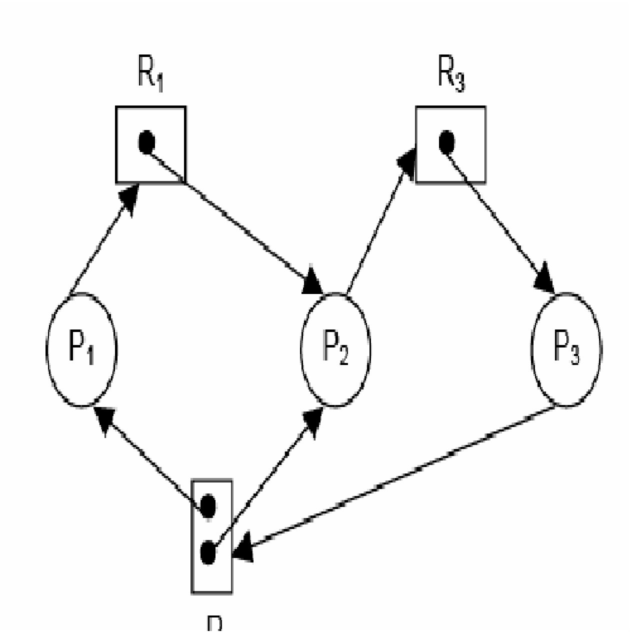
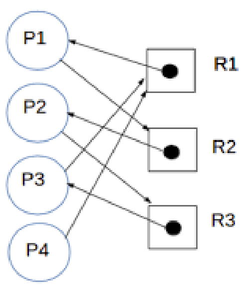
Question **2**

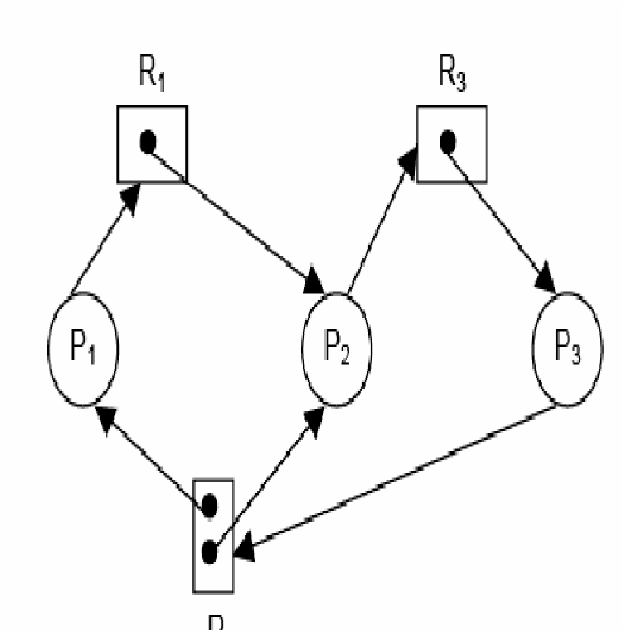
Partially correct

Mark 0.50 out of 1.00

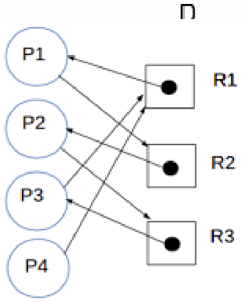
For each of the resource allocation diagram shown,

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

Yes	No		
◉ ✔	○ ✖		✔
○ ✔	◉ ✖		✖



: Yes



: Yes

Question **3**

Partially correct

Mark 0.25 out of 1.00

Match each suggested semaphore implementation (discussed in class)

with the problems that it faces

```
struct semaphore {
        int val;
        spinlock lk;
        list l;
};
sem_init(semaphore *s, int initval) {
        s->val = initval;
        s->sl = 0;
}
block(semaphore *s) {
        listappend(s->l, current);
        spinunlock(&(s->sl));
        schedule();
}
wait(semaphore *s) {
        spinlock(&(s->sl));
        while(s->val <=0) {
                block(s);
        }
        (s->val)--;
        spinunlock(&(s->sl));
}
signal(seamphore *s) {
        spinlock(*(s->sl));
        (s->val)++;
        x = dequeue(s->sl) and enqueue(readyq, x);
        spinunlock(*(s->sl));
}
```

deadlock ✖

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

not holding lock after unblock ✖

```
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) {
                spinunlock(&(s->sl));
                spinlock(&(s->sl));
        }
        (s->val)--;
        spinunlock(&(s->sl));
}
```

too much spinning, bounded wait not guaranteed ✔

```
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));
}
```

blocks holding a spinlock ✖

Your answer is partially correct.

You have correctly selected 1.
The correct answer is:

```
struct semaphore {
        int val;
        spinlock lk;
        list l;
};
sem_init(semaphore *s, int initval) {
        s->val = initval;
        s->sl = 0;
}
block(semaphore *s) {
        listappend(s->l, current);
        spinunlock(&(s->sl));
        schedule();
}
wait(semaphore *s) {
        spinlock(&(s->sl));
        while(s->val <=0) {
                block(s);
        }
        (s->val)--;
        spinunlock(&(s->sl));
}
signal(seamphore *s) {
        spinlock(*(s->sl));
        (s->val)++;
        x = dequeue(s->sl) and enqueue(readyq, x);
        spinunlock(*(s->sl));
}
```

→ not holding lock after unblock,

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

→ blocks holding a spinlock,

```
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) {
                spinunlock(&(s->sl));
                spinlock(&(s->sl));
        }
        (s->val)--;
        spinunlock(&(s->sl));
}
```

→ too much spinning, bounded wait not guaranteed,

```
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));
}
```

→ deadlock

Question **4**

Partially correct

Mark 0.75 out of 1.00

Match the code with it's functionality

S = 1
Wait(S)
Critical Section
Signal(S);

Binary Semaphore for mutual exclusion	✔

S = 0
P1:
Statement1;
Signal(S)

Execution order P1, then P2	✔

P2:
Wait(S)
Statment2;

S = 5
Wait(S)
Critical Section
Signal(S)

Execution order P1, P2, P3	✖

S1 = 0; S2 = 0;
P2:
Statement1;
Signal(S2);

P1:
Wait(S2);
Statemetn2;
Signal(S1);

Execution order P2, P1, P3	✔

P3:
Wait(S1);
Statement S3;

Your answer is partially correct.

You have correctly selected 3.
The correct answer is: S = 1
Wait(S)
Critical Section
Signal(S); → Binary Semaphore for mutual exclusion, S = 0
P1:
Statement1;
Signal(S)

P2:
Wait(S)
Statment2; → Execution order P1, then P2, S = 5
Wait(S)
Critical Section
Signal(S) → Counting semaphore, S1 = 0; S2 = 0;

P2:
Statement1;
Signal(S2);

P1:
Wait(S2);
Statemetn2;
Signal(S1);

P3:
Wait(S1);
Statement S3; → Execution order P2, P1, P3

Question **5**

Partially correct

Mark 1.71 out of 2.00

Compare paging with demand paging and select the correct statements.

Select one or more:

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

Your answer is partially correct.

You have correctly selected 6.
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 **6**

Partially correct

Mark 0.33 out of 1.00

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

```
void
acquire(struct spinlock *lk)
{
...
  getcallerpcs(&lk, lk->pcs);
```

Disable interrupts to avoid another process's pointer being returned ✖

```
void
panic(char *s)
{
...
  panicked = 1;
```

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

```
void
yield(void)
{
...
release(&ptable.lock);
}
```

Release the lock held by some another process ✔

Your answer is partially correct.

You have correctly selected 1.
The correct answer is: void
acquire(struct spinlock *lk)
{
...
  getcallerpcs(&lk, lk->pcs); → Traverse ebp chain to get sequence of instructions followed in functions calls, void
panic(char *s)
{
...
  panicked = 1; → Ensure that no printing happens on other processors, void
yield(void)
{
...
release(&ptable.lock);
} → Release the lock held by some another process

Question **7**

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.

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

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

Question **8**

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

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

Question **9**

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:

11010010

Now, there is a request for a chunk of 45 bytes.

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

Answer: 11001101 ✖

The correct answer is: 11011110

Question **10**

Not answered

Marked out of 2.00

Write all changes required to xv6 to add a buddy allocator.

Every change should be mentioned in terms of either of the following:

(a) pseudo-code of new function to be added

(b) prototype of any new function or new system call to be added

(c) pseudo-code of changes to an existing function, describing lines to be removed, and lines to be added

(d) **precise** declaration of new data structures to be added in C, or changes to the existing data structure

(e) Name and a one-line description of new userland functionality to be added

(f) Changes to Makefile

(g) Any other change in a maximum of 20 words per change.

Question **11**

Partially correct

Mark 0.80 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. one-one model can be implemented even if there are no kernel threads
- [x] c. all three models, that is many-one, one-one, many-many , require a user level thread library ✔
- [x] d. many-one model gives no speedup on multicore processors ✔
- [x] e. one-one model increases kernel's scheduling load ✔
- [ ] f. A process blocks in many-one model even if a single thread makes a blocking system call
- [ ] g. A process may not block in many-one model, if a thread makes a blocking system call

Your answer is partially correct.

You have correctly selected 4.
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 **12**

Partially correct

Mark 0.50 out of 1.00

Select all correct statements about journalling (logging) in file systems like ext3

Select one or more:

- [x] a. the journal contains a summary of all changes made as part of a single transaction ✔
- [ ] b. Journals are often stored circularly
- [ ] c. Journals must be maintained on the same device that hosts the file system
- [x] d. The purpose of journal is to speed up file system recovery ✔
- [ ] e. Most typically a transaction in journal is recorded atomically (full or none)
- [ ] f. Journal is hosted in the same device that hosts the swap space
- [ ] g. A different device driver is always needed to access the journal

Your answer is partially correct.

You have correctly selected 2.

The correct answers are: The purpose of journal is to speed up file system recovery, the journal contains a summary of all changes made as part of a single transaction, Most typically a transaction in journal is recorded atomically (full or none), Journals are often stored circularly

Question **13**

Correct

Mark 1.00 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.

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

Your answer is correct.

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 **14**

Complete

Mark 0.25 out of 3.00

List down all changes required to xv6 code, in order to add the system call chown().

Every change should be mentioned in terms of either of the following:

(a) pseudo-code of new function to be added

(b) prototype of any new function or new system call to be added

(c) pseudo-code of changes to an existing function, describing lines to be removed, and lines to be added

(d) **precise** declaration of new data structures to be added in C, or changes to the existing data structure

(e) Name and a one-line description of new userland functionality to be added

(f) Changes to Makefile

(g) Any other change in a maximum of 20 words per change.

(a) int chown (char * path , char *ownername);

(d)there is no need of new data structures or changes to the existing data structure

(e)by using this system call ,user to change the owner if the file

(f)_trychown\ at the end of UPROGS

Comment:

Question **15**

Partially correct

Mark 1.00 out of 2.00

For Virtual File System to work, which of the following changes are required to be done to an existing OS code (e.g. xv6)?

- [ ] a. The operating system in-memory inode needs to be a generic-inode representing "inode" like data structure across multiple file systems.
- [x] b. The generic inode needs to have a field representing if this inode is a mount point and also to refer/point to the root of the mounted file system's inode. ✔
- [x] c. Each file-system writer needs to provide the set of function pointers for VFS, and these function pointers need to be setup in generic inode of "/" of that file system during mount() ✔
- [x] d. A mount() system call should be provided to mount a partition onto some directory in existing namespace rooted at "/" ✔
- [x] e. Each open() needs to copy the function pointers from the inode of the parent directory into the inode of the child (if not already done), unless it's traversing a mount point. (This may be done as part of lookup() which is called by open()) ✔
- [ ] f. The file system specific function pointers, for file system system-calls, need to be setup in the generic inode during lookup.
- [ ] g. The lookup() operation needs to check if it's crossing a mount point and call FS specific operations to read inodes/directories
- [ ] h. The filesystem related system calls (e.g. read, write) need to invoke the file system specific functions (e.g. ext2_read, ext2_write, ntfs_read, ntfs_write) using function pointers.

The correct answers are: A mount() system call should be provided to mount a partition onto some directory in existing namespace rooted at "/", The filesystem related system calls (e.g. read, write) need to invoke the file system specific functions (e.g. ext2_read, ext2_write, ntfs_read, ntfs_write) using function pointers., The file system specific function pointers, for file system system-calls, need to be setup in the generic inode during lookup., The operating system in-memory inode needs to be a generic-inode representing "inode" like data structure across multiple file systems., The generic inode needs to have a field representing if this inode is a mount point and also to refer/point to the root of the mounted file system's inode., The lookup() operation needs to check if it's crossing a mount point and call FS specific operations to read inodes/directories, Each file-system writer needs to provide the set of function pointers for VFS, and these function pointers need to be setup in generic inode of "/" of that file system during mount(), Each open() needs to copy the function pointers from the inode of the parent directory into the inode of the child (if not already done), unless it's traversing a mount point. (This may be done as part of lookup() which is called by open())

Question **16**

Partially correct

Mark 0.75 out of 1.00

Map the technique with it's feature/problem

dynamic linking	small executable file	✔
static loading	wastage of physical memory	✔
static linking	large executable file	✔
dynamic loading	small executable file	✘

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

Question **17**

Correct

Mark 1.00 out of 1.00

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

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

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

Question **18**

Partially correct

Mark 0.75 out of 2.00

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

"..." means some code.

```
struct proc*
myproc(void) {
...
  pushcli();
  c = mycpu();
  p = c->proc;
  popcli();
...
}
```

Traverse ebp chain to get sequence of instructions followed in functions calls ✖

```
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);
  }
```

Release the lock held by some another process ✖

```
void
panic(char *s)
{
...
  panicked = 1;
```

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

```
void
acquire(struct spinlock *lk)
{
  pushcli();
```

Release the lock held by some another process ✖

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

Tell compiler not to reorder memory access beyond this line

```
void
acquire(struct spinlock *lk)
{
...
  getcallerpcs(&lk, lk->pcs);
```

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

```
void
yield(void)
{
...
release(&ptable.lock);
}
```

Release the lock held by some another process

Your answer is partially correct.

You have correctly selected 3.

The correct answer is: struct proc*

```
myproc(void) {
...
  pushcli();
  c = mycpu();
  p = c->proc;
  popcli();
...
}
```

→ Disable interrupts to avoid another process's pointer being returned, 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
panic(char *s)
{
...
  panicked = 1; → Ensure that no printing happens on other processors, void
acquire(struct spinlock *lk)
{
  pushcli();
  → Disable interrupts to avoid deadlocks, void
acquire(struct spinlock *lk)
{
...
__sync_synchronize();

  → Tell compiler not to reorder memory access beyond this line, void
acquire(struct spinlock *lk)
{
...
  getcallerpcs(&lk, lk->pcs);

  → Traverse ebp chain to get sequence of instructions followed in functions calls, void
yield(void)
{
...
release(&ptable.lock);
}

  → Release the lock held by some another process

Question **19**

Correct

Mark 1.00 out of 1.00

Calculate the average waiting time using
FCFS scheduling
for the following workload

assuming that they arrive in this order during the first time unit:

Process	Burst Time
P1	2
P2	6
P3	2
P4	3

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

Answer: 5

P2 waits for 2 units

P3 waits for 2+6 units

P4 waits for 2 + 6 +2 units of time

Total waiting = 2 + 2 + 6 + 2 + 6 + 2 = 20 units

Average waiting time = 20/4 = 5

The correct answer is: 5

Question **20**

Incorrect

Mark 0.00 out of 1.00

Select all correct statements about file system recovery (without journaling) programs e.g. fsck

Select one or more:

- [x] a. Recovery programs recalculate most of the metadata summaries (e.g. free inode count)✔
- [x] b. It is possible to lose data as part of recovery✔
- [x] c. Even with a write-through policy, it is possible to need a recovery program.✔
- [x] d. They can make changes to the on-disk file system✔
- [x] e. A recovery program, most typically, builds the file system data structure and checks for inconsistencies✔
- [x] f. They may take very long time to execute✔
- [x] g. They are used to recover deleted files✖
- [ ] h. Recovery programs are needed only if the file system has a delayed-write policy.
- [ ] i. Recovery is possible due to redundancy in file system data structures

Your answer is incorrect.

The correct answers are: Recovery is possible due to redundancy in file system data structures, A recovery program, most typically, builds the file system data structure and checks for inconsistencies, It is possible to lose data as part of recovery, They may take very long time to execute, They can make changes to the on-disk file system, Recovery programs recalculate most of the metadata summaries (e.g. free inode count), Recovery programs are needed only if the file system has a delayed-write policy., Even with a write-through policy, it is possible to need a recovery program.

Question **21**

Correct

Mark 1.00 out of 1.00

Given that the memory access time is 150 ns, probability of a page fault is 0.8 and page fault handling time is 6 ms,
The effective memory access time in nanoseconds is:

Answer: 4800030 ✔

The correct answer is: 4800030.00

Question **22**

Correct

Mark 1.00 out of 1.00

Assuming a 8- KB page size, what is the page numbers for the address 1005699 reference in decimal :

(give answer also in decimal)

Answer: 123

The correct answer is: 123

Question **23**

Correct

Mark 1.00 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		
○ ✗	◉ ☑	String arguments are first copied to trapframe and then from trapframe to kernel's other variables.	✔
○ ✗	◉ ☑	The arguments to system call are copied to kernel stack in trapasm.S	✔
◉ ☑	○ ✗	The arguments are accessed in the kernel code using esp on the trapframe.	✔
◉ ☑	○ ✗	String arguments are NOT copied in kernel memory, but just pointed to by a kernel memory pointer	✔
◉ ☑	○ ✗	Integer arguments are copied from user memory to kernel memory using argint()	✔
◉ ☑	○ ✗	The arguments to system call originally reside on process stack.	✔
○ ✗	◉ ☑	Integer arguments are stored in eax, ebx, ecx, etc. registers	✔
◉ ☑	○ ✗	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.: False
The arguments to system call are copied to kernel stack in trapasm.S: False
The arguments are accessed in the kernel code using esp on the trapframe.: True
String arguments are NOT copied in kernel memory, but just pointed to by a kernel memory pointer: True
Integer arguments are copied from user memory to kernel memory using argint(): True
The arguments to system call originally reside on process stack.: True
Integer arguments are stored in eax, ebx, ecx, etc. registers: False
The functions like argint(), argstr() make the system call arguments available in the kernel.: True

◄ Random Quiz - 6 (xv6 file system)

Jump to...

Homework questions: Basics of MM, xv6 booting ►

Started on	Wednesday, 19 April 2023, 8:46 PM
State	Finished
Completed on	Wednesday, 19 April 2023, 9:26 PM
Time taken	39 mins 31 secs
Grade	**15.60** out of 30.00 (**51.99**%)

Question **1**

Incorrect

Mark 0.00 out of 2.00

Select all the correct statements about synchronization primitives.

Select one or more:

- [ ] a. Blocking means moving the process to a wait queue and calling scheduler
- [x] b. Blocking means moving the process to a wait queue and spinning ✖
- [x] c. Semaphores are always a good substitute for spinlocks ✖
- [ ] d. Spinlocks are good for multiprocessor scenarios, for small critical sections
- [ ] e. All synchronization primitives are implemented essentially with some hardware assistance.
- [x] f. Blocking means one process passing over control to another process ✖
- [x] g. Semaphores can be used for synchronization scenarioes like ordered execution ✔
- [ ] h. Mutexes can be implemented without any hardware assistance
- [ ] i. Spinlocks consume CPU time
- [ ] j. Mutexes can be implemented using spinlock
- [ ] k. Thread that is going to block should not be holding any spinlock
- [ ] l. Mutexes can be implemented using blocking and wakeup

Your answer is incorrect.

The correct answers are: Spinlocks are good for multiprocessor scenarios, for small critical sections, Spinlocks consume CPU time, Semaphores can be used for synchronization scenarioes like ordered execution, Mutexes can be implemented using spinlock, Mutexes can be implemented using blocking and wakeup, Thread that is going to block should not be holding any spinlock, Blocking means moving the process to a wait queue and calling scheduler, All synchronization primitives are implemented essentially with some hardware assistance.

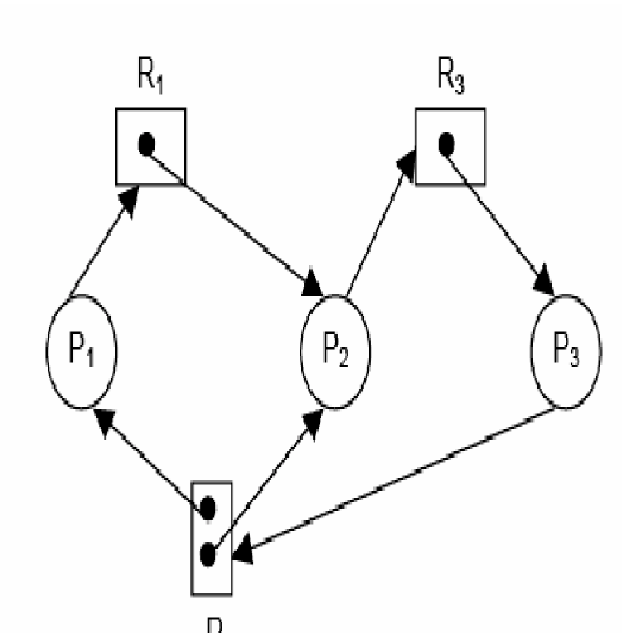
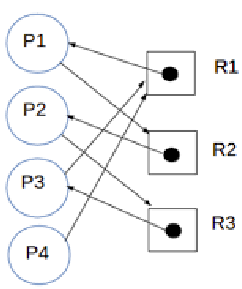
Question **2**

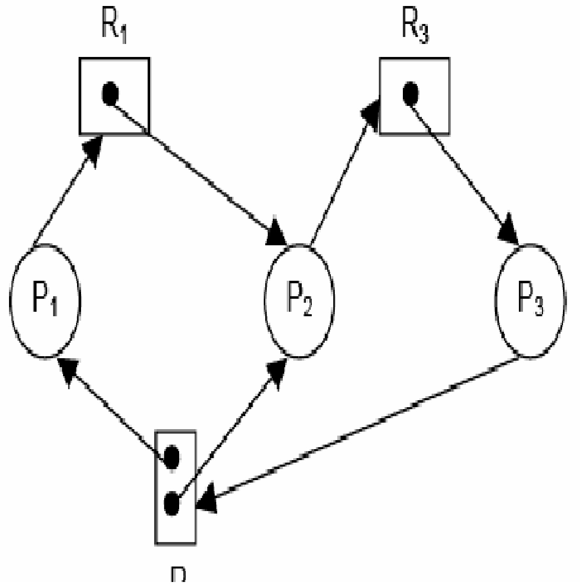
Partially correct

Mark 0.50 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

Started on	Tuesday, 16 January 2024, 5:08 PM
State	Finished
Completed on	Tuesday, 16 January 2024, 5:52 PM
Time taken	44 mins 52 secs
Grade	**10.20** out of 15.00 (**68**%)

Question **1**

Incorrect

Mark 0.00 out of 1.00

Select all the correct statements about the process init on Linuxes/Unixes.

Select one or more:

- [x] a. init can not be killed with SIGKILL ✔
- [ ] b. only a process run by 'root' user can exec 'init'
- [ ] c. init is created by kernel 'by hand'
- [x] d. any user process can fork and exec init ✖
- [x] e. init typically has a pid=1 ✔
- [x] f. init is created by kernel by forking itself ✖
- [ ] g. no process can exec 'init'

Your answer is incorrect.

The correct answers are: init is created by kernel 'by hand', init typically has a pid=1, init can not be killed with SIGKILL, only a process run by 'root' user can exec 'init'

Question **2**

Incorrect

Mark 0.00 out of 1.00

Write the possible contents of the file /tmp/xyz after this program.

In the answer if you want to mention any non-text character, then write \0 For example abc\0\0 means abc followed by any two non-text characters

```
int main(int argc, char *argv[]) {
    int fd1, fd2, n, i;
    char buf[128];

    fd1 = open("/tmp/xyz", O_WRONLY | O_CREAT, S_IRUSR|S_IWUSR);
    write(fd1, "hello", 5);
    fd2 = open("/tmp/xyz", O_WRONLY, S_IRUSR|S_IWUSR);
    write(fd2, "bye", 3);
    close(fd1);
    close(fd2);
    return 0;
}
```

Answer: byelo\0\0 ✖

The correct answer is: byelo

Question **3**

Partially correct

Mark 0.60 out of 1.00

Select all the correct statements about two modes of CPU operation

Select one or more:

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

Your answer is partially correct.

You have correctly selected 3.
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 **4**

Correct

Mark 0.50 out of 0.50

Compare multiprogramming with multitasking

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

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

Question **5**

Partially correct

Mark 0.25 out of 1.00

Given below is the output of "ps -eaf".

Answer the questions based on it.

```
UID          PID    PPID  C STIME TTY          TIME CMD
root           1       0  0 Jan05 ?        00:01:08 /sbin/init splash
root           2       0  0 Jan05 ?        00:00:00 [kthreadd]
root           3       2  0 Jan05 ?        00:00:00 [rcu_gp]
root           4       2  0 Jan05 ?        00:00:00 [rcu_par_gp]
root           9       2  0 Jan05 ?        00:00:00 [mm_percpu_wq]
root          10       2  0 Jan05 ?        00:00:00 [rcu_tasks_rude_]
root          11       2  0 Jan05 ?        00:00:00 [rcu_tasks_trace]
root          12       2  0 Jan05 ?        00:00:22 [ksoftirqd/0]
root          13       2  0 Jan05 ?        00:06:29 [rcu_sched]
root          14       2  0 Jan05 ?        00:00:02 [migration/0]
root          15       2  0 Jan05 ?        00:00:00 [idle_inject/0]
root          16       2  0 Jan05 ?        00:00:00 [cpuhp/0]
root          17       2  0 Jan05 ?        00:00:00 [cpuhp/1]
root          18       2  0 Jan05 ?        00:00:00 [idle_inject/1]
root          19       2  0 Jan05 ?        00:00:03 [migration/1]
root          20       2  0 Jan05 ?        00:00:13 [ksoftirqd/1]
root          22       2  0 Jan05 ?        00:00:00 [kworker/1:0H-events_highpri]
root          23       2  0 Jan05 ?        00:00:00 [cpuhp/2]
root          24       2  0 Jan05 ?        00:00:00 [idle_inject/2]
root          25       2  0 Jan05 ?        00:00:01 [migration/2]
root          26       2  0 Jan05 ?        00:00:09 [ksoftirqd/2]
root          28       2  0 Jan05 ?        00:00:00 [kworker/2:0H-kblockd]
root          29       2  0 Jan05 ?        00:00:00 [cpuhp/3]
root          30       2  0 Jan05 ?        00:00:00 [idle_inject/3]
root          31       2  0 Jan05 ?        00:00:02 [migration/3]
root          32       2  0 Jan05 ?        00:00:07 [ksoftirqd/3]
root          34       2  0 Jan05 ?        00:00:00 [kworker/3:0H-events_highpri]
root          35       2  0 Jan05 ?        00:00:00 [cpuhp/4]
root          36       2  0 Jan05 ?        00:00:00 [idle_inject/4]
root          37       2  0 Jan05 ?        00:00:02 [migration/4]
root          38       2  0 Jan05 ?        00:00:06 [ksoftirqd/4]
root          40       2  0 Jan05 ?        00:00:00 [kworker/4:0H-events_highpri]
root          41       2  0 Jan05 ?        00:00:00 [cpuhp/5]
root          42       2  0 Jan05 ?        00:00:00 [idle_inject/5]
root          43       2  0 Jan05 ?        00:00:02 [migration/5]
root          44       2  0 Jan05 ?        00:00:05 [ksoftirqd/5]
root          46       2  0 Jan05 ?        00:00:00 [kworker/5:0H-events_highpri]
root          47       2  0 Jan05 ?        00:00:00 [cpuhp/6]
root          48       2  0 Jan05 ?        00:00:00 [idle_inject/6]
root          49       2  0 Jan05 ?        00:00:02 [migration/6]
root          50       2  0 Jan05 ?        00:00:05 [ksoftirqd/6]
root          52       2  0 Jan05 ?        00:00:00 [kworker/6:0H-events_highpri]
root          53       2  0 Jan05 ?        00:00:00 [cpuhp/7]
root          54       2  0 Jan05 ?        00:00:00 [idle_inject/7]
root          55       2  0 Jan05 ?        00:00:02 [migration/7]
root          56       2  0 Jan05 ?        00:00:06 [ksoftirqd/7]
root          58       2  0 Jan05 ?        00:00:00 [kworker/7:0H-events_highpri]
root          59       2  0 Jan05 ?        00:00:00 [kdevtmpfs]
root          60       2  0 Jan05 ?        00:00:00 [netns]
root          61       2  0 Jan05 ?        00:00:00 [inet_frag_wq]
root          62       2  0 Jan05 ?        00:00:00 [kauditd]
root          63       2  0 Jan05 ?        00:00:00 [khungtaskd]
root          64       2  0 Jan05 ?        00:00:00 [oom_reaper]
root          65       2  0 Jan05 ?        00:00:00 [writeback]
root          66       2  0 Jan05 ?        00:01:58 [kcompactd0]
root          67       2  0 Jan05 ?        00:00:00 [ksmd]
root          68       2  0 Jan05 ?        00:00:04 [khugepaged]
root         115       2  0 Jan05 ?        00:00:00 [kintegrityd]
root         116       2  0 Jan05 ?        00:00:00 [kblockd]
root         117       2  0 Jan05 ?        00:00:00 [blkcg_punt_bio]
root         118       2  0 Jan05 ?        00:00:00 [tpm_dev_wq]
```

```
root         119       2  0 Jan05 ?        00:00:00 [ata_sff]
root         120       2  0 Jan05 ?        00:00:00 [md]
root         121       2  0 Jan05 ?        00:00:00 [edac-poller]
root         122       2  0 Jan05 ?        00:00:00 [devfreq_wq]
root         123       2  0 Jan05 ?        00:00:00 [watchdogd]
root         129       2  0 Jan05 ?        00:00:00 [irq/25-AMD-Vi]
root         131       2  0 Jan05 ?        00:04:33 [kswapd0]
root         132       2  0 Jan05 ?        00:00:00 [ecryptfs-kthrea]
root         134       2  0 Jan05 ?        00:00:00 [kthrotld]
root         135       2  0 Jan05 ?        00:00:00 [irq/27-pciehp]
root         139       2  0 Jan05 ?        00:00:00 [acpi_thermal_pm]
root         140       2  0 Jan05 ?        00:00:00 [vfio-irqfd-clea]
root         141       2  0 Jan05 ?        00:00:00 [ipv6_addrconf]
root         144       2  0 Jan05 ?        00:00:03 [kworker/6:1H-kblockd]
root         151       2  0 Jan05 ?        00:00:00 [kstrp]
root         154       2  0 Jan05 ?        00:00:00 [zswap-shrink]
root         162       2  0 Jan05 ?        00:00:00 [charger_manager]
root         164       2  0 Jan05 ?        00:00:03 [kworker/4:1H-kblockd]
root         197       2  0 Jan05 ?        00:00:03 [kworker/3:1H-kblockd]
root         213       2  0 Jan05 ?        00:00:03 [kworker/7:1H-kblockd]
root         215       2  0 Jan05 ?        00:00:00 [nvme-wq]
root         216       2  0 Jan05 ?        00:00:00 [nvme-reset-wq]
root         217       2  0 Jan05 ?        00:00:00 [nvme-delete-wq]
root         223       2  0 Jan05 ?        00:00:00 [irq/42-ELAN2513]
root         224       2  0 Jan05 ?        00:04:20 [irq/41-ELAN071B]
root         225       2  0 Jan05 ?        00:00:00 [cryptd]
root         226       2  0 Jan05 ?        00:00:00 [amd_iommu_v2]
root         249       2  0 Jan05 ?        00:00:00 [ttm_swap]
root         250       2  0 Jan05 ?        00:20:29 [gfx]
root         251       2  0 Jan05 ?        00:00:00 [comp_1.0.0]
root         252       2  0 Jan05 ?        00:00:00 [comp_1.1.0]
root         253       2  0 Jan05 ?        00:00:00 [comp_1.2.0]
root         254       2  0 Jan05 ?        00:00:00 [comp_1.3.0]
root         255       2  0 Jan05 ?        00:00:00 [comp_1.0.1]
root         256       2  0 Jan05 ?        00:00:00 [comp_1.1.1]
root         257       2  0 Jan05 ?        00:00:00 [comp_1.2.1]
root         258       2  0 Jan05 ?        00:00:00 [comp_1.3.1]
root         259       2  0 Jan05 ?        00:00:27 [sdma0]
root         260       2  0 Jan05 ?        00:00:00 [vcn_dec]
root         261       2  0 Jan05 ?        00:00:00 [vcn_enc0]
root         262       2  0 Jan05 ?        00:00:00 [vcn_enc1]
root         263       2  0 Jan05 ?        00:00:00 [jpeg_dec]
root         265       2  0 Jan05 ?        00:00:00 [card0-crtc0]
root         266       2  0 Jan05 ?        00:00:00 [card0-crtc1]
root         267       2  0 Jan05 ?        00:00:00 [card0-crtc2]
root         268       2  0 Jan05 ?        00:00:00 [card0-crtc3]
root         271       2  0 Jan05 ?        00:00:03 [kworker/5:1H-kblockd]
root         277       2  0 Jan05 ?        00:00:03 [kworker/1:1H-kblockd]
root         320       2  0 Jan05 ?        00:00:00 [raid5wq]
root         380       2  0 Jan05 ?        00:00:11 [jbd2/nvme0n1p5-]
root         381       2  0 Jan05 ?        00:00:00 [ext4-rsv-conver]
root         432       2  0 Jan05 ?        00:00:03 [kworker/2:1H-kblockd]
root         446       1  0 Jan05 ?        00:01:16 /lib/systemd/systemd-journald
root         470       2  0 Jan05 ?        00:00:00 [rpciod]
root         473       2  0 Jan05 ?        00:00:00 [xprtiod]
root         474       2  0 Jan05 ?        00:00:00 bpfilter_umh
root         503       1  0 Jan05 ?        00:00:07 /lib/systemd/systemd-udevd
root         517       2  0 Jan05 ?        00:00:00 [loop0]
root         554       2  0 Jan05 ?        00:00:00 [loop1]
root         563       2  0 Jan05 ?        00:00:00 [loop2]
root         586       2  0 Jan05 ?        00:00:00 [loop3]
root         588       2  0 Jan05 ?        00:00:00 [loop4]
root         589       2  0 Jan05 ?        00:00:00 [loop5]
root         612       2  0 Jan05 ?        00:00:00 [loop6]
root         613       2  0 Jan05 ?        00:00:00 [loop7]
root         629       2  0 Jan05 ?        00:00:00 [loop8]
root         637       2  0 Jan05 ?        00:00:00 [cfg80211]
root         676       2  0 Jan05 ?        00:05:00 [irq/75-iwlwifi:]
root         678       2  0 Jan05 ?        00:01:07 [irq/76-iwlwifi:]
root         682       2  0 Jan05 ?        00:01:27 [irq/77-iwlwifi:]
```

```
root         688       2  0 Jan05 ?        00:00:49 [irq/78-iwlwifi:]
root         695       2  0 Jan05 ?        00:01:39 [irq/79-iwlwifi:]
root         700       2  0 Jan05 ?        00:01:22 [irq/80-iwlwifi:]
root         703       2  0 Jan05 ?        00:01:13 [irq/81-iwlwifi:]
root         704       2  0 Jan05 ?        00:01:38 [irq/82-iwlwifi:]
root         708       2  0 Jan05 ?        00:00:44 [irq/83-iwlwifi:]
root         713       2  0 Jan05 ?        00:00:00 [loop9]
root         715       2  0 Jan05 ?        00:00:00 [irq/84-iwlwifi:]
root         782       2  0 Jan05 ?        00:00:00 [loop10]
root         797       2  0 Jan05 ?        00:00:00 [loop11]
root         811       2  0 Jan05 ?        00:00:00 [loop12]
root         838       2  0 Jan05 ?        00:00:00 [loop13]
root         847       2  0 Jan05 ?        00:00:00 [loop14]
root         879       2  0 Jan05 ?        00:00:00 [loop15]
root         884       2  0 Jan05 ?        00:00:00 [loop16]
root         885       2  0 Jan05 ?        00:00:00 [loop17]
root         945       2  0 Jan05 ?        00:00:00 [loop18]
root         946       2  0 Jan05 ?        00:00:00 [loop19]
root         947       2  0 Jan05 ?        00:00:00 [loop20]
root        1012       2  0 Jan05 ?        00:00:00 [jbd2/nvme0n1p8-]
root        1013       2  0 Jan05 ?        00:00:00 [ext4-rsv-conver]
root        1015       2  0 Jan05 ?        00:01:09 [jbd2/nvme0n1p7-]
root        1016       2  0 Jan05 ?        00:00:00 [ext4-rsv-conver]
_rpc        1062       1  0 Jan05 ?        00:00:00 /sbin/rpcbind -f -w
systemd+    1063       1  0 Jan05 ?        00:01:24 /lib/systemd/systemd-resolved
systemd+    1064       1  0 Jan05 ?        00:00:00 /lib/systemd/systemd-timesyncd
root        1144       1  0 Jan05 ?        00:00:46 /usr/sbin/acpid
avahi       1146       1  0 Jan05 ?        00:00:06 avahi-daemon: running [abhijit-laptop.local]
root        1149       1  0 Jan05 ?        00:00:01 /usr/lib/bluetooth/bluetoothd
message+    1150       1  0 Jan05 ?        00:04:21 /usr/bin/dbus-daemon --system --address=systemd: --nofork --
nopidfile --systemd-activation --syslog-only
root        1152       1  0 Jan05 ?        00:03:12 /usr/sbin/NetworkManager --no-daemon
root        1157       1  0 Jan05 ?        00:01:02 /usr/sbin/iio-sensor-proxy
root        1159       1  0 Jan05 ?        00:00:27 /usr/sbin/irqbalance --foreground
root        1162       1  0 Jan05 ?        00:00:01 /usr/bin/lxcfs /var/lib/lxcfs
root        1165       1  0 Jan05 ?        00:00:00 /usr/bin/python3 /usr/bin/networkd-dispatcher --run-startup-
triggers
root        1170       1  0 Jan05 ?        00:00:29 /usr/lib/policykit-1/polkitd --no-debug
syslog      1175       1  0 Jan05 ?        00:00:20 /usr/sbin/rsyslogd -n -iNONE
root        1182       1  0 Jan05 ?        00:01:13 /usr/lib/snapd/snapd
root        1187       1  0 Jan05 ?        00:00:12 /usr/lib/accountsservice/accounts-daemon
root        1192       1  0 Jan05 ?        00:00:00 /usr/sbin/cron -f
root        1198       1  0 Jan05 ?        00:00:00 /usr/libexec/switcheroo-control
root        1201       1  0 Jan05 ?        00:00:12 /lib/systemd/systemd-logind
root        1202       1  0 Jan05 ?        00:00:07 /lib/systemd/systemd-machined
root        1203       1  0 Jan05 ?        00:01:28 /usr/lib/udisks2/udisksd
root        1204       1  0 Jan05 ?        00:00:15 /sbin/wpa_supplicant -u -s -O /run/wpa_supplicant
daemon      1209       1  0 Jan05 ?        00:00:00 /usr/sbin/atd -f
avahi       1216    1146  0 Jan05 ?        00:00:00 avahi-daemon: chroot helper
docker-+    1279       1  0 Jan05 ?        00:00:22 /usr/bin/docker-registry serve /etc/docker/registry/config.yml
root        1282       1  0 Jan05 ?        00:00:00 /usr/bin/python3 /usr/bin/twistd3 --nodaemon --pidfile= epoptes
jenkins     1285       1  0 Jan05 ?        00:15:01 /usr/bin/java -Djava.awt.headless=true -jar
/usr/share/java/jenkins.war --webroot=/var/cache/jenkins/war --httpPort=8080
root        1296       1  0 Jan05 ?        00:00:18 php-fpm: master process (/etc/php/7.4/fpm/php-fpm.conf)
vnstat      1314       1  0 Jan05 ?        00:00:15 /usr/sbin/vnstatd -n
root        1326       1  0 Jan05 ?        00:00:02 /usr/sbin/ModemManager
root        1327       1  0 Jan05 ?        00:00:31 /usr/bin/anydesk --service
colord      1359       1  0 Jan05 ?        00:00:01 /usr/libexec/colord
root        1374       1  0 Jan05 ?        00:00:00 /usr/sbin/gdm3
root        1420       1  0 Jan05 ?        00:00:14 /usr/sbin/apache2 -k start
www-data    1436    1296  0 Jan05 ?        00:00:00 php-fpm: pool www
www-data    1437    1296  0 Jan05 ?        00:00:00 php-fpm: pool www
mysql       1461       1  0 Jan05 ?        00:38:52 /usr/sbin/mysqld
root        1490       1  0 Jan05 ?        00:00:04 /usr/sbin/libvirtd
root        1491       1  0 Jan05 ?        00:00:00 /usr/bin/python3 /usr/share/unattended-upgrades/unattended-upgrade-
shutdown --wait-for-signal
rtkit       1593       1  0 Jan05 ?        00:00:07 /usr/libexec/rtkit-daemon
libvirt+    1766       1  0 Jan05 ?        00:00:00 /usr/sbin/dnsmasq --conf-file=/var/lib/libvirt/dnsmasq/default.conf
--leasefile-ro --dhcp-script=/usr/lib/libvirt/libvirt_leaseshelper
root        1767    1766  0 Jan05 ?        00:00:00 /usr/sbin/dnsmasq --conf-file=/var/lib/libvirt/dnsmasq/default.conf
```

```
--leasefile-ro --dhcp-script=/usr/lib/libvirt/libvirt_leaseshelper
root        1859       1  0 Jan05 ?        00:00:18 /usr/lib/upower/upowerd
root        1995       1  0 Jan05 ?        00:00:00 /opt/saltstack/salt/run/run minion
root        2041    1995  0 Jan05 ?        00:05:18 /opt/saltstack/salt/run/run minion MultiMinionProcessManager
MinionProcessManager
root        2278       1  0 Jan05 ?        00:00:50 /usr/bin/dockerd -H fd:// --
containerd=/run/containerd/containerd.sock
root        2282       1  0 Jan05 ?        00:00:17 /usr/sbin/inetd
whoopsie    2302       1  0 Jan05 ?        00:00:01 /usr/bin/whoopsie -f
kernoops    2330       1  0 Jan05 ?        00:00:19 /usr/sbin/kerneloops --test
kernoops    2341       1  0 Jan05 ?        00:00:19 /usr/sbin/kerneloops
root        2366       2  0 Jan05 ?        00:00:00 [iprt-VBoxWQueue]
lxc-dns+    2370       1  0 Jan05 ?        00:00:00 dnsmasq --conf-file=/dev/null -u lxc-dnsmasq --strict-order --bind-
interfaces --pid-file=/run/lxc/dnsmasq.pid --listen-address 10.0.3.1 --dhcp-range 10.0.3.2,10.0.3.254 --dhcp-lease-
max=253 --dhcp-no-override --except-interface=lo --interface=lxcbr0 --dhcp-
leasefile=/var/lib/misc/dnsmasq.lxcbr0.leases --dhcp-authoritative
root        2404       2  0 Jan05 ?        00:00:00 [iprt-VBoxTscThr]
root        3508       1  0 Jan05 ?        00:00:01 /usr/lib/postfix/sbin/master -w
root        3615    1374  0 Jan05 ?        00:00:01 gdm-session-worker [pam/gdm-password]
abhijit     3629       1  0 Jan05 ?        00:00:17 /lib/systemd/systemd --user
abhijit     3630    3629  0 Jan05 ?        00:00:00 (sd-pam)
abhijit     3636    3629  1 Jan05 ?        04:31:21 /usr/bin/pulseaudio --daemonize=no --log-target=journal
abhijit     3638    3629  0 Jan05 ?        00:20:35 /usr/libexec/tracker-miner-fs
abhijit     3642    3629  0 Jan05 ?        00:01:11 /usr/bin/dbus-daemon --session --address=systemd: --nofork --
nopidfile --systemd-activation --syslog-only
abhijit     3644       1  0 Jan05 ?        00:00:03 /usr/bin/gnome-keyring-daemon --daemonize --login
abhijit     3662    3629  0 Jan05 ?        00:00:01 /usr/libexec/gvfsd
abhijit     3667    3629  0 Jan05 ?        00:00:00 /usr/libexec/gvfsd-fuse /run/user/1000/gvfs -f -o big_writes
abhijit     3673    3629  0 Jan05 ?        00:00:02 /usr/libexec/gvfs-udisks2-volume-monitor
abhijit     3681    3629  0 Jan05 ?        00:00:01 /usr/libexec/gvfs-mtp-volume-monitor
abhijit     3685    3629  0 Jan05 ?        00:00:11 /usr/libexec/gvfs-afc-volume-monitor
abhijit     3690    3629  0 Jan05 ?        00:00:01 /usr/libexec/gvfs-gphoto2-volume-monitor
abhijit     3695    3629  0 Jan05 ?        00:00:01 /usr/libexec/gvfs-goa-volume-monitor
abhijit     3700    3629  0 Jan05 ?        00:00:01 /usr/libexec/goa-daemon
root        3701       2  0 Jan05 ?        00:00:00 [krfcommd]
abhijit     3708    3629  0 Jan05 ?        00:00:04 /usr/libexec/goa-identity-service
abhijit     3724    3615  0 Jan05 tty2     00:00:00 /usr/lib/gdm3/gdm-x-session --run-script env
GNOME_SHELL_SESSION_MODE=ubuntu /usr/bin/gnome-session --systemd --session=ubuntu
abhijit     3726    3724  1 Jan05 tty2     02:47:57 /usr/lib/xorg/Xorg vt2 -displayfd 3 -auth
/run/user/1000/gdm/Xauthority -background none -noreset -keeptty -verbose 3
abhijit     3747    3724  0 Jan05 tty2     00:00:00 /usr/libexec/gnome-session-binary --systemd --systemd --
session=ubuntu
abhijit     3816    3747  0 Jan05 ?        00:00:01 /usr/bin/ssh-agent /usr/bin/im-launch env
GNOME_SHELL_SESSION_MODE=ubuntu /usr/bin/gnome-session --systemd --session=ubuntu
abhijit     3845    3629  0 Jan05 ?        00:00:00 /usr/libexec/at-spi-bus-launcher
abhijit     3850    3845  0 Jan05 ?        00:00:07 /usr/bin/dbus-daemon --config-file=/usr/share/defaults/at-
spi2/accessibility.conf --nofork --print-address 3
abhijit     3869    3629  0 Jan05 ?        00:00:00 /usr/libexec/gnome-session-ctl --monitor
abhijit     3876    3629  0 Jan05 ?        00:00:04 /usr/libexec/gnome-session-binary --systemd-service --
session=ubuntu
abhijit     3890    3629  1 Jan05 ?        03:43:47 /usr/bin/gnome-shell
abhijit     3927    3890  0 Jan05 ?        00:31:49 ibus-daemon --panel disable --xim
abhijit     3931    3927  0 Jan05 ?        00:00:00 /usr/libexec/ibus-dconf
abhijit     3932    3927  0 Jan05 ?        00:01:51 /usr/libexec/ibus-extension-gtk3
abhijit     3934    3629  0 Jan05 ?        00:00:04 /usr/libexec/ibus-x11 --kill-daemon
abhijit     3937    3629  0 Jan05 ?        00:00:02 /usr/libexec/ibus-portal
abhijit     3949    3629  0 Jan05 ?        00:00:27 /usr/libexec/at-spi2-registryd --use-gnome-session
abhijit     3953    3629  0 Jan05 ?        00:00:00 /usr/libexec/xdg-permission-store
abhijit     3955    3629  0 Jan05 ?        00:00:01 /usr/libexec/gnome-shell-calendar-server
abhijit     3964    3629  0 Jan05 ?        00:00:00 /usr/libexec/evolution-source-registry
abhijit     3973    3629  0 Jan05 ?        00:00:02 /usr/libexec/evolution-calendar-factory
abhijit     3986    3629  0 Jan05 ?        00:00:01 /usr/libexec/dconf-service
abhijit     3992    3629  0 Jan05 ?        00:00:01 /usr/libexec/evolution-addressbook-factory
abhijit     4007    3629  0 Jan05 ?        00:00:00 /usr/bin/gjs /usr/share/gnome-shell/org.gnome.Shell.Notifications
abhijit     4023    3629  0 Jan05 ?        00:00:00 /usr/libexec/gsd-a11y-settings
abhijit     4025    3629  0 Jan05 ?        00:00:08 /usr/libexec/gsd-color
abhijit     4029    3629  0 Jan05 ?        00:00:00 /usr/libexec/gsd-datetime
abhijit     4032    3629  0 Jan05 ?        00:00:21 /usr/libexec/gsd-housekeeping
abhijit     4033    3629  0 Jan05 ?        00:00:05 /usr/libexec/gsd-keyboard
abhijit     4036    3629  0 Jan05 ?        00:00:12 /usr/libexec/gsd-media-keys
```

```
abhijit     4037    3629  0 Jan05 ?        00:00:11 /usr/libexec/gsd-power
abhijit     4038    3629  0 Jan05 ?        00:00:00 /usr/libexec/gsd-print-notifications
abhijit     4039    3629  0 Jan05 ?        00:00:01 /usr/libexec/gsd-rfkill
abhijit     4041    3629  0 Jan05 ?        00:00:01 /usr/libexec/gsd-screensaver-proxy
abhijit     4042    3876  0 Jan05 ?        00:01:06 /usr/lib/x86_64-linux-gnu/libexec/kdeconnectd
abhijit     4045    3629  0 Jan05 ?        00:00:35 /usr/libexec/gsd-sharing
abhijit     4047    3629  0 Jan05 ?        00:00:00 /usr/libexec/gsd-smartcard
abhijit     4051    3629  0 Jan05 ?        00:00:00 /usr/libexec/gsd-sound
abhijit     4057    3629  0 Jan05 ?        00:00:00 /usr/libexec/gsd-usb-protection
abhijit     4063    3629  0 Jan05 ?        00:00:05 /usr/libexec/gsd-wacom
abhijit     4071    3629  0 Jan05 ?        00:00:00 /usr/libexec/gsd-wwan
abhijit     4072    3876  0 Jan05 ?        00:01:00 baloo_file
abhijit     4075    3876  0 Jan05 ?        00:00:00 /usr/libexec/gsd-disk-utility-notify
abhijit     4076    3629  0 Jan05 ?        00:00:08 /usr/libexec/gsd-xsettings
abhijit     4078    3876  0 Jan05 ?        00:00:10 /usr/bin/python3 /usr/bin/blueman-applet
abhijit     4082    3876  0 Jan05 ?        00:00:14 /usr/bin/anydesk --tray
abhijit     4108    3876  0 Jan05 ?        00:00:00 /usr/lib/x86_64-linux-gnu/indicator-messages/indicator-messages-
service
abhijit     4109    3876  0 Jan05 ?        00:00:06 /usr/libexec/evolution-data-server/evolution-alarm-notify
abhijit     4129    3629  0 Jan05 ?        00:00:50 /snap/snap-store/959/usr/bin/snap-store --gapplication-service
abhijit     4191    3629  0 Jan05 ?        00:00:00 /usr/libexec/gsd-printer
abhijit     4219    3629  0 Jan05 ?        00:00:03 /usr/libexec/xdg-document-portal
abhijit     4265    3927  0 Jan05 ?        00:03:20 /usr/libexec/ibus-engine-simple
abhijit     4291    3629  0 Jan05 ?        00:00:10 /usr/bin/python3 /usr/bin/blueman-tray
abhijit     4301    3629  0 Jan05 ?        00:00:00 /usr/lib/bluetooth/obexd
abhijit     4377    3662  0 Jan05 ?        00:00:02 /usr/libexec/gvfsd-trash --spawner :1.3 /org/gtk/gvfs/exec_spaw/0
abhijit     4395    3629  0 Jan05 ?        00:00:12 /usr/libexec/xdg-desktop-portal
abhijit     4399    3629  0 Jan05 ?        00:01:08 /usr/libexec/xdg-desktop-portal-gtk
abhijit     4480    3629  0 Jan05 ?        00:00:21 /usr/libexec/gvfsd-metadata
abhijit     5687    3662  0 Jan05 ?        00:00:00 /usr/libexec/gvfsd-network --spawner :1.3 /org/gtk/gvfs/exec_spaw/1
abhijit     5701    3662  0 Jan05 ?        00:00:01 /usr/libexec/gvfsd-dnssd --spawner :1.3 /org/gtk/gvfs/exec_spaw/3
root        6228       2  0 Jan05 ?        00:00:00 [kdmflush]
root        6236       2  0 Jan05 ?        00:00:00 [kcryptd_io/253:]
root        6237       2  0 Jan05 ?        00:00:00 [kcryptd/253:0]
root        6238       2  0 Jan05 ?        00:00:23 [dmcrypt_write/2]
root        6260       2  0 Jan05 ?        00:00:24 [jbd2/dm-0-8]
root        6261       2  0 Jan05 ?        00:00:00 [ext4-rsv-conver]
abhijit     6421    3927  0 Jan05 ?        00:00:36 /usr/lib/ibus/ibus-engine-m17n --ibus
abhijit     6434    3876  0 Jan05 ?        00:00:13 update-notifier
abhijit    30565    3629  0 Jan05 ?        00:08:56 /usr/libexec/gnome-terminal-server
abhijit    30576   30565  0 Jan05 pts/0    00:00:00 bash
abhijit   131017  364845  1 Jan09 ?        03:12:27 /usr/lib/virtualbox/VirtualBoxVM --comment ubuntu 18.04 --startvm
45993a5c-3ded-452f-941e-4579d12c1ad9 --no-startvm-errormsgbox
abhijit   159668   30565  0 Jan09 pts/10   00:00:00 bash
abhijit   161637   30565  0 Jan05 pts/1    00:00:01 bash
abhijit   171109  159668  0 Jan09 pts/10   00:00:33 evince......
abhijit   171114    3629  0 Jan09 ?        00:00:00 /usr/libexec/evinced
abhijit   204055    3629  0 Jan10 ?        00:00:13 /usr/bin/python3 /usr/bin/update-manager --no-update --no-focus-on-
map
abhijit   270190    3629  0 Jan05 ?        00:56:34 /usr/lib/x86_64-linux-gnu/libexec/kactivitymanagerd
abhijit   270199    3629  0 Jan05 ?        00:00:24 /usr/bin/kglobalaccel5
abhijit   270207    3629  0 Jan05 ?        00:00:00 kdeinit5: Running...
abhijit   270208  270207  0 Jan05 ?        00:00:19 /usr/lib/x86_64-linux-gnu/libexec/kf5/klauncher --fd=8
abhijit   279971    3629  0 Jan05 ?        01:20:12 telegram-desktop
abhijit   280011  279971  0 Jan05 ?        00:00:00 sh -c /usr/lib/x86_64-linux-gnu/libproxy/0.4.15/pxgsettings
org.gnome.system.proxy org.gnome.system.proxy.http org.gnome.system.proxy.https org.gnome.system.proxy.ftp
org.gnome.system.proxy.socks
abhijit   280015  280011  0 Jan05 ?        00:00:00 /usr/lib/x86_64-linux-gnu/libproxy/0.4.15/pxgsettings
org.gnome.system.proxy org.gnome.system.proxy.http org.gnome.system.proxy.https org.gnome.system.proxy.ftp
org.gnome.system.proxy.socks
abhijit   325756    3629  0 Jan12 ?        00:01:09 /usr/libexec/tracker-store
root      326592       1  0 Jan12 ?        00:00:00 sshd: /usr/sbin/sshd -D [listener] 0 of 10-100 startups
abhijit   347912    3644  0 Jan06 ?        00:00:00 /usr/bin/ssh-agent -D -a /run/user/1000/keyring/.ssh
root      348716       1  0 Jan12 ?        00:00:28 /usr/bin/containerd
abhijit   351306    3629  3 Jan12 ?        03:32:42 /usr/lib/firefox/firefox
abhijit   351429  351306  0 Jan12 ?        00:00:00 /usr/lib/firefox/firefox -contentproc -parentBuildID 20240108143603
-prefsLen 37272 -prefMapSize 247458 -appDir /usr/lib/firefox/browser {83364ade-74ec-4bcc-94f8-9f3779a869f1} 351306 true
socket
abhijit   351458  351306  0 Jan12 ?        00:29:34 /usr/lib/firefox/firefox -contentproc -childID 1 -isForBrowser -
prefsLen 37337 -prefMapSize 247458 -jsInitLen 229864 -parentBuildID 20240108143603 -greomni /usr/lib/firefox/omni.ja -
```

```
appomni /usr/lib/firefox/browser/omni.ja -appDir /usr/lib/firefox/browser {4dbb54ee-e1a2-41e4-b10b-587526532b78} 351306
true tab
abhijit   351495  351306  0 Jan12 ?        00:04:06 /usr/lib/firefox/firefox -contentproc -childID 2 -isForBrowser -
prefsLen 38090 -prefMapSize 247458 -jsInitLen 229864 -parentBuildID 20240108143603 -greomni /usr/lib/firefox/omni.ja -
appomni /usr/lib/firefox/browser/omni.ja -appDir /usr/lib/firefox/browser {02f1a4e7-d831-4a5f-a9fd-59a809bb03e8} 351306
true tab
abhijit   351717  351306  0 Jan12 ?        00:01:14 /usr/lib/firefox/firefox -contentproc -parentBuildID 20240108143603
-sandboxingKind 0 -prefsLen 42823 -prefMapSize 247458 -appDir /usr/lib/firefox/browser {a1e46009-cfd1-46c8-ac9c-
19ba8bf3f46a} 351306 true utility
abhijit   351844  351306  0 Jan12 ?        00:07:23 /usr/lib/firefox/firefox -contentproc -parentBuildID 20240108143603
-prefsLen 43306 -prefMapSize 247458 -appDir /usr/lib/firefox/browser {91008bef-b6d6-452f-b4a6-e78d887b3569} 351306 true
rdd
abhijit   353500    3662  0 Jan06 ?        00:00:00 /usr/libexec/gvfsd-http --spawner :1.3 /org/gtk/gvfs/exec_spaw/4
abhijit   364809    3890  0 Jan06 ?        00:19:20 /usr/lib/virtualbox/VirtualBox
abhijit   364837    3629  0 Jan06 ?        00:16:49 /usr/lib/virtualbox/VBoxXPCOMIPCD
abhijit   364845    3629  0 Jan06 ?        00:29:44 /usr/lib/virtualbox/VBoxSVC --auto-shutdown
root      364957       2  0 Jan06 ?        00:00:00 [dio/dm-0]
abhijit   369749   30565  0 Jan06 pts/3    00:00:00 bash
abhijit   369879   30565  0 Jan06 pts/4    00:00:00 bash
abhijit   379620   30565  0 Jan06 pts/6    00:00:00 bash
abhijit   381917    3890  0 Jan06 ?        00:00:46 flameshot
root      386201       2  0 Jan06 ?        00:00:02 [kworker/0:2H-acpi_thermal_pm]
postfix   389291    3508  0 Jan06 ?        00:00:00 qmgr -l -t unix -u
root      486171       2  0 Jan12 ?        00:00:01 [kworker/0:0H-kblockd]
abhijit   527395    3629  0 Jan12 ?        00:00:49 /usr/bin/gedit --gapplication-service
abhijit   551299  351306  0 Jan12 ?        00:01:54 /usr/lib/firefox/firefox -contentproc -childID 438 -isForBrowser -
prefsLen 35109 -prefMapSize 247458 -jsInitLen 229864 -parentBuildID 20240108143603 -greomni /usr/lib/firefox/omni.ja -
appomni /usr/lib/firefox/browser/omni.ja -appDir /usr/lib/firefox/browser {c4ebd70d-bad7-4e7c-9066-39827103c84e} 351306
true tab
abhijit   552002    3890  0 Jan12 ?        00:03:32 /opt/Signal/signal-desktop --no-sandbox
abhijit   552005  552002  0 Jan12 ?        00:00:00 /opt/Signal/signal-desktop --type=zygote --no-zygote-sandbox --no-
sandbox
abhijit   552006  552002  0 Jan12 ?        00:00:00 /opt/Signal/signal-desktop --type=zygote --no-sandbox
abhijit   552037  552005  0 Jan12 ?        00:03:28 /opt/Signal/signal-desktop --type=gpu-process --no-sandbox --
enable-crash-reporter=18887fa1-4d37-46f0-bf9f-b37513c81cf9,no_channel --user-data-dir=/home/abhijit/.config/Signal --
gpu-
preferences=WAAAAAAAAAAgAAAEAAAAAAAAAAAAAAAAAAAABgAAAAAAA4AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAABAAAAGAAAAAAAAAAAAYAAAA
--use-gl=angle --use-angle=swiftshader-webgl --shared-files --field-trial-
handle=0,i,3213315393136307567,15155041764863120512,262144 --disable-
features=HardwareMediaKeyHandling,SpareRendererForSitePerProcess
abhijit   552045  552002  0 Jan12 ?        00:00:04 /opt/Signal/signal-desktop --type=utility --utility-sub-
type=network.mojom.NetworkService --lang=en-GB --service-sandbox-type=none --no-sandbox --enable-crash-
reporter=18887fa1-4d37-46f0-bf9f-b37513c81cf9,no_channel --user-data-dir=/home/abhijit/.config/Signal --shared-
files=v8_context_snapshot_data:100 --field-trial-handle=0,i,3213315393136307567,15155041764863120512,262144 --disable-
features=HardwareMediaKeyHandling,SpareRendererForSitePerProcess
abhijit   552103  552002  1 Jan12 ?        00:57:51 /opt/Signal/signal-desktop --type=renderer --enable-crash-
reporter=18887fa1-4d37-46f0-bf9f-b37513c81cf9,no_channel --user-data-dir=/home/abhijit/.config/Signal --app-
path=/opt/Signal/resources/app.asar --no-sandbox --no-zygote --enable-blink-features=CSSPseudoDir,CSSLogical --disable-
blink-features=Accelerated2dCanvas,AcceleratedSmallCanvases --first-renderer-process --no-sandbox --disable-gpu-
compositing --lang=en-GB --num-raster-threads=4 --enable-main-frame-before-activation --renderer-client-id=4 --time-
ticks-at-unix-epoch=-1704847690836396 --launch-time-ticks=218630770368 --shared-files=v8_context_snapshot_data:100 --
field-trial-handle=0,i,3213315393136307567,15155041764863120512,262144 --disable-
features=HardwareMediaKeyHandling,SpareRendererForSitePerProcess
abhijit   552149  552002  0 Jan12 ?        00:00:12 /opt/Signal/signal-desktop --type=utility --utility-sub-
type=audio.mojom.AudioService --lang=en-GB --service-sandbox-type=none --no-sandbox --enable-crash-reporter=18887fa1-
4d37-46f0-bf9f-b37513c81cf9,no_channel --user-data-dir=/home/abhijit/.config/Signal --shared-
files=v8_context_snapshot_data:100 --field-trial-handle=0,i,3213315393136307567,15155041764863120512,262144 --disable-
features=HardwareMediaKeyHandling,SpareRendererForSitePerProcess
abhijit   554660  351306  0 Jan13 ?        00:04:20 /usr/lib/firefox/firefox -contentproc -childID 442 -isForBrowser -
prefsLen 35109 -prefMapSize 247458 -jsInitLen 229864 -parentBuildID 20240108143603 -greomni /usr/lib/firefox/omni.ja -
appomni /usr/lib/firefox/browser/omni.ja -appDir /usr/lib/firefox/browser {2cffa6f3-6f8c-4a3f-880d-01cc71baf983} 351306
true tab
abhijit   554855  351306  4 Jan13 ?        03:24:06 /usr/lib/firefox/firefox -contentproc -childID 445 -isForBrowser -
prefsLen 35109 -prefMapSize 247458 -jsInitLen 229864 -parentBuildID 20240108143603 -greomni /usr/lib/firefox/omni.ja -
appomni /usr/lib/firefox/browser/omni.ja -appDir /usr/lib/firefox/browser {abcba8e9-5b97-4281-ad64-1356841dcf34} 351306
true tab
abhijit   556349  351306  0 Jan13 ?        00:01:18 /usr/lib/firefox/firefox -contentproc -childID 451 -isForBrowser -
prefsLen 35109 -prefMapSize 247458 -jsInitLen 229864 -parentBuildID 20240108143603 -greomni /usr/lib/firefox/omni.ja -
appomni /usr/lib/firefox/browser/omni.ja -appDir /usr/lib/firefox/browser {6318d453-643c-4b85-9f0f-bfd0a429cd41} 351306
true tab
```

```
abhijit   557600    3629  0 Jan13 ?        00:00:08 /usr/lib/speech-dispatcher-modules/sd_espeak-ng /etc/speech-
dispatcher/modules/espeak-ng.conf
abhijit   557607    3629  0 Jan13 ?        00:00:08 /usr/lib/speech-dispatcher-modules/sd_dummy /etc/speech-
dispatcher/modules/dummy.conf
abhijit   557610    3629  0 Jan13 ?        00:00:08 /usr/lib/speech-dispatcher-modules/sd_generic /etc/speech-
dispatcher/modules/mary-generic.conf
abhijit   557613    3629  0 Jan13 ?        00:00:00 /usr/bin/speech-dispatcher --spawn --communication-method
unix_socket --socket-path /run/user/1000/speech-dispatcher/speechd.sock
abhijit   571320  351306  0 Jan13 ?        00:01:16 /usr/lib/firefox/firefox -contentproc -childID 486 -isForBrowser -
prefsLen 35110 -prefMapSize 247458 -jsInitLen 229864 -parentBuildID 20240108143603 -greomni /usr/lib/firefox/omni.ja -
appomni /usr/lib/firefox/browser/omni.ja -appDir /usr/lib/firefox/browser {71966a59-b1df-4c44-975c-62d7cd41188c} 351306
true tab
abhijit   571410  351306  0 Jan13 ?        00:01:34 /usr/lib/firefox/firefox -contentproc -childID 488 -isForBrowser -
prefsLen 35110 -prefMapSize 247458 -jsInitLen 229864 -parentBuildID 20240108143603 -greomni /usr/lib/firefox/omni.ja -
appomni /usr/lib/firefox/browser/omni.ja -appDir /usr/lib/firefox/browser {bfbc15ff-6ca5-4837-b97e-334e0bfc8855} 351306
true tab
abhijit   571582  351306  0 Jan13 ?        00:03:32 /usr/lib/firefox/firefox -contentproc -childID 492 -isForBrowser -
prefsLen 35110 -prefMapSize 247458 -jsInitLen 229864 -parentBuildID 20240108143603 -greomni /usr/lib/firefox/omni.ja -
appomni /usr/lib/firefox/browser/omni.ja -appDir /usr/lib/firefox/browser {04e791e5-9cde-4e03-9211-141bdb440139} 351306
true tab
abhijit   591339 4139495  0 Jan13 pts/9    00:00:00 vi mh-pyt.txt
root      594356       2  0 Jan13 ?        00:00:00 [dio/nvme0n1p7]
abhijit   594726   30565  0 Jan13 pts/5    00:00:00 bash
abhijit   791421  351306  1 Jan14 ?        00:40:27 /usr/lib/firefox/firefox -contentproc -childID 839 -isForBrowser -
prefsLen 35227 -prefMapSize 247458 -jsInitLen 229864 -parentBuildID 20240108143603 -greomni /usr/lib/firefox/omni.ja -
appomni /usr/lib/firefox/browser/omni.ja -appDir /usr/lib/firefox/browser {b6f69be3-7ebe-43c9-9d55-c72161180b5e} 351306
true tab
abhijit   794226  351306  0 Jan14 ?        00:02:03 /usr/lib/firefox/firefox -contentproc -childID 848 -isForBrowser -
prefsLen 35227 -prefMapSize 247458 -jsInitLen 229864 -parentBuildID 20240108143603 -greomni /usr/lib/firefox/omni.ja -
appomni /usr/lib/firefox/browser/omni.ja -appDir /usr/lib/firefox/browser {28fddc81-ea39-4496-a1b6-b70d9a8a9c31} 351306
true tab
abhijit   797871  351306  0 Jan14 ?        00:02:41 /usr/lib/firefox/firefox -contentproc -childID 851 -isForBrowser -
prefsLen 35226 -prefMapSize 247458 -jsInitLen 229864 -parentBuildID 20240108143603 -greomni /usr/lib/firefox/omni.ja -
appomni /usr/lib/firefox/browser/omni.ja -appDir /usr/lib/firefox/browser {b254a3c7-e1f7-4245-9fcf-820074a4e69f} 351306
true tab
abhijit   799607   30565  0 Jan14 pts/11   00:00:00 bash
abhijit   803503   30565  0 Jan14 pts/12   00:00:00 bash
abhijit   815513  799607  0 Jan15 pts/11   00:00:00 vi timetable-todo2
abhijit   821738  351306  0 Jan15 ?        00:07:30 /usr/lib/firefox/firefox -contentproc -childID 995 -isForBrowser -
prefsLen 35227 -prefMapSize 247458 -jsInitLen 229864 -parentBuildID 20240108143603 -greomni /usr/lib/firefox/omni.ja -
appomni /usr/lib/firefox/browser/omni.ja -appDir /usr/lib/firefox/browser {90cad852-941e-44b3-8fa4-0d44cbcfda7a} 351306
true tab
abhijit   895193  351306  0 Jan15 ?        00:00:04 /usr/lib/firefox/firefox -contentproc -childID 1131 -isForBrowser -
prefsLen 35227 -prefMapSize 247458 -jsInitLen 229864 -parentBuildID 20240108143603 -greomni /usr/lib/firefox/omni.ja -
appomni /usr/lib/firefox/browser/omni.ja -appDir /usr/lib/firefox/browser {218262bd-4f9e-423a-9368-4d2e44f33125} 351306
true tab
root      942398       1  0 10:40 ?        00:00:00 /usr/sbin/cupsd -l
root      942399       1  0 10:40 ?        00:00:00 /usr/sbin/cups-browsed
www-data  942465    1420  0 10:40 ?        00:00:00 /usr/sbin/apache2 -k start
www-data  942466    1420  0 10:40 ?        00:00:00 /usr/sbin/apache2 -k start
www-data  942468    1420  0 10:40 ?        00:00:00 /usr/sbin/apache2 -k start
www-data  942469    1420  0 10:40 ?        00:00:00 /usr/sbin/apache2 -k start
www-data  942470    1420  0 10:40 ?        00:00:00 /usr/sbin/apache2 -k start
www-data  942471    1420  0 10:40 ?        00:00:00 /usr/sbin/apache2 -k start
abhijit   954109   30565  0 11:21 pts/2    00:00:00 bash
abhijit   961628  351306  3 12:36 ?        00:05:34 /usr/lib/firefox/firefox -contentproc -childID 1704 -isForBrowser -
prefsLen 35227 -prefMapSize 247458 -jsInitLen 229864 -parentBuildID 20240108143603 -greomni /usr/lib/firefox/omni.ja -
appomni /usr/lib/firefox/browser/omni.ja -appDir /usr/lib/firefox/browser {cb82b978-ef9d-4a76-b59c-5b13b652a0ec} 351306
true tab
root      961877       2  0 12:36 ?        00:00:02 [kworker/u32:5-events_unbound]
root      962113       2  0 12:39 ?        00:00:08 [kworker/u33:3-hci0]
abhijit   968060  351306  0 13:09 ?        00:00:05 /usr/lib/firefox/firefox -contentproc -childID 1749 -isForBrowser -
prefsLen 35227 -prefMapSize 247458 -jsInitLen 229864 -parentBuildID 20240108143603 -greomni /usr/lib/firefox/omni.ja -
appomni /usr/lib/firefox/browser/omni.ja -appDir /usr/lib/firefox/browser {c0ba3673-966e-48a6-b029-91346e348342} 351306
true tab
root      968299       2  0 13:11 ?        00:00:02 [kworker/u32:4-events_unbound]
root      969560       2  0 13:23 ?        00:00:01 [kworker/5:1-cgroup_destroy]
root      969608       2  0 13:24 ?        00:00:00 [kworker/u32:7-events_unbound]
root      969648       2  0 13:24 ?        00:00:01 [kworker/2:0-inet_frag_wq]
abhijit   969715  379620  0 13:25 pts/6    00:00:00 ssh root@10.1.101.41
```

```
root      970435       2  0 13:27 ?        00:00:01 [kworker/3:2-rcu_gp]
root      971921       2  0 13:33 ?        00:00:00 [kworker/1:2-rcu_gp]
root      972048       2  0 13:35 ?        00:00:00 [kworker/7:2-rcu_gp]
root      972127       2  0 13:37 ?        00:00:01 [kworker/0:1-events_long]
root      972207       2  0 13:37 ?        00:00:01 [kworker/4:1-events]
root      972378       2  0 13:39 ?        00:00:00 [kworker/1:0-cgroup_destroy]
root      972435       2  0 13:39 ?        00:00:00 [kworker/6:2-events]
root      972964       2  0 13:41 ?        00:00:00 [kworker/7:0-events]
root      973166       2  0 13:41 ?        00:00:00 [kworker/u32:0-events_unbound]
root      973193       2  0 13:42 ?        00:00:00 [kworker/2:1-events]
root      973282       2  0 13:43 ?        00:00:00 [kworker/4:2-events]
root      973384       2  0 13:44 ?        00:00:00 [kworker/3:0-rcu_gp]
root      973389       2  0 13:44 ?        00:00:01 [kworker/u33:0-hci0]
root      973391       2  0 13:44 ?        00:00:00 [kworker/6:0-rcu_gp]
root      973392       2  0 13:44 ?        00:00:00 [kworker/5:2-events]
root      973655       2  0 13:45 ?        00:00:00 [kworker/1:1-events]
root      973664       2  0 13:46 ?        00:00:00 [kworker/7:1-events]
root      973965       2  0 13:47 ?        00:00:00 [kworker/2:2-events]
root      974126       2  0 13:48 ?        00:00:00 [kworker/u32:1-kcryptd/253:0]
root      974189       2  0 13:48 ?        00:00:00 [kworker/4:0-events]
root      974190       2  0 13:48 ?        00:00:00 [kworker/0:2-events]
root      974369       2  0 13:49 ?        00:00:00 [kworker/u32:2-nvme-wq]
root      974539       2  0 13:49 ?        00:00:00 [kworker/u32:3-events_unbound]
root      974655       2  0 13:50 ?        00:00:00 [kworker/6:1-events]
root      974690       2  0 13:50 ?        00:00:00 [kworker/5:0-events]
root      974740       2  0 13:50 ?        00:00:00 [kworker/7:3-events]
root      974742       2  0 13:50 ?        00:00:00 [kworker/3:1-pm]
root      974743       2  0 13:50 ?        00:00:00 [kworker/u32:6-events_unbound]
root      974744       2  0 13:50 ?        00:00:00 [kworker/u32:8-kcryptd/253:0]
root      974745       2  0 13:50 ?        00:00:00 [kworker/u32:9-events_unbound]
root      974746       2  0 13:50 ?        00:00:00 [kworker/u32:10-events_unbound]
root      974747       2  0 13:50 ?        00:00:00 [kworker/u32:11-events_unbound]
root      974748       2  0 13:50 ?        00:00:00 [kworker/u32:12-kcryptd/253:0]
root      974749       2  0 13:50 ?        00:00:00 [kworker/u32:13-events_unbound]
root      974750       2  0 13:50 ?        00:00:00 [kworker/u32:14-events_unbound]
root      974751       2  0 13:50 ?        00:00:00 [kworker/u32:15-events_unbound]
root      974752       2  0 14:18 ?        00:00:00 [kworker/u32:16-events_unbound]
root      974753       2  0 14:18 ?        00:00:00 [kworker/u32:17-events_unbound]
root      974754       2  0 14:18 ?        00:00:00 [kworker/u32:18-events_unbound]
root      974755       2  0 14:18 ?        00:00:00 [kworker/u32:19-events_unbound]
root      974756       2  0 14:18 ?        00:00:00 [kworker/u32:20-kcryptd/253:0]
root      974757       2  0 14:18 ?        00:00:00 [kworker/u32:21-events_unbound]
root      974758       2  0 14:18 ?        00:00:00 [kworker/u32:22-events_unbound]
root      974759       2  0 14:18 ?        00:00:00 [kworker/u32:23-events_unbound]
root      974760       2  0 14:18 ?        00:00:00 [kworker/u32:24-events_unbound]
root      974761       2  0 14:18 ?        00:00:00 [kworker/u32:25-events_unbound]
root      974762       2  0 14:18 ?        00:00:00 [kworker/u32:26-events_unbound]
root      974763       2  0 14:18 ?        00:00:00 [kworker/u32:27-kcryptd/253:0]
root      974764       2  0 14:18 ?        00:00:00 [kworker/u32:28-kcryptd/253:0]
root      974765       2  0 14:18 ?        00:00:00 [kworker/u32:29-events_unbound]
root      974766       2  0 14:18 ?        00:00:00 [kworker/u32:30+events_unbound]
root      974767       2  0 14:18 ?        00:00:00 [kworker/u32:31-events_unbound]
root      974768       2  0 14:18 ?        00:00:00 [kworker/u32:32-events_unbound]
root      974769       2  0 14:18 ?        00:00:00 [kworker/u32:33-events_unbound]
root      974770       2  0 14:18 ?        00:00:00 [kworker/u32:34-events_unbound]
root      974771       2  0 14:18 ?        00:00:00 [kworker/u32:35-events_unbound]
root      974772       2  0 14:18 ?        00:00:00 [kworker/u32:36-events_unbound]
root      974773       2  0 14:18 ?        00:00:00 [kworker/u32:37-events_unbound]
root      974774       2  0 14:18 ?        00:00:00 [kworker/u32:38-events_unbound]
root      974775       2  0 14:18 ?        00:00:00 [kworker/u32:39-events_unbound]
root      974776       2  0 14:18 ?        00:00:00 [kworker/u32:40-kcryptd/253:0]
root      974778       2  0 14:18 ?        00:00:00 [kworker/u32:42-events_unbound]
root      974779       2  0 14:18 ?        00:00:00 [kworker/3:3-events]
root      974780       2  0 14:18 ?        00:00:00 [kworker/3:4-events]
root      974798       2  0 14:18 ?        00:00:00 [kworker/3:5-events]
root      974995       2  0 14:18 ?        00:00:00 [kworker/u33:2-rb_allocator]
root      975505       2  0 14:20 ?        00:00:00 [kworker/6:3-events]
root      975656       2  0 14:20 ?        00:00:00 [kworker/5:3-events]
root      975657       2  0 14:29 ?        00:00:00 [kworker/1:3-pm]
root      975658       2  0 14:29 ?        00:00:00 [kworker/1:4-events]
```

```
abhijit   975722  351306  0 14:29 ?        00:00:02 /usr/lib/firefox/firefox -contentproc -childID 1836 -isForBrowser -
prefsLen 35227 -prefMapSize 247458 -jsInitLen 229864 -parentBuildID 20240108143603 -greomni /usr/lib/firefox/omni.ja -
appomni /usr/lib/firefox/browser/omni.ja -appDir /usr/lib/firefox/browser {d9ea12ea-5b3f-477c-9c08-9a4196ea8d04} 351306
true tab
root      976242       2  0 15:16 ?        00:00:00 [kworker/0:0-events]
root      976243       2  0 15:16 ?        00:00:00 [kworker/0:3-events]
root      976244       2  0 15:16 ?        00:00:00 [kworker/0:4-events]
postfix   976248    3508  0 15:16 ?        00:00:00 pickup -l -t unix -u -c
abhijit   976770  351306  0 15:18 ?        00:00:00 /usr/lib/firefox/firefox -contentproc -childID 1840 -isForBrowser -
prefsLen 35227 -prefMapSize 247458 -jsInitLen 229864 -parentBuildID 20240108143603 -greomni /usr/lib/firefox/omni.ja -
appomni /usr/lib/firefox/browser/omni.ja -appDir /usr/lib/firefox/browser {cb567d37-1c03-46db-966c-632f4b708354} 351306
true tab
abhijit   976836  351306  0 15:19 ?        00:00:00 /usr/lib/firefox/firefox -contentproc -childID 1841 -isForBrowser -
prefsLen 35227 -prefMapSize 247458 -jsInitLen 229864 -parentBuildID 20240108143603 -greomni /usr/lib/firefox/omni.ja -
appomni /usr/lib/firefox/browser/omni.ja -appDir /usr/lib/firefox/browser {b62e1986-221e-481e-bda4-62fb37caaa67} 351306
true tab
abhijit   977138  351306  0 15:20 ?        00:00:00 /usr/lib/firefox/firefox -contentproc -childID 1842 -isForBrowser -
prefsLen 35227 -prefMapSize 247458 -jsInitLen 229864 -parentBuildID 20240108143603 -greomni /usr/lib/firefox/omni.ja -
appomni /usr/lib/firefox/browser/omni.ja -appDir /usr/lib/firefox/browser {989da731-7bd5-44ce-abd0-200ca4c84b9b} 351306
true tab
abhijit   977184  594726  0 15:21 pts/5    00:00:00 ps -eaf
abhijit  1664880    3629  0 Jan07 ?        00:00:06 /usr/bin/gnome-calendar --gapplication-service
abhijit  1665340    3629  0 Jan07 ?        00:00:00 /usr/bin/gpg-agent --supervised
abhijit  3872409    3629  0 Jan07 ?        00:00:05 /usr/bin/seahorse --gapplication-service
root     3873244       1  0 Jan07 ?        00:00:55 /sbin/mount.ntfs /dev/nvme0n1p3 /media/abhijit/windows -o
rw,nodev,nosuid,windows_names,uid=1000,gid=1000,uhelper=udisks2
abhijit  3884359   30565  0 Jan07 pts/7    00:00:00 bash
abhijit  4108623   30565  0 Jan08 pts/8    00:00:00 bash
abhijit  4136834    3890  0 Jan08 ?        00:00:00 /usr/lib/libreoffice/program/oosplash --calc
abhijit  4136869 4136834  0 Jan08 ?        00:32:11 /usr/lib/libreoffice/program/soffice.bin --calc
abhijit  4139495   30565  0 Jan08 pts/9    00:00:00 bash
```

The PID of the grand-parent of the process with PID 4108623 is :

3629

✔

The two processes which were created by kernel have PIDs (in increasing order)

0

✖ and

1

✖

The process that created most of the "graphical" processes is having PID

2

✖

Question **6**

Partially correct

Mark 0.67 out of 1.00

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

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

Your answer is partially correct.

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

Question **7**

Correct

Mark 0.50 out of 0.50

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

- a. yes
- b. no ✔ wrong

The correct answer is: no

Question **8**

Correct

Mark 1.00 out of 1.00

Consider the following programs

exec1.c

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

exec2.c

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

Compiled as

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

And run as

```
$ ./exec1
```

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

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

Select one:

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

Your answer is correct.

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

Question **9**

Correct

Mark 0.50 out of 0.50

When you turn your computer ON, on BIOS based systems, you are often shown an option like "Press F9 for boot options". What does this mean?

- a. The choice of booting slowly or fast
- b. The BIOS allows us to choose the boot device, the device from which the boot loader will be loaded✔
- c. The choice of which OS to boot from
- d. The choice of the boot loader (e.g. GRUB or Windows-Loader)

The correct answer is: The BIOS allows us to choose the boot device, the device from which the boot loader will be loaded

Question **10**

Correct

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

Your answer is correct.

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

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.

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

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 **13**

Correct

Mark 1.00 out of 1.00

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

Your answer is correct.

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 **14**

Partially correct

Mark 0.60 out of 1.00

Select all the correct statements about two modes of CPU operation

Select one or more:

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

Your answer is partially correct.

You have correctly selected 3.
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 **15**

Partially correct

Mark 0.25 out of 0.50

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

Your answer is partially correct.

You have selected too many options.
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 **16**

Partially correct

Mark 0.33 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 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
- [ ] b. P1 running
  keyboard hardware interrupt
  keyboard interrupt handler running
  interrupt handler returns
  P1 running
  P1 makes sytem call
  system call returns
  P1 running
  timer interrupt
  scheduler
  P2 running
- [x] c. P1 running ✔
  P1 makes sytem call and blocks
  Scheduler
  P2 running
  P2 makes sytem call and blocks
  Scheduler
  P1 running again
- [ ] d.
  P1 running
  P1 makes sytem call
  Scheduler
  P2 running
  P2 makes sytem call and blocks
  Scheduler
  P1 running again
- [ ] e. P1 running
  P1 makes system call
  system call returns
  P1 running
  timer interrupt
  Scheduler running
  P2 running
- [ ] f. P1 running
  P1 makes system call
  timer interrupt
  Scheduler
  P2 running
  timer interrupt
  Scheuler
  P1 running
  P1's system call return

Your answer is partially correct.

You have correctly selected 1.
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 **17**

Correct

Mark 1.00 out of 1.00

What will this program do?

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

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

Your answer is correct.

The correct answer is: run ls twice

◄ Surprise Quiz - 1 (pre-requisites)

Jump to...

Surprise Quiz - 3 (processes, memory management, event driven kernel), compilation-linking-loading) ►

Started on	Wednesday, 7 February 2024, 6:09 PM
State	Finished
Completed on	Wednesday, 7 February 2024, 7:10 PM
Time taken	1 hour
Grade	**18.14** out of 20.00 (**90.68**%)

Question **1**

Correct

Mark 1.00 out of 1.00

Consider the following code and MAP the file to which each fd points at the end of the code. Assume that files/folders exist when needed with proper permissions and open() calls work.

```
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;
}
```

fd2	/tmp/2 ✔
fd1	/tmp/1 ✔
0	/tmp/2 ✔
fd3	closed ✔
2	stderr ✔
1	/tmp/3 ✔
fd4	/tmp/2 ✔

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

Question **2**

Partially correct

Mark 1.43 out of 2.00

Order the events that occur on a timer interrupt:

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

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

Question **3**

Partially correct

Mark 0.75 out of 1.00

Select the sequence of events that are NOT possible, assuming an interruptible kernel code

Select one or more:

- [ ] a. 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

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

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

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

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

- [ ] f. P1 running
  P1 makes system call
  system call returns

P1 running
timer interrupt
Scheduler running
P2 running

The correct answers are: P1 running
P1 makes sytem call and blocks
Scheduler
P2 running
P2 makes sytem call and blocks
Scheduler
P1 running again,
P1 running
P1 makes sytem call
Scheduler
P2 running
P2 makes sytem call and blocks
Scheduler
P1 running again

Question **4**

Partially correct

Mark 1.60 out of 2.00

Match the elements of C program to their place in memory

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

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

Question **5**

Correct

Mark 1.00 out of 1.00

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

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

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

Question **6**

Correct

Mark 2.00 out of 2.00

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

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 7

Correct

Mark 1.00 out of 1.00

Select all the correct statements about zombie processes

Select one or more:

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

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

Question **8**

Correct

Mark 1.00 out of 1.00

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

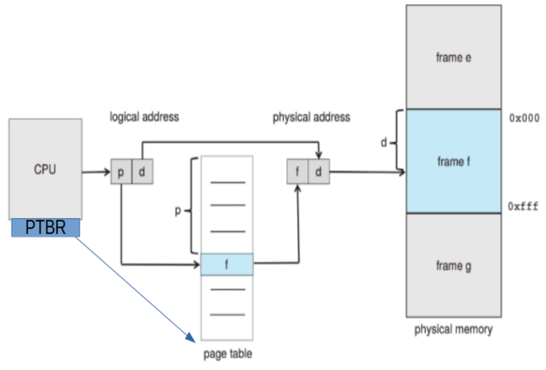


Figure 9.8 Paging hardware.

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

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

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

Question **9**

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

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 **10**

Partially correct

Mark 0.86 out of 1.00

Mark the statements as True/False w.r.t. the basic concepts of memory management.

True	False		
○ ✗	◉ ✓	When a process is executing, each virtual address is converted into physical address by the kernel directly.	✓
◉ ✓	○ ✗	The compiler generates address references for code/data/stack/heap in the executable file, depending on the MM architecture provided by CPU and kernel.	✓
○ ✗	◉ ✓	The compiler generates the address references for code/data/stack/heap in the executable file as per the memory management schema chosen by the compiler itself, and then the kernel ensures that program is executed with this schema.	✓
○ ✗	◉ ✓	The compiler interacts with the kernel continuously while compiling a program and obtains the correct set of memory addresses for code/stack/heap/data and then generates the machine code file.	✓
◉ ✓	○ ✗	The kernel ensures that the MMU is setup before scheduling a process and then the CPU/MMU ensures that the address translation takes place.	✓
◉ ✓	○ ✗	When a process is executing, each virtual address is converted into physical address by the CPU hardware directly.	✓
◉ ✗	○ ✓	The kernel refers to the page table for converting each virtual address to physical address.	✗

When a process is executing, each virtual address is converted into physical address by the kernel directly.: False
The compiler generates address references for code/data/stack/heap in the executable file, depending on the MM architecture provided by CPU and kernel.: True
The compiler generates the address references for code/data/stack/heap in the executable file as per the memory management schema chosen by the compiler itself, and then the kernel ensures that program is executed with this schema.: False
The compiler interacts with the kernel continuously while compiling a program and obtains the correct set of memory addresses for code/stack/heap/data and then generates the machine code file.: False
The kernel ensures that the MMU is setup before scheduling a process and then the CPU/MMU ensures that the address translation takes place.: True
When a process is executing, each virtual address is converted into physical address by the CPU hardware directly.: True
The kernel refers to the page table for converting each virtual address to physical address.: False

Question **11**

Partially correct

Mark 0.50 out of 1.00

Select the correct statements about paging (not demand paging) mechanism

Select one or more:

- [ ] a. User process can update it's own PTBR
- [ ] b. An invalid entry on a page means, it was an illegal memory reference
- [x] c. The PTBR is loaded by the OS ✔
- [ ] d. Page table is accessed by the OS as part of execuation of an instruction
- [ ] e. User process can update it's own page table entries
- [x] f. Page table is accessed by the MMU as part of execution of an instruction ✔
- [x] g. OS creates the page table for every process ✔
- [x] h. An invalid entry on a page means, either it was illegal memory reference or the page was not present in memory. ✖

The correct answers are: OS creates the page table for every process, The PTBR is loaded by the OS, Page table is accessed by the MMU as part of execution of an instruction, An invalid entry on a page means, it was an illegal memory reference

Question **12**

Correct

Mark 1.00 out of 1.00

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	many continuous chunks of variable size	✔
Segmentation, then paging	many continuous chunks of variable size	✔
Relocation + Limit	one continuous chunk	✔
Paging	one continuous chunk	✔

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

Question **13**

Correct

Mark 1.00 out of 1.00

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 **14**

Correct

Mark 1.00 out of 1.00

A process blocks itself means

- a. The kernel code of system call, called by the process, moves the process to a waiting queue and calls scheduler ✔
- b. The application code calls the scheduler
- c. The kernel code of an interrupt handler, 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 **15**

Correct

Mark 1.00 out of 1.00

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

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

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

Question **16**

Correct

Mark 1.00 out of 1.00

Predict the output of the program given here.

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`

`--`

```
int main() {
  int pid;
  printf("hi\n");
  pid = fork();
  if(pid == 0) {
    exit(0);
  }
  printf("bye\n");
  fork();
  printf("ok\n");
}
```

Answer: hi bye ok ok

The correct answer is: hi bye ok ok

Question **17**

Correct

Mark 1.00 out of 1.00

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

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

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

◄ Surprise Quiz - 2

Jump to...

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

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:

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

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

Dashboard / My courses / Computer Engineering & IT / CEIT-Even-sem-20-21 / OS-Even-sem-2020-21 / 14 March - 20 March / Quiz - 2 (18 March)

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.

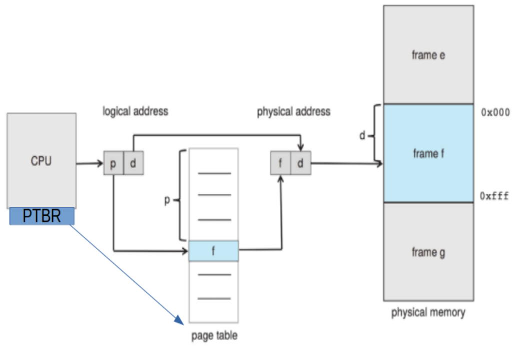


Figure 9.8 Paging hardware.

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...

Question **14**

Not yet answered

Marked out of 3.00

Time left 1:53:37

List down all changes required to xv6 code, in order to add the system call chown().

Every change should be mentioned in terms of either of the following:

(a) pseudo-code of new function to be added

(b) prototype of any new function or new system call to be added

(c) pseudo-code of changes to an existing function, describing lines to be removed, and lines to be added

(d) **precise** declaration of new data structures to be added in C, or changes to the existing data structure

(e) Name and a one-line description of new userland functionality to be added

(f) Changes to Makefile

(g) Any other change in a maximum of 20 words per change.

Paragraph

(a) int chown (char * path , char *ownername);

(d)there is no need of new data structures or changes to the existing data structure

(e)by using this system call ,user to change the owner if the file

(f)_trychown\ at the end of UPROGS

Path: p

◄ Random Quiz - 6 (xv6 file system)

Jump to...

Homework questions: Basics of MM, xv6 booting ►

Question **18**

Not yet answered

Marked out of 2.00

Time left 1:34:55

Write all changes required to xv6 to add a buddy allocator.

Every change should be mentioned in terms of either of the following:

(a) pseudo-code of new function to be added

(b) prototype of any new function or new system call to be added

(c) pseudo-code of changes to an existing function, describing lines to be removed, and lines to be added

(d) **precise** declaration of new data structures to be added in C, or changes to the existing data structure

(e) Name and a one-line description of new userland functionality to be added

(f) Changes to Makefile

(g) Any other change in a maximum of 20 words per change.

Paragraph

Path: p

◄ Random Quiz - 6 (xv6 file system)

Jump to...

Homework questions: Basics of MM, xv6 booting ►

Question **17**

Not yet answered

Marked out of 1.00

Match the code with it's functionality

S1 = 0; S2 = 0;
P2:
Statement1;
Signal(S2);

P1:
Wait(S2);
Statemetn2;
Signal(S1);

P3:
Wait(S1);
Statement S3;

Execution order P1, P2, P3

Time left 1:35:43

S = 0
P1:
Statement1;
Signal(S)

P2:
Wait(S)
Statment2;

Execution order P3, P2, P1

S = 5
Wait(S)
Critical Section
Signal(S)

Execution order P2, then P1

S = 1
Wait(S)
Critical Section
Signal(S);

Counting semaphore

Time left 1:48:32

Question **13**

Not yet answered

Marked out of 2.00

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

"..." means some code.

```
void
yield(void)
{
...
release(&ptable.lock);
}
```

Ensure that no printing happens on other processors

```
struct proc*
myproc(void) {
...
  pushcli();
  c = mycpu();
  p = c->proc;
  popcli();
...
}
```

Disable interrupts to avoid another process's pointer being returned

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

Avoid a self-deadlock

```
void
acquire(struct spinlock *lk)
{
...
  getcallerpcs(&lk, lk->pcs);
```

Disable interrupts to avoid deadlocks

```
void
acquire(struct spinlock *lk)
{
  pushcli();
```

Traverse ebp chain to get sequence of instructions followed in functions calls

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

Release the lock held by some another process

```
void
panic(char *s)
{
...
  panicked = 1;
```

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

◄ Random Quiz - 6 (xv6 file system)

Jump to...

Homework questions: Basics of MM, xv6 booting ►

Question **12**

Not yet answered

Marked out of 2.00

Select all the correct statements about synchronization primitives.

Time left 2:04:57

Select one or more:

- [ ] a. Thread that is going to block should not be holding any spinlock
- [x] b. Semaphores can be used for synchronization scenarioes like ordered execution
- [x] c. Mutexes can be implemented using spinlock
- [ ] d. Blocking means one process passing over control to another process
- [ ] e. Semaphores are always a good substitute for spinlocks
- [ ] f. Blocking means moving the process to a wait queue and spinning
- [ ] g. Blocking means moving the process to a wait queue and calling scheduler
- [ ] h. All synchronization primitives are implemented essentially with some hardware assistance.
- [ ] i. Spinlocks are good for multiprocessor scenarios, for small critical sections
- [ ] j. Mutexes can be implemented without any hardware assistance
- [x] k. Spinlocks consume CPU time
- [x] l. Mutexes can be implemented using blocking and wakeup

◄ Random Quiz - 6 (xv6 file system)

Jump to...

Homework questions: Basics of MM, xv6 booting ►

Question **10**

Not yet answered

Marked out of 1.00

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

Time left 2:12:36

```
void
yield(void)
{
...
release(&ptable.lock);
}
```

Release the lock held by some another process

```
void
panic(char *s)
{
...
  panicked = 1;
```

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

```
void
acquire(struct spinlock *lk)
{
...
  getcallerpcs(&lk, lk->pcs);
```

Disable interrupts to avoid another process's pointer being returned

◄ Random Quiz - 6 (xv6 file system)

Jump to...

Homework questions: Basics of MM, xv6 booting ►

Time left 2:50:14

Question **3**

Not yet answered

Marked out of 1.00

Match each suggested semaphore implementation (discussed in class)

with the problems that it faces

```
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) {
                spinunlock(&(s->sl));
                spinlock(&(s->sl));
        }
        (s->val)--;
        spinunlock(&(s->sl));
}
```

Choose...

```
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));
}
```

Choose...

```
struct semaphore {
        int val;
        spinlock lk;
        list l;
};
sem_init(semaphore *s, int initval) {
        s->val = initval;
        s->sl = 0;
}
block(semaphore *s) {
        listappend(s->l, current);
        spinunlock(&(s->sl));
        schedule();
}
wait(semaphore *s) {
        spinlock(&(s->sl));
        while(s->val <=0) {
                block(s);
        }
        (s->val)--;
        spinunlock(&(s->sl));
}
signal(seamphore *s) {
        spinlock(*(s->sl));
        (s->val)++;
        x = dequeue(s->sl) and enqueue(readyq, x);
        spinunlock(*(s->sl));
}
```

Choose...

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

Choose...

Time left 2:50:14

Question 3

Not yet answered

Marked out of 1.00

Match each suggested semaphore implementation (discussed in class)

with the problems that it faces

```
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) {
                spinunlock(&(s->sl));
                spinlock(&(s->sl));
        }
        (s->val)--;
        spinunlock(&(s->sl));
}
```

Choose...

```
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));
}
```

Choose...

```
struct semaphore {
	int val;
	spinlock lk;
	list l;
};
sem_init(semaphore *s, int initval) {
	s->val = initval;
	s->sl = 0;
}
block(semaphore *s) {
	listappend(s->l, current);
	spinunlock(&(s->sl));
	schedule();
}
wait(semaphore *s) {
	spinlock(&(s->sl));
	while(s->val <=0) {
		block(s);
	}
	(s->val)--;
	spinunlock(&(s->sl));
}
signal(seamphore *s) {
	spinlock(*(s->sl));
	(s->val)++;
	x = dequeue(s->sl) and enqueue(readyq, x);
	spinunlock(*(s->sl));
}
```

Choose...

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

Choose...

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.

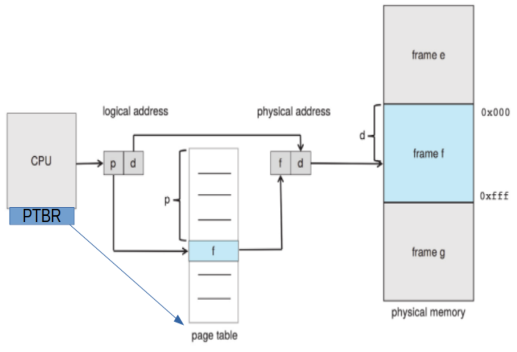


Figure 9.8 Paging hardware.

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...

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:

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

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

Dashboard / My courses / Computer Engineering & IT / CEIT-Even-sem-21-22 / OS-even-sem-21-22 / 21 March - 27 March / Quiz-2

Started on	Tuesday, 22 March 2022, 2:06:46 PM
State	Finished
Completed on	Tuesday, 22 March 2022, 5:29:11 PM
Time taken	3 hours 22 mins
Grade	**23.12** out of 40.00 (**58**%)

Question **1**

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: possible ✖

Because

Process P2 has to return from interrupt context before it's user code executes ✔

Question **2**

Correct

Mark 1.00 out of 1.00

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

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

Your answer is correct.

The correct answer is: Singly linked NULL terminated list

Question **3**

Incorrect

Mark 0.00 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.

220 KB, best fit	100 KB	✖
50 KB, worst fit	100 KB	✖
100 KB, worst fit	100 KB	✖
150 KB, best fit	100 KB	✖
200 KB, first fit	100 KB	✖
150 KB, first fit	100 KB	✖

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

Question **4**

Incorrect

Mark 0.00 out of 1.00

Select the most common causes of use of IPC by processes

- [ ] a. Sharing of information of common interest
- [x] b. Breaking up a large task into small tasks and speeding up computation, on multiple core machines ✔
- [ ] c. More modular code
- [ ] d. More security checks
- [x] 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 **5**

Partially correct

Mark 0.33 out of 1.00

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

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

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 **6**

Correct

Mark 1.00 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: 11111010 ✔

The correct answer is: 11111010

Question **7**

Partially correct

Mark 0.67 out of 1.00

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

SLEEPING	sleep(), called by any process blocking itself	✔
EMBRYO	fork()->allocproc() before setting up the UVM	✔
UNUSED	exit(), called by an interrupt handler	✘
ZOMBIE	exit(), called by process itself	✔
RUNNING	fork()->allocproc() before setting up the UVM	✘
RUNNABLE	wakeup(), called by an interrupt handler	✔

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

Question **8**

Partially correct

Mark 0.67 out of 1.00

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

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

Your answer is partially correct.

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

Question **9**

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 **10**

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		
(selected) ✔	✘	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).	✔
✔	(selected) ✘	Local replacement results in more predictable per-process completion time because number of page faults can be better predicted.	✘
(selected) ✔	✘	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.	✔
✔	(selected) ✘	Global replacement may give highly variable per process completion time because number of page faults become un-predictable.	✘
(selected) ✔	✘	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 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
Global replacement may give highly variable per process completion time because number of page faults become un-predictable.: Consistent
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

Question **11**

Partially correct

Mark 0.60 out of 1.00

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

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

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

Question **12**

Incorrect

Mark 0.00 out of 1.00

Select all the correct statements about signals

Select one or more:

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

Your answer is incorrect.

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 **13**

Incorrect

Mark 0.00 out of 1.00

Select the correct statements about interrupt handling in xv6 code

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

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 **14**

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

Your answer is correct.

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

Question **15**

Partially correct

Mark 0.10 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 changed only by interrupt handlers
- [ ] b. The scheduler can change state of a process from RUNNALBE to RUNNING
- [x] c. A process becomes ZOMBIE when it calls exit() ✔
- [ ] d. Process state is stored in the processor
- [ ] e. A process becomes ZOMBIE when another process bites into it's memory
- [x] f. The scheduler can change state of a process from RUNNALBE to RUNNING and vice-versa ✖
- [ ] g. Process state can be implemented as just a number
- [x] h. Process state is stored in the PCB ✔
- [x] i. Process state is implemented as a string ✖

Your answer is partially correct.

You have correctly selected 2.
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 **16**

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

Your answer is correct.

The correct answer is: end, P2V(PHYSTOP)

Question **17**

Incorrect

Mark 0.00 out of 1.00

If one thread opens a file with read privileges then

Select one:

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

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() ✔
Create page table entries for a given range of virtual and physical addresses; including page directory entries if needed	mappages() ✔
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() ✔
Array listing the kernel memory mappings, to be used by setupkvm()	kmap[] ✔
return a free page, if available; 0, otherwise	kalloc() ✔

Your answer is correct.

The correct answer is: Setup kernel part of a page table, and switch to that page table → kvmalloc(), Create page table entries for a given range of virtual and physical addresses; including page directory entries if needed → mappages(), 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(), Array listing the kernel memory mappings, to be used by setupkvm() → kmap[], return a free page, if available; 0, otherwise → kalloc()

Question **19**

Partially correct

Mark 0.63 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. :
Yes ✔

h. Increase the page size. : Yes ✔

Question **20**

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 **21**

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 **22**

Partially correct

Mark 0.38 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	Statement	
○ ✗	◉ ✓	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.	✗
○ ✓	◉ ✗	String arguments are NOT copied in kernel memory, but just pointed to by a kernel memory pointer	✗
○ ✗	◉ ✓	The arguments to system call are copied to kernel stack in trapasm.S	✓
○ ✓	◉ ✗	The arguments are accessed in the kernel code using esp on the trapframe.	✗
◉ ✓	○ ✗	Integer arguments are copied from user memory to kernel memory using argint()	✓
○ ✓	◉ ✗	The functions like argint(), argstr() make the system call arguments available in the kernel.	✗
◉ ✗	○ ✓	Integer arguments are stored in eax, ebx, ecx, etc. registers	✗

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

Question **23**

Partially correct

Mark 0.60 out of 1.00

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

Select one or more:

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

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 24
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	✔
○ ✘	◉ ✔	Virtual addresses become available to executing process	✔
◉ ✔	○ ✘	Code need not be completely in memory	✔
○ ✘	◉ ✔	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	✔
◉ ✔	○ ✘	One Program's size can be larger than physical memory size	✔

Cumulative size of all programs can be larger than physical memory size: True
Virtual addresses become available to executing process: False
Code need not be completely in memory: True
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
One Program's size can be larger than physical memory size: True

Question 25
Partially correct
Mark 0.50 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 ✔
- [ ] b. POSIX allows giving name to shared memory, System V does not
- [ ] c. System V is more prevalent than POSIX even today
- [x] d. POSIX shared memory is "thread safe", System V is 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 **26**

Partially correct

Mark 0.50 out of 1.00

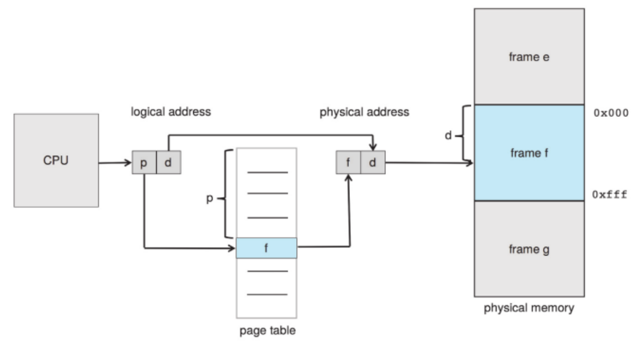


Figure 9.8 Paging hardware.

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

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

Question 27

Partially correct

Mark 0.60 out of 1.00

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

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

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

Question **28**

Partially correct

Mark 0.20 out of 2.00

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

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

Your answer is partially correct.

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

Grade details: 2 / 20 = 10%

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. 1 / 1 = 100%
9. 0 / 1 = 0%
10. 1 / 1 = 100%
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%
20. 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 **29**

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 **30**

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

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 stack allocated in entry.S is used as stack for scheduler's context for first processor: True
The kernel's page table given by kpgdir variable is used as stack for scheduler's context: False
The free page-frame are created out of nearly 222 MB: True
The switchkvm() call in scheduler() changes CR3 to use page directory of new process: False
The switchkvm() call in scheduler() is invoked after control comes to it from sched(), thus demanding execution in kernel's context: True
The switchkvm() call in scheduler() is invoked after control comes to it from swtch() scheduler(), thus demanding execution in new process's context: False
PHYSTOP can be increased to some extent, simply by editing memlayout.h: True
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

Question **31**

Partially correct

Mark 0.60 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!

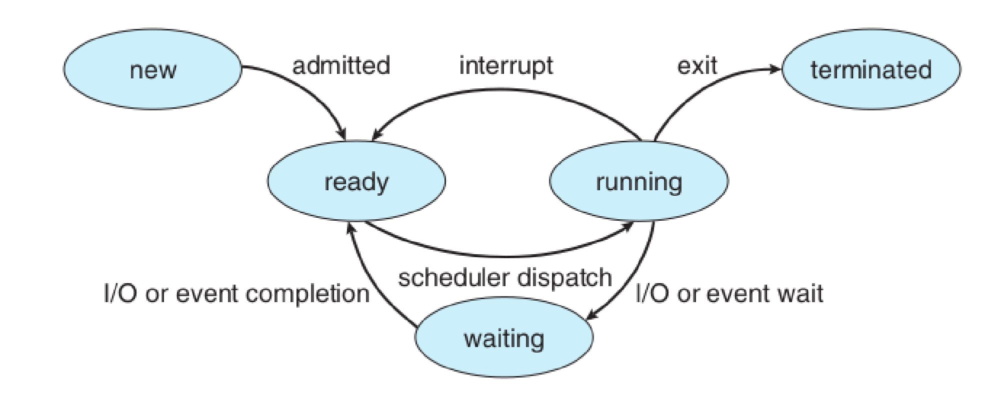


Figure 3.2 Diagram of process state.

True	False		
◉ ✔	○ ✘	Only a process in READY state is considered by scheduler	✔
◉ ✘	○ ✔	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	✔
◉ ✔	○ ✘	Every forked process has to go through ZOMBIE state, at least for a small duration.	✔
◉ ✘	○ ✔	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.	✘

Only a process in READY state is considered by scheduler: 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
Every forked process has to go through ZOMBIE state, at least for a small duration.: 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 **32**

Partially correct

Mark 0.89 out of 2.00

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

Copy the code pages of a process	copyuvm()	✖
Setup and load the user page table for initcode process	setupkvm()	✖
Load contents from ELF into existing pages	loaduvm()	✔
Create a copy of the page table of a process	clearpteu()	✖
Switch to kernel page table	switchuvm()	✖
Mark the page as in-accessible	clearpteu()	✔
Switch to user page table	switchuvm()	✔
setup the kernel part in the page table	setupkvm()	✔
Load contents from ELF into pages after allocating the pages first	loaduvm()	✖

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

Question **33**

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

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 **34**

Partially correct

Mark 0.67 out of 1.00

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

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

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

Question **35**

Partially correct

Mark 0.10 out of 1.00

Select all the correct statements about linking and loading.

Select one or more:

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

Your answer is partially correct.

You have correctly selected 3.
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.

◄ (Optional Assignment) lseek system call in xv6

Jump to...

Feedback on Quiz-2 ►

Started on	Monday, 7 March 2022, 7:00:46 PM
State	Finished
Completed on	Monday, 7 March 2022, 8:20:00 PM
Time taken	1 hour 19 mins
Grade	**11.42** out of 15.00 (**76**%)

Question **1**

Complete

Mark 1.00 out of 1.00

Map the virtual address to physical address in xv6

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

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

Question **2**

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

Question **3**

Complete

Mark 1.00 out of 1.00

What does seginit() do?

- a. Nothing significant, just repetition of earlier GDT setup but with kernel page table allocated now
- b. Adds two additional entries to GDT corresponding to Code and Data segments, but to be used in privilege level 0
- c. Nothing significant, just repetition of earlier GDT setup but with 2-level paging setup done
- 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 free frames list created 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 **4**

Complete

Mark 1.50 out of 1.50

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

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

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

Question **5**

Complete

Mark 1.00 out of 1.00

Why is there a call to kinit2? Why is it not merged with kinit1?

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

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

Question **6**

Complete

Mark 0.67 out of 1.00

Which of the following is done by mappages()?

- [ ] a. create page table mappings to the range given by "pa" and "pa + size"
- [x] b. allocate page table if required
- [ ] c. allocate page frame if required
- [ ] d. allocate page directory if required
- [x] e. create page table mappings for the range given by "va" and "va + 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 **7**

Complete

Mark 0.08 out of 1.00

Select all the correct statements about initcode

- [x] a. code of initcode is loaded in memory by the kernel during userinit()
- [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. initcode essentially calls exec("/init",...)
- [x] e. the size of 'initcode' is 2c
- [x] f. code of initcode is loaded at virtual address 0
- [ ] g. initcode is the 'init' process

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 **8**

Complete

Mark 1.00 out of 1.00

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

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

The correct answer is: 801154a8

Question **9**

Complete

Mark 1.00 out of 1.00

What does userinit() do ?

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

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

Question **10**

Complete

Mark 1.50 out of 1.50

Which of the following is DONE by allocproc() ?

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

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 **11**

Complete

Mark 0.67 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

- [ ] a. elf.entry is anyways 0, so both statements mean the same
- [ ] b. the initcode is created using objcopy, which discards all relocation information and symbols (like entry)
- [ ] c. the 'entry' in initcode is anyways 0
- [x] d. exec() loads from ELF file and the address of first instruction to be executed is given by 'entry'
- [x] e. the code of 'initcode' is loaded at physical address 0
- [x] f. In userinit() the function inituvm() has mapped the code of 'initcode' to be starting at virtual 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 **12**

Complete

Mark 0.00 out of 1.00

The approximate number of page frames created by kinit1 is

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

The correct answer is: 3000

Question **13**

Complete

Mark 1.00 out of 1.00

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

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

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

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

Jump to...

(Optional Assignment) Slab allocator in xv6 ►

Dashboard / My courses / Computer Engineering & IT / CEIT-Even-sem-21-22 / OS-even-sem-21-22 / 21 February - 27 February / Topic-wise Quiz-4 (Virtual Memory)

Started on	Saturday, 26 February 2022, 5:20:40 PM
State	Finished
Completed on	Saturday, 26 February 2022, 6:35:00 PM
Time taken	1 hour 14 mins
Grade	**9.05** out of 15.00 (**60**%)

Question **1**

Complete

Mark 0.75 out of 1.00

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

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

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 **2**

Complete

Mark 0.67 out of 1.00

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

Select one or more:

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

The correct answers are: paging, segmentation, demand paging

Question **3**

Complete

Mark 0.29 out of 1.00

Compare paging with demand paging and select the correct statements.

Select one or more:

- [x] a. With demand paging, it's possible to have user programs bigger than physical memory.
- [ ] b. Both demand paging and paging support shared memory pages.
- [x] c. Demand paging requires additional hardware support, compared to paging.
- [ ] d. Paging requires NO hardware support in CPU
- [ ] e. The meaning of valid-invalid bit in page table is different in paging and demand-paging.
- [ ] f. Paging requires some hardware support in CPU
- [ ] g. TLB hit ration has zero impact in effective memory access time in demand paging.
- [ ] h. With paging, it's possible to have user programs bigger than physical memory.
- [ ] i. Calculations of number of bits for page number and offset are same in paging and demand paging.
- [ ] 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 **4**

Complete

Mark 0.00 out of 1.00

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

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

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 **5**

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

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

Question **6**

Complete

Mark 1.00 out of 1.00

Page sizes are a power of 2 because

Select one:

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

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

Question **7**

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

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

Question **8**
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
static loading	wastage of physical memory
dynamic linking	small executable file

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

Question **9**

Complete

Mark 0.36 out of 0.50

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

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

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

Question **10**

Complete

Mark 0.75 out of 1.00

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

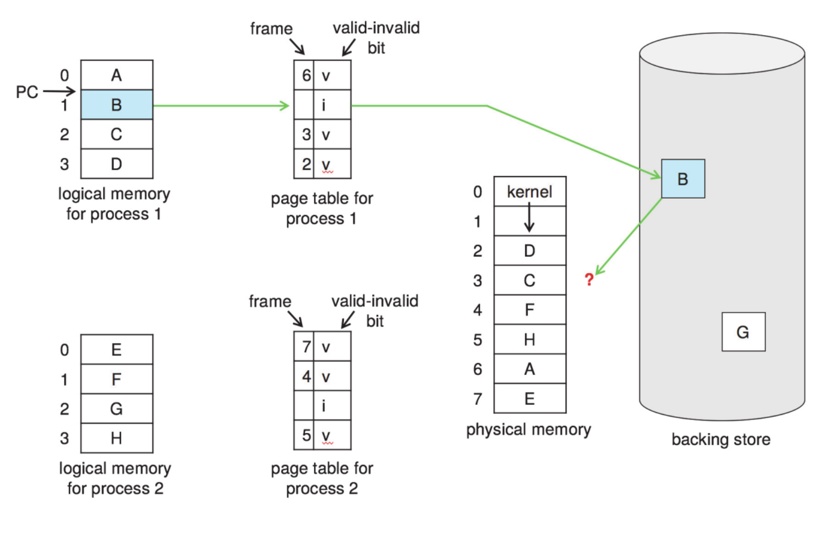


Figure 10.9 Need for page replacement.

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

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

Question **11**

Complete

Mark 0.75 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
```

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

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 **12**

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 = 105 ns

Average page fault service time = 13 ms

Page fault rate = 0.6

Answer: 7.80

The correct answer is: 7800042.00

Question **13**

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

Question **14**

Complete

Mark 1.00 out of 1.00

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

(give answer also in decimal)

Answer: 107

The correct answer is: 107

Question **15**

Complete

Mark 0.75 out of 1.00

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

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. Page faulting process is made to wait in a queue
6. Other processes scheduled by scheduler
7. Empty frame is found
8. Disk read is issued
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

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 **16**

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)?

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

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

◄ (Code) mmap related programs

Jump to...

Points from Mid-term feedback ►

Operating Systems Even-sem-21-22

Topic-wise Quiz-3 (processes, trap handling, scheduler)

Topic-wise Quiz-3 (processes, trap handling, scheduler)

Attempts allowed: 1

This quiz closed on Monday, 21 February 2022, 8:00:00 PM

To continue with this quiz attempt you must open your webcam, and it will be used to monitor you during the quiz.

Time limit: 55 mins

Summary of your previous attempts

State	Grade / 10.00	Review
Finished Submitted Monday, 21 February 2022, 7:59:27 PM	8.90	Review

Your final grade for this quiz is 8.90/10.00.

No more attempts are allowed

Back to the course

◄ Description of some possible course mini projects

Jump to...

(Code) mmap related programs ►

Dashboard / My courses / Computer Engineering & IT / CEIT-Even-sem-21-22 / OS-even-sem-21-22 / 7 February - 13 February / Quiz-1: 10 AM

Started on	Saturday, 12 February 2022, 10:01:48 AM
State	Finished
Completed on	Saturday, 12 February 2022, 11:56:09 AM
Time taken	1 hour 54 mins
Grade	**5.82** out of 10.00 (**58**%)

Question **1**

Complete

Mark 0.75 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
- [x] b. On any interrupt/syscall/exception the control first jumps in vectors.S
- [ ] c. On any interrupt/syscall/exception the control first jumps in trapasm.S
- [x] d. Each entry in IDT essentially gives the values of CS and EIP to be used in handling that interrupt
- [ ] e. xv6 uses the 0x64th entry in IDT for system calls
- [x] f. The CS and EIP are changed only after pushing user code's SS,ESP on stack
- [x] g. The trapframe pointer in struct proc, points to a location on kernel stack
- [x] h. All the 256 entries in the IDT are filled
- [x] i. xv6 uses the 64th entry in IDT for system calls
- [ ] j. The CS and EIP are changed only immediately on a hardware interrupt
- [ ] k. The function trap() is the called only in case of hardware interrupt
- [ ] l. Before going to alltraps, the kernel stack contains upto 5 entries.
- [ ] m. The function trap() is the called irrespective of hardware interrupt/system-call/exception

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 **2**

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
  system call returns
  P1 running
  timer interrupt
  Scheduler running
  P2 running

- [x] b. 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

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

- [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

- [x] f. P1 running

P1 makes system call
timer interrupt
Scheduler
P2 running
timer interrupt
Scheuler
P1 running
P1's system call return

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 **3**
Complete
Mark 0.00 out of 0.50

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

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

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 **4**

Complete

Mark 0.13 out of 0.50

Select all the correct statements about zombie processes

Select one or more:

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

Question **5**

Complete

Mark 0.50 out of 0.50

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

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

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

Question 6

Complete

Mark 0.50 out of 0.50

Suppose a program does a scanf() call.

Essentially the scanf does a read() system call.

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

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

Select one:

- a. read() will return and process will be taken to a wait queue
- b. OS code for read() will move the PCB of this process to a wait queue and return from the system call
- c. OS code for read() will move PCB of current process to a wait queue and call scheduler
- d. OS code for read() will call scheduler
- 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 7

Complete

Mark 0.50 out of 0.50

Consider the following programs

exec1.c

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

exec2.c

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

Compiled as

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

And run as

$ ./exec1

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

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

Select one:

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

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

Question **8**

Complete

Mark 0.25 out of 0.50

The bootmain() function has this code

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

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

In these statements 0x1000 is referred to as ADDRESS

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

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
It the value of ADDRESS is changed to a higher number (upto a limit), the program could still work: True
It the value of ADDRESS is changed to a lower number (upto a limit), the program could still work: True
The value ADDRESS is changed to a 0 the program could still work: False
This line loads the kernel code at ADDRESS: False

Question **9**

Complete

Mark 0.80 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()`
(•)	( )	The variable c->scheduler on first processor uses the stack allocated entry.S
(•)	( )	`sched()` and `scheduler()` are co-routines
(•)	( )	the control returns to `switchkvm();` after `swtch(&(c->scheduler), p->context);` in `scheduler()`
(•)	( )	When a process is scheduled for execution, it resumes execution in `sched()` after the call to `swtch()`
(•)	( )	`swtch` is a function that saves old context, loads new context, and returns to last EIP in the new context
( )	(•)	`sched()` *calls* `scheduler()` and `scheduler()` *calls* `sched()`
( )	(•)	`swtch` is a function that does not return to the caller
(•)	( )	The function `scheduler()` executes using the kernel-only stack
( )	(•)	the control returns to `mycpu()->intena = intena; ();` after `swtch(&p->context, mycpu()->scheduler);` in `sched()`

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

Question **10**

Complete

Mark 0.36 out of 0.50

Order the events that occur on a timer interrupt:

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

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

Question **11**

Complete

Mark 0.20 out of 0.50

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

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

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

Question **12**

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

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 **13**

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

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

Question **14**

Complete

Mark 0.43 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 8010000c
- [x] b. The kernel file gets loaded at the Physical address 0x10000 in memory.
- [ ] c. The elf->entry is set by the linker in the kernel file and it's 0x80000000
- [x] d. The readseg finally invokes the disk I/O code using assembly instructions
- [x] e. The kernel file in memory is not necessarily a continuously filled in chunk, it may have holes in it.
- [x] f. The kernel ELF file contains actual physical address where particular sections of 'kernel' file should be loaded
- [x] g. The stosb() is used here, to fill in some space in memory with zeroes
- [ ] h. The kernel file gets loaded at the Physical address 0x10000 +0x80000000 in memory.
- [ ] i. The elf->entry is set by the linker in the kernel file and it's 0x80000000
- [ ] j. The kernel file has only two program headers
- [ ] k. The condition   if(ph->memsz > ph->filesz) is never true.

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 **15**

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	
○	◉	PID of Init
○	◉	Pointer to IDT
◉	○	EIP at the time of context switch
○	◉	Pointer to the parent process
◉	○	PID
◉	○	Process state
◉	○	Function pointers to all system calls
◉	○	Process context
◉	○	List of opened files
◉	○	Memory management information about that process

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

Question **16**

Not answered

Marked 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.

◄ 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	Wednesday, 9 February 2022, 7:01:59 PM
State	Finished
Completed on	Wednesday, 9 February 2022, 8:01:50 PM
Time taken	59 mins 51 secs
Grade	**10.00** out of 11.00 (**91**%)

Question **1**

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 new value of EIP
- b. change the CS and EIP to 32 bit mode, and jumps to next line of code
- 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 **2**

Complete

Mark 1.00 out of 1.00

The kernel is loaded at Physical Address

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

The correct answer is: 0x00100000

Question **3**

Complete

Mark 1.00 out of 1.00

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

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

The correct answer is: 3

Question **4**

Complete

Mark 1.00 out of 1.00

The variable $stack in entry.S is

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

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

Question **5**

Complete

Mark 1.00 out of 1.00

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

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

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

Question **6**

Complete

Mark 1.00 out of 1.00

The kernel ELF file contains how many Program headers?

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

The correct answer is: 3

Question **7**

Complete

Mark 1.00 out of 1.00

ELF Magic number is

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

The correct answer is: 0x464C457FU

Question **8**

Complete

Mark 1.00 out of 1.00

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

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

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

Question **9**

Complete

Mark 1.00 out of 1.00

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

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

The correct answer is: Because it needs to setup paging

Question **10**

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 **11**

Not answered

Marked out of 0.50

Match the pairs of which action is taken by whom

Answer:

The correct answer is: kernel

Question **12**

Complete

Mark 1.00 out of 1.00

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

- a. Get the "entry" in ELF structure and convert it into a function pointer accepting no arguments and returning nothing
- b. Get the "entry" in ELF structure and convert it into a function void pointer
- c. Convert the "entry" in ELF structure into void
- 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, 24 January 2022, 7:02:26 PM
State	Finished
Completed on	Monday, 24 January 2022, 8:57:38 PM
Time taken	1 hour 55 mins
Grade	**10.40** out of 20.00 (**52**%)

Question **1**

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 **2**

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. Because the memory is volatile
- c. Because the variables of the program are stored in memory
- d. Becase the processor can run instructions and access data only from memory

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

Question **3**

Complete

Mark 2.00 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() { 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() { execl("/usr/bin/echo", "/usr/bin/echo", "hi\n", NULL); }	hi
main() { int i = NULL; fork(); printf("hi\n"); }	hi hi

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

Question **4**

Complete

Mark 1.33 out of 2.00

Which of the following instructions should be privileged?

Select one or more:

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

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 **5**

Complete

Mark 0.00 out of 0.50

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

- a. yes
- b. no

The correct answer is: yes

Question **6**

Complete

Mark 1.00 out of 1.00

Match the register with the segment used with it.

esp	ss
ebp	ss
eip	cs
esi	ds
edi	es

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

Question **7**

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

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 **8**

Complete

Mark 0.50 out of 0.50

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

- ( ) a. yes
- (•) b. no

The correct answer is: no

Question **9**

Complete

Mark 1.00 out of 2.00

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

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

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

Question **10**

Complete

Mark 0.00 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 0.40 out of 1.00

Select all the correct statements about two modes of CPU operation

Select one or more:

- [ ] a. The two modes are essential for a multiprogramming system
- [ ] b. Some instructions are allowed to run only in user mode, while all instructions can run in kernel mode
- [x] c. The two modes are essential for a multitasking system
- [x] d. The software interrupt instructions change the mode from user mode to kernel mode and jumps to predefined location simultaneously
- [ ] 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 1.00 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.

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

Question **13**

Complete

Mark 0.83 out of 2.00

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

Select one or more:

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

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 **14**

Complete

Mark 0.33 out of 0.50

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

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

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

Question **15**

Complete

Mark 0.00 out of 2.00

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

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

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 0.00 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 prohibits invocation of kernel code completely, if a user program 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 disallows hardware interrupts when a process is running

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

◄ (Task) Compulsory xv6 task

Jump to...

(Optional Assignment) Shell Programming(Conformance tests) ►

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:

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

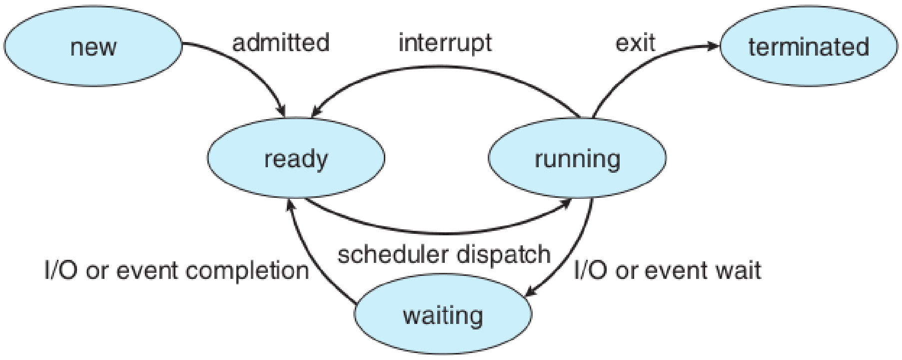


Figure 3.2 Diagram of process state.

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:

- [x] 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
- [x] 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
- [x] g. MMU is inside the processor ✔
- [x] 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		
(selected) ✗	✓	Integer arguments are stored in eax, ebx, ecx, etc. registers	✗
(selected) ✓	✗	String arguments are NOT copied in kernel memory, but just pointed to by a kernel memory pointer	✓
(selected) ✓	✗	The functions like argint(), argstr() make the system call arguments available in the kernel.	✓
(selected) ✗	✓	String arguments are first copied to trapframe and then from trapframe to kernel's other variables.	✗
(selected) ✓	✗	The arguments to system call originally reside on process stack.	✓
(selected) ✗	✓	The arguments to system call are copied to kernel stack in trapasm.S	✗
(selected) ✓	✗	Integer arguments are copied from user memory to kernel memory using argint()	✓
(selected) ✓	✗	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

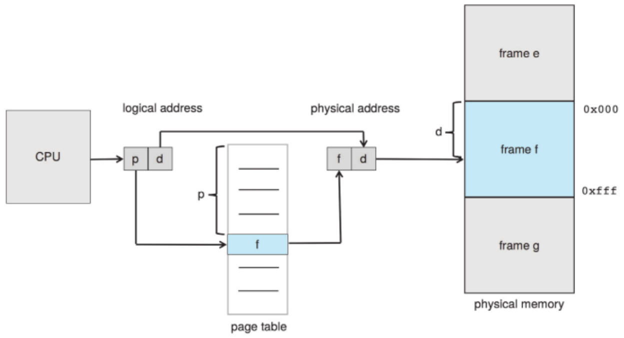


Figure 9.8 Paging hardware.

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.

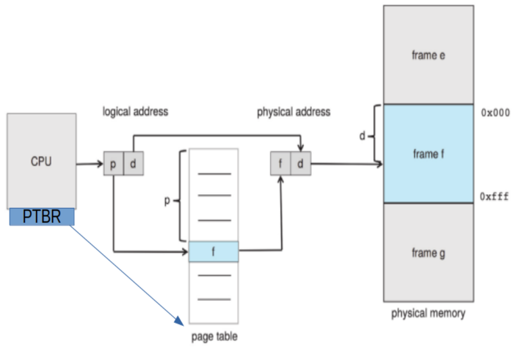


Figure 9.8 Paging hardware.

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

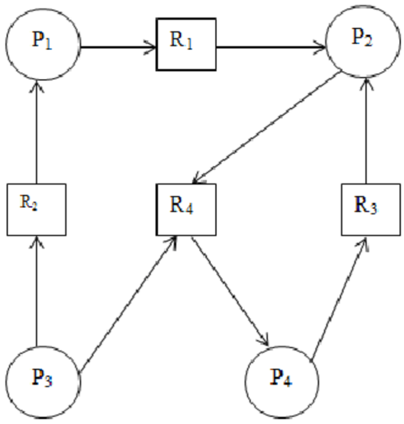
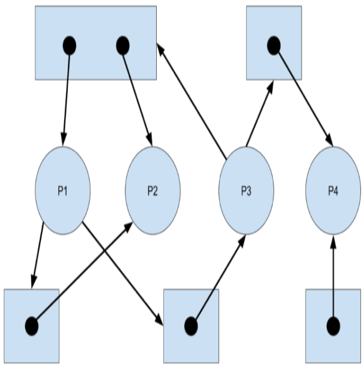
Question **12**

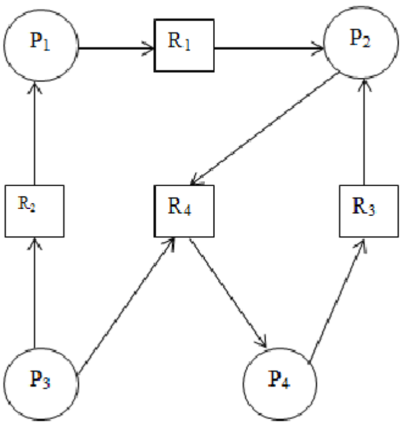
Incorrect

Mark 0.00 out of 1.00

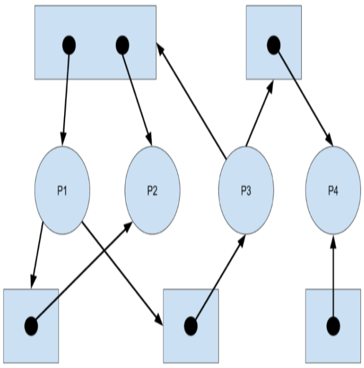
For each of the resource allocation diagram shown,

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.

	inode	rec_len	name_len	file_type	name							
0	21	12	1	2	.	\0	\0	\0				
12	22	12	2	2	.	.	\0	\0				
24	53	16	5	2	h	o	m	e	1	\0	\0	\0
40	67	28	3	2	u	s	r	\0				
52	0	16	7	1	o	l	d	f	i	l	e	\0
68	34	12	4	2	s	b	i	n				

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)

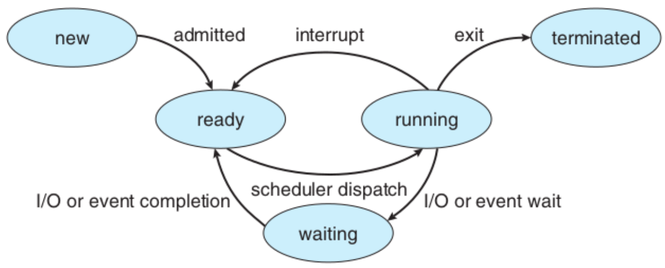


Figure 3.2 Diagram of process state.

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?

- a. The xv6.img is the virtual disk that is created by combining the bootblock and the kernel file.
- 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)
- 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 ►

Dashboard / My courses / Computer Engineering & IT / CEIT-Even-sem-21-22 / OS-even-sem-21-22 / 21 February - 27 February / Topic-wise Quiz-4 (Virtual Memory)

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.

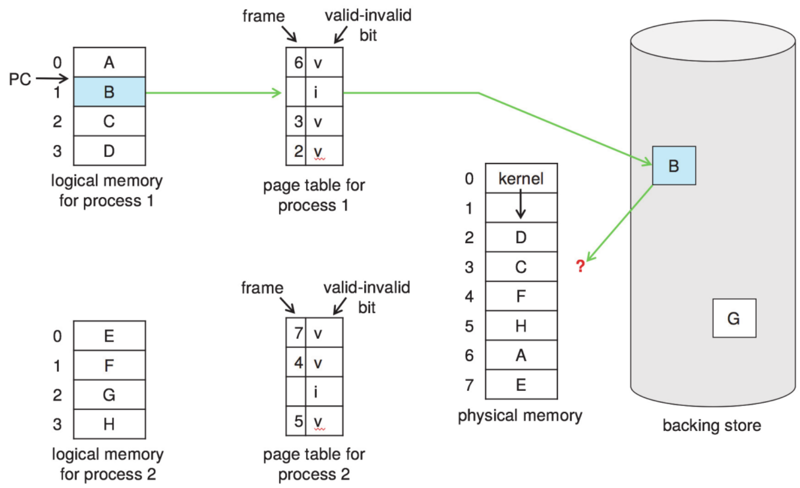


Figure 10.9 Need for page replacement.

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.
- [x] b. Vfork() assumes that there will be no write, but rather exec()
- [x] 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()
- [x] 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 ►

Dashboard / My courses / Computer Engineering & IT / CEIT-Even-sem-21-22 / OS-even-sem-21-22 / 7 March - 13 March / Topic-wise Quiz-5 (xv6 memory management, userinit, exec)

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 ►

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

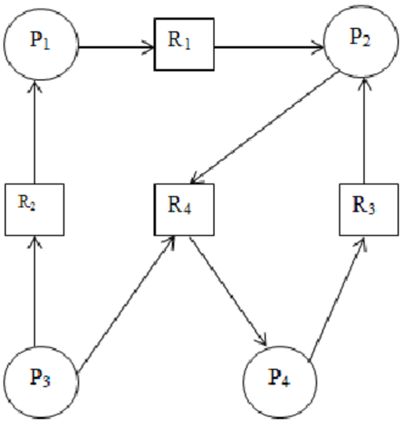
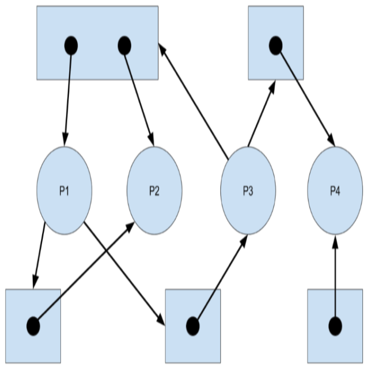
Question **12**

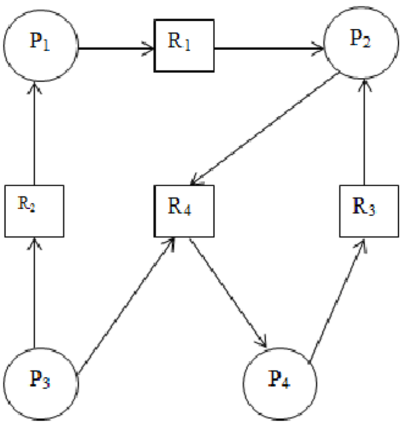
Incorrect

Mark 0.00 out of 1.00

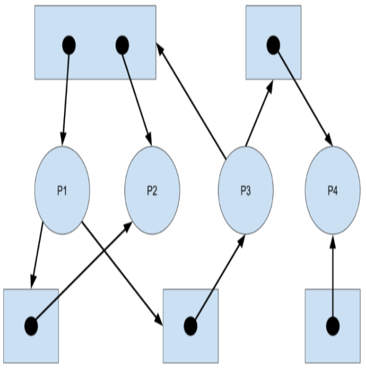
For each of the resource allocation diagram shown,

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.

	inode	rec_len	name_len	file_type	name							
0	21	12	1	2	.	\0	\0	\0				
12	22	12	2	2	.	.	\0	\0				
24	53	16	5	2	h	o	m	e	1	\0	\0	\0
40	67	28	3	2	u	s	r	\0				
52	0	16	7	1	o	l	d	f	i	l	e	\0
68	34	12	4	2	s	b	i	n				

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)

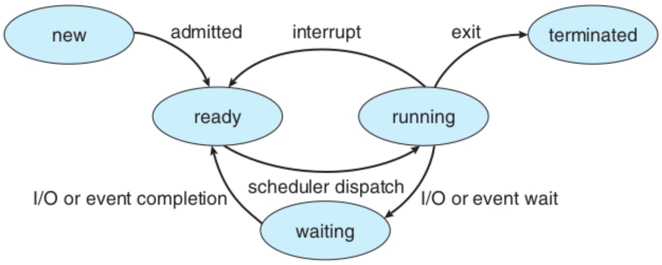


Figure 3.2 Diagram of process state.

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	Saturday, 30 April 2022, 2:05:41 PM
State	Finished
Completed on	Saturday, 30 April 2022, 6:47:56 PM
Time taken	4 hours 42 mins
Grade	**34.99** out of 40.00 (**87**%)

Question **1**
Correct
Mark 1.00 out of 1.00

which of the following scheduling algorithms discriminate either in favor of or against short processes, from the perspective of minimizing waiting time.

For	Against		
○ ✗	◉ ☑	FCFS	✔
◉ ☑	○ ✗	Round Robin	✔
◉ ☑	○ ✗	Multilevel feedback queues	✔

FCFS: Against
Round Robin: For
Multilevel feedback queues: For

Question **2**
Correct
Mark 1.00 out of 1.00

Consider a computer system with a 32-bit logical address and 8- KB page size. The system supports up to 1 GB of physical memory. How many entries are there in each of the following?

a. A conventional, single-level page table (write a decimal number):

524288

✔

b. An inverted page table (number of entries only, write a decimal number):

131072

✔

Question **3**

Correct

Mark 1.00 out of 1.00

Select T/F for the disk block allocation scheme related statements

True	False		
◉ ✔	○ ✖	Continuous allocation leads to faster file access	✔
◉ ✔	○ ✖	FAT uses linked allocation	✔
◉ ✔	○ ✖	The unix inode is based on indexed allocation	✔
◉ ✔	○ ✖	Continuous allocation allows to fetch any block with just one seek	✔
◉ ✔	○ ✖	NVM storage devices are pushing for search for new block allocation schemes	✔
◉ ✔	○ ✖	Continuous allocation may involve a costly search for free space	✔
◉ ✔	○ ✖	Maximum file size limit is determined by disk block allocation scheme, as one of the factors.	✔
◉ ✔	○ ✖	Linked allocation does away with file size limitation to a large extent	✔

Continuous allocation leads to faster file access: True
FAT uses linked allocation: True
The unix inode is based on indexed allocation: True
Continuous allocation allows to fetch any block with just one seek: True
NVM storage devices are pushing for search for new block allocation schemes: True
Continuous allocation may involve a costly search for free space: True
Maximum file size limit is determined by disk block allocation scheme, as one of the factors.: True
Linked allocation does away with file size limitation to a large extent: True

Question **4**

Correct

Mark 1.00 out of 1.00

Which one of the following is not a system call

- ◉ a. lock ✔
- ○ b. open
- ○ c. lseek
- ○ d. mount

The correct answer is: lock

Question **5**

Partially correct

Mark 1.13 out of 1.50

The following processes are being scheduled using a pre-emptive, priority-based, round-robin scheduling algorithm.

Process	Priority	Burst	Arrival
P_1	8	15	0
P_2	3	20	0
P_3	4	20	20
P_4	4	20	25
P_5	5	5	45
P_6	5	15	55

Each process is assigned a numerical priority, with a higher number indicating a higher relative priority. The scheduler will execute the currently highest priority process for it's full duration, unless it gets pre-empted by newly arriving higher priority process. For processes with the same priority, a round-robin scheduler will be used with a time quantum of 10 units. If a process is pre-empted by a higher-priority process, the pre-empted process is placed at the front of the queue of same priority processes (so that it's turn continues when higher priority process is over).

The order in which the processes get scheduled is (write your answer without a space, e.g. P1,P2,P3,P4,P5) :

P1,P2,P3,P4,P3,P5,P3,P6,P4,P2

✖

The turn-around time for the process P2 is:

95

✔

The turn-around time for the process P6 is:

15

✔

The process P5 finishes at time unit:

50

✔

0-15 P1

15-20 P2

20-40 P3

40-45 P4

45-50 P5

50-55 P4

55-70 P6

70-80 P4

80-95 P2

--

P2 turnaround time = 95 - 15 = 80

Question **6**

Partially correct

Mark 1.62 out of 3.00

Suppose it is required to add the chown() (without notion of a group, just the notion of owner and others) system call to xv6. Select the changes, from the options given below, which are an absolute must to effect the addition of this system call.

Changes w.r.t. other system calls should be selected if those system calls are necessary for the implementation of the chown() system call.

Must	Not-Must		
◉ ✗	○ ✓	Add a EUID field in the struct proc representing Effective user ID	✗
◉ ✓	○ ✗	Add a mode field representing file permissions, inside dinode	✓
◉ ✗	○ ✓	Add a UID field in the struct proc represenging the USER-ID of the user who started this process	✗
◉ ✓	○ ✗	Add a uid field representing the owner, inside dinode	✓
○ ✗	◉ ✓	Add few extra files belonging to different users, using mkfs.c	✓
◉ ✗	○ ✓	Modify exec() to check SUID bit on the executable file and set EUID of the process to UID of the executable	✗
◉ ✓	○ ✗	Add the chown() system call which checks if the user with id "0" is calling this system call and then change the ownership of the on-disk and in-memory inode.	✓
◉ ✗	○ ✓	Add a setuid(), seteuid() system call, callable only by the user with ID or EUID equal to "0" to set the new user id of the process	✗
◉ ✗	○ ✓	Create an application program su.c which itself is a SUID program, and calls setuid() and setuid() with specified user-ID and then does exec() of a shell.	✗
◉ ✓	○ ✗	Modify mkfs.c to add owner and permissions to each file created	✓
◉ ✗	○ ✓	Inherit the UID and EUID of the parent in fork()	✗
○ ✗	◉ ✓	Mandatorily create a file like "passwd" which maps user-names to user-IDs and modify other utilities (like "ls") to show user-name instead of user-ID for the files.	✓
○ ✗	◉ ✓	Add the username field to the on disk inode, that is dinode.	✓

Add a EUID field in the struct proc representing Effective user ID: Not-Must
Add a mode field representing file permissions, inside dinode: Must
Add a UID field in the struct proc represenging the USER-ID of the user who started this process: Not-Must
Add a uid field representing the owner, inside dinode: Must
Add few extra files belonging to different users, using mkfs.c: Not-Must
Modify exec() to check SUID bit on the executable file and set EUID of the process to UID of the executable: Not-Must
Add the chown() system call which checks if the user with id "0" is calling this system call and then change the ownership of the on-disk and in-memory inode.: Must
Add a setuid(), seteuid() system call, callable only by the user with ID or EUID equal to "0" to set the new user id of the process: Not-Must
Create an application program su.c which itself is a SUID program, and calls setuid() and setuid() with specified user-ID and then does exec() of a shell.: Not-Must

Modify mkfs.c to add owner and permissions to each file created: Must
Inherit the UID and EUID of the parent in fork(): Not-Must
Mandatorily create a file like "passwd" which maps user-names to user-IDs and modify other utilities (like "ls") to show user-name instead of user-ID for the files.: Not-Must
Add the username field to the on disk inode, that is dinode.: Not-Must

Question **7**
Correct
Mark 1.00 out of 1.00

The following diagram that explains the concept of multi-threading

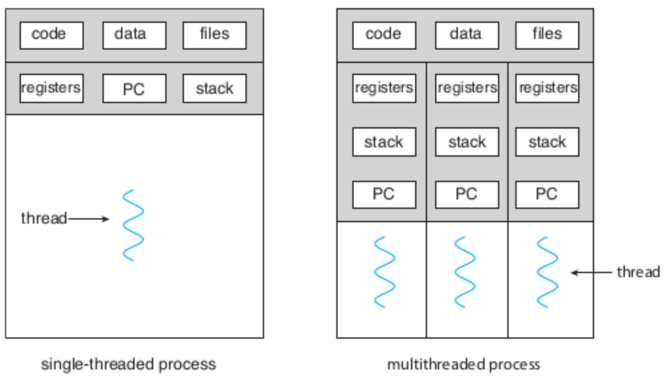


Figure 4.1 Single-threaded and multithreaded processes.

Leads to the following conclusions about implementation of threads

True	False		
○ ✗	◉ ✓	A multi-threaded program can be split in multiple ELF files.	✓
◉ ✓	○ ✗	The memory management for the process in the kernel should not (most typically) change due to creation of a thread.	✓
◉ ✓	○ ✗	Switching between threads requires a "context switch" with change of registers involved in it	✓
◉ ✓	○ ✗	Each thread should be represented by a function that starts execution on a separate stack.	✓

A multi-threaded program can be split in multiple ELF files.: False
The memory management for the process in the kernel should not (most typically) change due to creation of a thread.: True
Switching between threads requires a "context switch" with change of registers involved in it: True
Each thread should be represented by a function that starts execution on a separate stack.: True

Question **8**

Correct

Mark 1.00 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 = c1; //A
        c1++; //B
    }
}
void *thread2(void *arg) {
    while(run == 1) {//F
        c = c2;//C
        c2++;//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		
○ ✗	◉ ✓	E	✓
○ ✗	◉ ✓	F	✓
○ ✗	◉ ✓	B	✓
○ ✗	◉ ✓	D	✓
◉ ✓	○ ✗	A	✓
◉ ✓	○ ✗	C	✓

As per Remzi's book: "A critical section is a piece of code that accesses a shared variable (or more generally, a shared resource and must not be concurrently executed by more than one thread."

As per Galvin's Book: a critical section, in which the process may be accessing — and updating — data that is shared with at least one other process. The important feature of the system is that when one process is executing in its critical section, no other process is allowed to execute in its critical section.

Since A and C refer to a variable that is shared, it's critical section. Here the hardware guarantees (as = is atomic) that the critical section is accessed by only one thread at a time.

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

Question **9**

Partially correct

Mark 0.38 out of 1.00

It is proposed that when a process does an illegal memory access, xv6 terminate the process by printing the error message "Illegal Memory Access". Select all the changes that need to be done to xv6 for this as True (Note that the changes proposed here may not cover the exhaustive list of all changes required) and the un-necessary/wrong changes as False.

Required	Un-necessary/Wrong		
○ ✔	◉ ✘	Change in the Makefile and instruct cc/ld to start the code of each program at some address other than 0	✘
○ ✔	◉ ✘	Ensure that the address 0 is mapped to invalid	✘
○ ✘	◉ ✔	Mark each page as readonly in the page table mappings	✔
○ ✔	◉ ✘	Change allocuvm() to call mappages() with proper permissions on each page table entry	✘
◉ ✘	○ ✔	Add code that checks if the illegal memory access trap was due to an actual illegal memory access.	✘
◉ ✔	○ ✘	Handle the Illegal memory access trap in trap() function, and terminate the currently running process.	✔
◉ ✔	○ ✘	Change exec to treat text/data sections separately and call allocuvm() with proper flags for page table entries	✔
◉ ✘	○ ✔	Change mappages() to set specified permissions on each page table entry	✘

Change in the Makefile and instruct cc/ld to start the code of each program at some address other than 0: Required
Ensure that the address 0 is mapped to invalid: Required
Mark each page as readonly in the page table mappings: Un-necessary/Wrong
Change allocuvm() to call mappages() with proper permissions on each page table entry: Required
Add code that checks if the illegal memory access trap was due to an actual illegal memory access.: Un-necessary/Wrong
Handle the Illegal memory access trap in trap() function, and terminate the currently running process.: Required
Change exec to treat text/data sections separately and call allocuvm() with proper flags for page table entries: Required
Change mappages() to set specified permissions on each page table entry: Un-necessary/Wrong

Question **10**

Partially correct

Mark 0.63 out of 1.00

Select which of the following data structures may need an update, in ext2 file system, when 5 bytes get removed from the end of an existing file

Yes	No		
○ ✗	◉ ✓	The inode of the parent directory	✓
○ ✓	◉ ✗	The superblock	✗
◉ ✓	○ ✗	Inode of the file	✓
○ ✗	◉ ✓	The inode bitmap	✓
○ ✓	◉ ✗	The group descriptor	✗
◉ ✗	○ ✓	A free block on the file-system	✗
◉ ✓	○ ✗	One of the block bitmaps	✓
○ ✗	◉ ✓	The boot sector	✓

The inode of the parent directory: No
The superblock: Yes
Inode of the file: Yes
The inode bitmap: No
The group descriptor: Yes
A free block on the file-system: No
One of the block bitmaps: Yes
The boot sector: No

Question **11**

Partially correct

Mark 1.75 out of 2.00

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

"..." means some code.

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

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

```
void
panic(char *s)
{
...
  panicked = 1;
```

Ensure that no printing happens on other processors ✔

```
struct proc*
myproc(void) {
...
  pushcli();
  c = mycpu();
  p = c->proc;
  popcli();
...
}
```

Disable interrupts to avoid another process's pointer being returned ✔

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

Tell compiler not to reorder memory access beyond this line ✔

```
void
acquire(struct spinlock *lk)
{
...
  getcallerpcs(&lk, lk->pcs);
```

Traverse ebp chain to get sequence of instructions followed in functions calls ✔

```
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
yield(void)
{
...
release(&ptable.lock);
}
```

Release the lock held by some another process ✔

```
void
acquire(struct spinlock *lk)
{
  pushcli();
```

Disable interrupts to avoid deadlocks ✔

Your answer is partially correct.

You have correctly selected 7.

The correct answer is: void
sleep(void *chan, struct spinlock *lk)
{
  ...
  if(lk != &ptable.lock){
    acquire(&ptable.lock);
    release(lk);
  } → Avoid a self-deadlock, void
panic(char *s)
{
...
  panicked = 1; → Ensure that no printing happens on other processors, struct proc*
myproc(void) {
...
  pushcli();
  c = mycpu();
  p = c->proc;
  popcli();
...
}

  → Disable interrupts to avoid another process's pointer being returned, void
acquire(struct spinlock *lk)
{
...
__sync_synchronize();

  → Tell compiler not to reorder memory access beyond this line, void
acquire(struct spinlock *lk)
{
...
  getcallerpcs(&lk, lk->pcs);

  → Traverse ebp chain to get sequence of instructions followed in functions calls, 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

```
yield(void)
{
...
release(&ptable.lock);
}
```

→ Release the lock held by some another process, void

```
acquire(struct spinlock *lk)
{
  pushcli();
```

→ Disable interrupts to avoid deadlocks

Question **12**

Correct

Mark 1.00 out of 1.00

Mark statements as T/F

All statements are in the context of preventing deadlocks.

True	False		
◉ ✓	○ ✗	Mutual exclusion is a necessary condition for deadlock because it brings in locks on which deadlock happens	✔
○ ✗	◉ ✓	The lock ordering to be followed to avoid circular wait is a code in OS that checks for compliance with decided order	✔
◉ ✓	○ ✗	The lock ordering to be followed to avoid circular wait is a protocol to be followed by programmers.	✔
◉ ✓	○ ✗	Circular wait is avoided by enforcing a lock ordering	✔
◉ ✓	○ ✗	If a resource allocation graph contains a cycle then there is a possibility of a deadlock	✔
○ ✗	◉ ✓	If a resource allocation graph contains a cycle then there is a guarantee of a deadlock	✔
◉ ✓	○ ✗	Deadlock is not possible if any of these conditions is not met: 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: True
The lock ordering to be followed to avoid circular wait is a code in OS that checks for compliance with decided order: False
The lock ordering to be followed to avoid circular wait is a protocol to be followed by programmers.: True
Circular wait is avoided by enforcing a lock ordering: True
If a resource allocation graph contains a cycle then there is a possibility of a deadlock: True
If a resource allocation graph contains a cycle then there is a guarantee of a deadlock: False
Deadlock is not possible if any of these conditions is not met: Mutual exclusion, hold and wait, no pre-emption, circular wait.: True

Question **13**

Correct

Mark 1.00 out of 1.00

Predict the output of the program given here.

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`

`--`

```
int main() {
  int pid;
  printf("hi\n");
  pid = fork();
  if(pid == 0) {
    exit(0);
  }
  printf("bye\n");
  fork();
  printf("ok\n");
}
```

Answer: hi bye ok ok

The correct answer is: hi bye ok ok

Question **14**

Partially correct

Mark 0.63 out of 1.00

For Virtual File System to work, which of the following changes are required to be done to an existing OS code (e.g. xv6)?

- [ ] a. A mount() system call should be provided to mount a partition onto some directory in existing namespace rooted at "/"
- [x] b. The filesystem related system calls (e.g. read, write) need to invoke the file system specific functions (e.g. ext2_read, ext2_write, ntfs_read, ntfs_write) using function pointers. ✔
- [x] c. The file system specific function pointers, for file system system-calls, need to be setup in the generic inode during lookup. ✔
- [ ] d. The generic inode needs to have a field representing if this inode is a mount point and also to refer/point to the root of the mounted file system's inode.
- [x] e. Each open() needs to copy the function pointers from the inode of the parent directory into the inode of the child (if not already done), unless it's traversing a mount point. (This may be done as part of lookup() which is called by open()) ✔
- [x] f. Each file-system writer needs to provide the set of function pointers for VFS, and these function pointers need to be setup in generic inode of "/" of that file system during mount() ✔
- [ ] g. The operating system in-memory inode needs to be a generic-inode representing "inode" like data structure across multiple file systems.
- [x] h. The lookup() operation needs to check if it's crossing a mount point and call FS specific operations to read inodes/directories ✔

The correct answers are: A mount() system call should be provided to mount a partition onto some directory in existing namespace rooted at "/", The filesystem related system calls (e.g. read, write) need to invoke the file system specific functions (e.g. ext2_read, ext2_write, ntfs_read, ntfs_write) using function pointers., The file system specific function pointers, for file system system-calls, need to be setup in the generic inode during lookup., The operating system in-memory inode needs to be a generic-inode representing "inode" like data structure across multiple file systems., The generic inode needs to have a field representing if this inode is a mount point and also to refer/point to the root of the mounted file system's inode., The lookup() operation needs to check if it's crossing a mount point and call FS specific operations to read inodes/directories, Each file-system writer needs to provide the set of function pointers for VFS, and these function pointers need to be setup in generic inode of "/" of that file system during mount(), Each open() needs to copy the function pointers from the inode of the parent directory into the inode of the child (if not already done), unless it's traversing a mount point. (This may be done as part of lookup() which is called by open())

Question **15**

Partially correct

Mark 0.40 out of 0.50

Which of the following statements are correct about xv6 and multiprocessor support ?

- [ ] a. Each processor on xv6 starts code execution in mpenter() when first processor sets "started" to 1.
- [x] b. xv6 supports only SMP ✔
- [x] c. In xv6 on x86, the first processor configures other processors in mpinit() ✔
- [x] d. xv6 supports a variable number of processors (upto 8) and adjusts it's data structures according to number of processors ✔
- [x] e. At any point in time, after main(), the kernel may be parallely executing on any of the processors. ✔

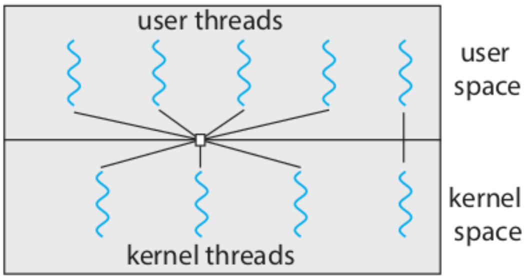
The correct answers are: xv6 supports only SMP, xv6 supports a variable number of processors (upto 8) and adjusts it's data structures according to number of processors, Each processor on xv6 starts code execution in mpenter() when first processor sets "started" to 1., In xv6 on x86, the first processor configures other processors in mpinit(), At any point in time, after main(), the kernel may be parallely executing on any of the processors.

Question **16**

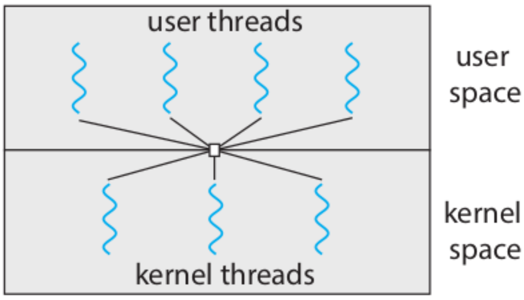
Correct

Mark 1.00 out of 1.00

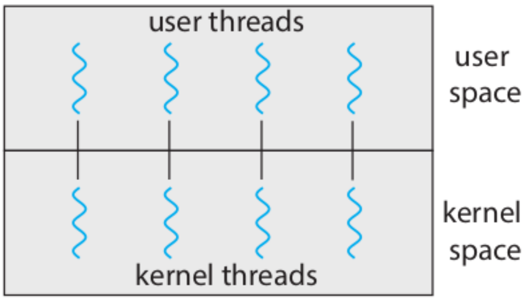
Match the diagram with the threading model



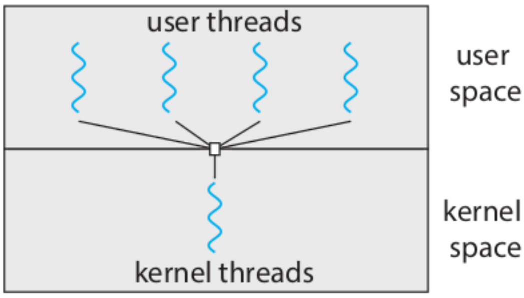
Two-Level ✔



Many-Many ✔

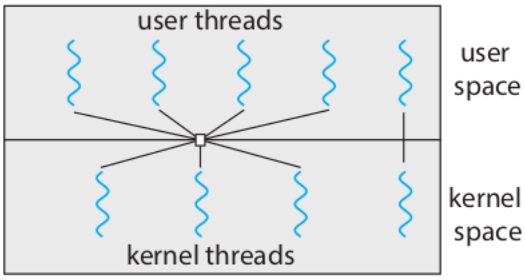


One-One ✔

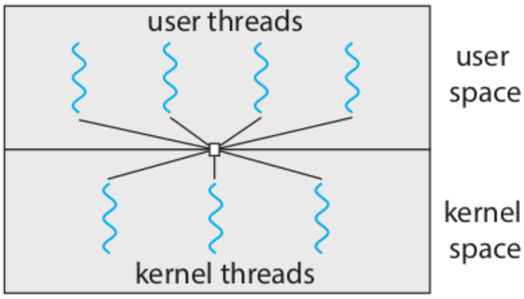


Many-One ✔

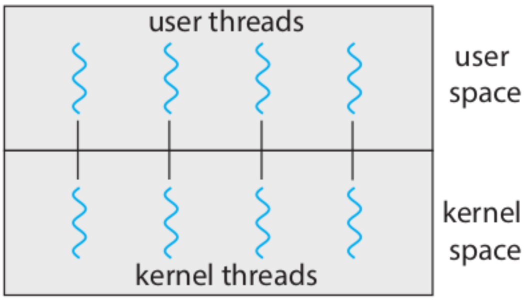
Your answer is correct.

The correct answer is: 

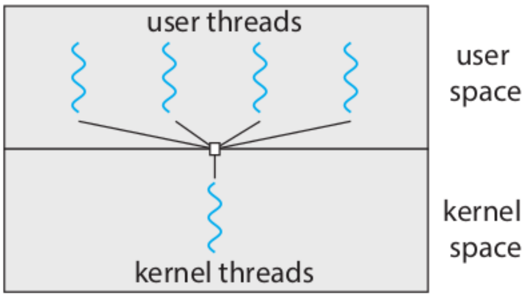
→ Two-Level,



→ Many-Many,



→ One-One,



→ Many-One

Question **17**

Correct

Mark 1.00 out of 1.00

Select all the correct statements about Shell

- [x] a. The essential job of the shell is to fork-exec the specified application ✔
- [ ] b. Examples of shell are: bash, ksh, csh, msh, ooosh, nosh, etc.
- [ ] c. Shell is a layer on top of hardware
- [ ] d. Shell converts the application code into system calls.
- [x] e. The default shell for a user is specified in /etc/passwd on typical GNU/Linux systems. ✔
- [x] f. Examples of shell are: bash, ksh, csh, zsh, sh, etc. ✔

The correct answers are: The essential job of the shell is to fork-exec the specified application, Examples of shell are: bash, ksh, csh, zsh, sh, etc., The default shell for a user is specified in /etc/passwd on typical GNU/Linux systems.

Question **18**

Correct

Mark 1.00 out of 1.00

Which one of the following is not a scheduling algorithm

- a. Multilevel Feedback Queue
- b. Priority
- c. FCFS
- d. paging

The correct answer is: paging

Question **19**

Correct

Mark 1.00 out of 1.00

Mark the statements as True/False w.r.t. the basic concepts of memory management.

True	False		
○ ✗	◉ ✓	The compiler interacts with the kernel continuously while compiling a program and obtains the correct set of memory addresses for code/stack/heap/data and then generates the machine code file.	✓
○ ✗	◉ ✓	The compiler generates the address references for code/data/stack/heap in the executable file as per the memory management schema chosen by the compiler itself, and then the kernel ensures that program is executed with this schema.	✓
◉ ✓	○ ✗	When a process is executing, each virtual address is converted into physical address by the CPU hardware directly.	✓
○ ✗	◉ ✓	When a process is executing, each virtual address is converted into physical address by the kernel directly.	✓
◉ ✓	○ ✗	The compiler generates address references for code/data/stack/heap in the executable file, depending on the MM architecture provided by CPU and kernel.	✓
○ ✗	◉ ✓	The kernel refers to the page table for converting each virtual address to physical address.	✓
◉ ✓	○ ✗	The kernel ensures that the MMU is setup before scheduling a process and then the CPU/MMU ensures that the address translation takes place.	✓

The compiler interacts with the kernel continuously while compiling a program and obtains the correct set of memory addresses for code/stack/heap/data and then generates the machine code file.: False
The compiler generates the address references for code/data/stack/heap in the executable file as per the memory management schema chosen by the compiler itself, and then the kernel ensures that program is executed with this schema.: False
When a process is executing, each virtual address is converted into physical address by the CPU hardware directly.: True
When a process is executing, each virtual address is converted into physical address by the kernel directly.: False
The compiler generates address references for code/data/stack/heap in the executable file, depending on the MM architecture provided by CPU and kernel.: True
The kernel refers to the page table for converting each virtual address to physical address.: False
The kernel ensures that the MMU is setup before scheduling a process and then the CPU/MMU ensures that the address translation takes place.: True

Question **20**

Partially correct

Mark 0.83 out of 1.00

Which of the following instructions should be privileged?

- [x] a. Modify entries in device-status table. ✔
- [ ] b. Read the clock.
- [x] c. Switch from user to kernel mode. ✔
- [x] d. Set value of timer. ✔
- [x] e. Turn off interrupts. ✔
- [x] f. Access I/O device. ✔
- [ ] g. Issue a trap instruction.

The correct answers are: Set value of timer., Access I/O device., Issue a trap instruction., Switch from user to kernel mode., Modify entries in device-status table., Turn off interrupts.

Question **21**

Correct

Mark 1.00 out of 1.00

Calculate the average waiting time using
FCFS scheduling
for the following workload

assuming that they arrive in this order during the first time unit:

Process	Burst Time
P1	2
P2	6
P3	2
P4	3

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

Answer: 5.00 ✔

P2 waits for 2 units

P3 waits for 2+6 units

P4 waits for 2 + 6 +2 units of time

Total waiting = 2 + 2 + 6 + 2 + 6 + 2 = 20 units

Average waiting time = 20/4 = 5

The correct answer is: 5

Question **22**

Partially correct

Mark 0.25 out of 0.50

Select all the correct statements related to implementation of a Hypervisor like VirtualBox

- [ ] a. All Traps in hardware are handled by HOST OS, but Host OS may hand it over to Guest OS if trap was from within guest OS
- [ ] b. Typically they run using CPU privilege level 1 or 2 (lower than kernel's but higher than applications)
- [x] c. When an application runs a privileged instruction it traps into the actual hardware ✔
- [x] d. The HOST OS determines whether a process (in Guest) or the guest OS itself was doing privileged instruction depending on the privilege level. ✔

The correct answers are: Typically they run using CPU privilege level 1 or 2 (lower than kernel's but higher than applications), When an application runs a privileged instruction it traps into the actual hardware, All Traps in hardware are handled by HOST OS, but Host OS may hand it over to Guest OS if trap was from within guest OS, The HOST OS determines whether a process (in Guest) or the guest OS itself was doing privileged instruction depending on the privilege level.

Question **23**

Partially correct

Mark 0.75 out of 1.00

Match each suggested semaphore implementation (discussed in class)

with the problems that it faces

```
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) {
                spinunlock(&(s->sl));
                spinlock(&(s->sl));
        }
        (s->val)--;
        spinunlock(&(s->sl));
}
```

too much spinning, bounded wait not guaranteed ✔

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

blocks holding a spinlock ✔

```
struct semaphore {
        int val;
        spinlock lk;
        list l;
};
sem_init(semaphore *s, int initval) {
        s->val = initval;
        s->sl = 0;
}
block(semaphore *s) {
        listappend(s->l, current);
        spinunlock(&(s->sl));
        schedule();
}
wait(semaphore *s) {
        spinlock(&(s->sl));
        while(s->val <=0) {
                block(s);
        }
        (s->val)--;
        spinunlock(&(s->sl));
}
signal(seamphore *s) {
        spinlock(*(s->sl));
        (s->val)++;
        x = dequeue(s->sl) and enqueue(readyq, x);
        spinunlock(*(s->sl));
}
```

not holding lock after unblock ✔

```
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));
}
```

livelock ✖

Your answer is partially correct.

You have correctly selected 3.

The correct answer is:

```
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) {
                spinunlock(&(s->sl));
                spinlock(&(s->sl));
        }
        (s->val)--;
        spinunlock(&(s->sl));
}
```

→ too much spinning, bounded wait not guaranteed,

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

→ blocks holding a spinlock,

```
struct semaphore {
        int val;
        spinlock lk;
        list l;
};
sem_init(semaphore *s, int initval) {
        s->val = initval;
        s->sl = 0;
}
block(semaphore *s) {
        listappend(s->l, current);
        spinunlock(&(s->sl));
        schedule();
}
wait(semaphore *s) {
        spinlock(&(s->sl));
        while(s->val <=0) {
                block(s);
        }
        (s->val)--;
        spinunlock(&(s->sl));
}
signal(seamphore *s) {
        spinlock(*(s->sl));
        (s->val)++;
        x = dequeue(s->sl) and enqueue(readyq, x);
        spinunlock(*(s->sl));
}
```

→ not holding lock after unblock,

```
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));
}
```

→ deadlock

Question **24**

Correct

Mark 1.00 out of 1.00

In an ext2 file system, if the block size is 4KB and partition size is 2 GB, then the number of block groups will be:

Answer: 16 ✔

size * 1024 * 1024 / 4 --> no of blocks

each group = 8 * 4 * 1024 blocks = 32768 blocks

so size * 1024 * 1024 / (4 * 32768) number of groups

The correct answer is: 16.00

Question **25**

Correct

Mark 1.00 out of 1.00

Match the pair

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

Your answer is correct.

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

Question **26**

Correct

Mark 1.00 out of 1.00

Write the possible contents of the file /tmp/xyz after this program.

In the answer if you want to mention any non-text character, then write \0 For example abc\0\0 means abc followed by any two non-text characters

```
int main(int argc, char *argv[]) {
   int fd1, fd2, n, i;
   char buf[128];

   fd1 = open("/tmp/xyz", O_WRONLY | O_CREAT, S_IRUSR|S_IWUSR);
   write(fd1, "hello", 5);
   fd2 = open("/tmp/xyz", O_WRONLY, S_IRUSR|S_IWUSR);
   write(fd2, "bye", 3);
   close(fd1);
   close(fd2);
   return 0;
}
```

Answer: byelo

The correct answer is: byelo

Question **27**

Correct

Mark 1.00 out of 1.00

Select the correct statements about hard and soft links

Select one or more:

- [ ] a. Soft links increase the link count of the actual file inode
- [x] b. Hard links enforce separation of filename from it's metadata in on-disk data structures. ✔
- [ ] c. Hard links can span across partitions while soft links can't
- [ ] d. Deleting a soft link deletes only the actual file
- [x] e. Soft links can span across partitions while hard links can't ✔
- [x] f. Deleting a soft link deletes the link, not the actual file ✔
- [ ] g. Soft link shares the inode of actual file
- [x] h. Deleting a hard link deletes the file, only if link count was 1 ✔
- [ ] i. Deleting a soft link deletes both the link and the actual file
- [ ] j. Deleting a hard link always deletes the file
- [x] k. Hard links share the inode ✔
- [x] l. Hard links increase the link count of the actual file inode ✔

Your answer is correct.

The correct answers are: Soft links can span across partitions while hard links can't, Hard links increase the link count of the actual file inode, Deleting a soft link deletes the link, not the actual file, Deleting a hard link deletes the file, only if link count was 1, Hard links share the inode, Hard links enforce separation of filename from it's metadata in on-disk data structures.

Question **28**

Partially correct

Mark 2.54 out of 3.00

Suppose you are required to implement the priority scheduling algorithm in xv6. Select all the options that correctly reflect the changes that are required to be done in code:

Needed/Optional	Not Needed		
◉ ✔	○ ✘	Set the priority of the new process to the default(using inheritance) during fork()	✔
◉ ✔	○ ✘	Add a system call (like nice()) that allows to set priority of a process	✔
○ ✘	◉ ✔	Change sched() to set the priority of the process.	✔
◉ ✔	○ ✘	Add a priority field to the struct proc	✔
○ ✘	◉ ✔	Change scheduler() to calculate the priority of the process using user specified value	✔
◉ ✔	○ ✘	The nice() system call needs to acquire the ptable.lock for setting the priority.	✔
○ ✘	◉ ✔	Change swtch() to set the timer value for the process	✔
◉ ✘	○ ✔	Change yield() to re-set the timer value to zero for the process	✘
◉ ✔	○ ✘	Optionally remove the default timer value of 10000000 in lapicinit()	✔
◉ ✘	○ ✔	Change exec() to add a priority to the process.	✘
○ ✘	◉ ✔	Set the priority of the process as per information in ELF file	✔
◉ ✔	○ ✘	Insert code in scheduler() after if(p->state != RUNNABLE); continue; by a code that selects the highest priority process.	✔
◉ ✔	○ ✘	Set a new value for the timer before you call swtch() in scheduler()	✔

Set the priority of the new process to the default(using inheritance) during fork(): Needed/Optional
Add a system call (like nice()) that allows to set priority of a process: Needed/Optional
Change sched() to set the priority of the process.: Not Needed
Add a priority field to the struct proc: Needed/Optional
Change scheduler() to calculate the priority of the process using user specified value: Not Needed
The nice() system call needs to acquire the ptable.lock for setting the priority.: Needed/Optional
Change swtch() to set the timer value for the process: Not Needed
Change yield() to re-set the timer value to zero for the process: Not Needed
Optionally remove the default timer value of 10000000 in lapicinit(): Needed/Optional
Change exec() to add a priority to the process.: Not Needed
Set the priority of the process as per information in ELF file: Not Needed
Insert code in scheduler() after if(p->state != RUNNABLE); continue; by a code that selects the highest priority process.: Needed/Optional
Set a new value for the timer before you call swtch() in scheduler(): Needed/Optional

Question **29**

Correct

Mark 0.50 out of 0.50

Doing a lookup on the pathname /a/b/b/c/d for opening the file "d" requires reading 6 ✔ no. of inodes. Assume that there are no hard/soft links on the path.

Write the answer as a number.

The correct answer is: 6

Question **30**

Partially correct

Mark 1.86 out of 2.00

Given below is an incomplete code for reader-writer lock with preference for readers. Fill in the blanks to complete the code.

Note1:

In your answer, if an expression is to be written, then please separate all tokens by exactly one space in between. For example

`i = i + 1` is correct while

`i=i+1` is not correct or

`i= i +1` is also not correct.

Note2:

Correct: a->b++ (no spaces here!)

Any other notation is incorrect.

Note3: *The code of downgrade() and upgrade() has proportional to 4/7 marks.*

Code of rwlock:

typedef struct rwlock {
  int nActive;
  int nPendingReads;
  int nPendingWrites;
  spinlock_t

sl

✔ ;
  condition canRead;
  condition canWrite;
}rwlock;

void lockShared(rwlock *r) {
  spin_lock(&r->sl);

r->nPendingReads++

✔ ;
  while(r->nActive < 0)
    wait(

&r->canRead

✔ , &r->sl);

r->nActive++

✔ ;
  r->nPendingReads--;
  spin_unlock(&r->sl);
}
void unlockShared(rwlock *r) {
  spin_lock(&r->sl);

r->nActive--

✔ ;
  if(r->nActive == 0) {
    spin_unlock(&r->sl);
    do_signal(

&r->canWrite

✔);
  } else
    spin_unlock(&r->sl);
}
void lockExclusive(rwlock *r) {
  spin_lock(&r->sl);
  r->nPendingWrites++;
  while(r->nActive || r->nPendingReads)
    wait(

```
&r->canWrite
,
&r->sl
);
    r->nPendingWrites--;

r->nActive = -1
;
    spin_unlock(&r->sl);
}
void unlockExclusive(rwlock *r) {
    boolean_t wakeReaders;
    int i;
    spin_lock(&r->sl);

r->nActive = 0
;
    if(r->nPendingReads != 0) {
        for(i = 0; i <
r->nPendingReads
; i++)
            do_signal(&r->canRead);
    }
    else
        do_signal(
&r->canWrite
);
    spin_unlock(&r->sl);
}
void downgrade(rwlock *r) {
    boolean_t wakeReaders;
    int i;
    spin_lock(&r->sl);
    r->nActive =
1
;
    if(r->nPendingReads != 0)
        for(i = 0; i <
nPendingReads
; i++)
            do_signal(
&r->canRead
);
    spin_unlock(&r->sl);
}
void upgrade(rwlock *r) {
    spin_lock(&r->sl);
    if(r->nActive == 1) {
        r->nActive =
-1
;
    } else {
        r->nPendingWrites++;
        r->nActive--;
        while(r->nActive !=
0
)
            wait(
&r->canWrite
```

✔ , &r->sl);

r->nPendingWrites--

✔ ;
r->nActive =

-1

✔ ;
}
spin_unlock(&r->sl);
}

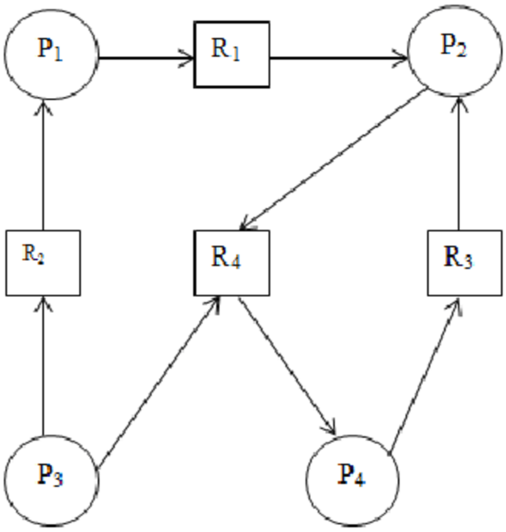
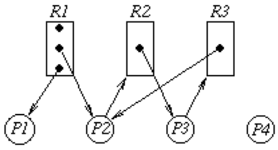
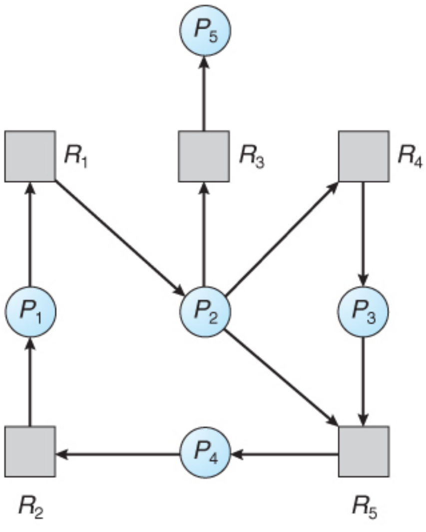
Question **31**

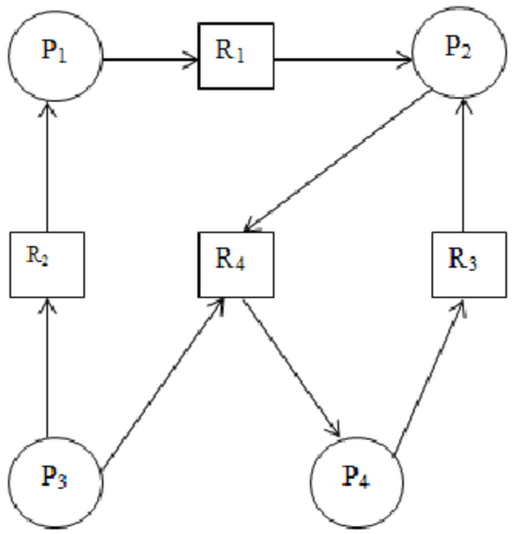
Correct

Mark 1.00 out of 1.00

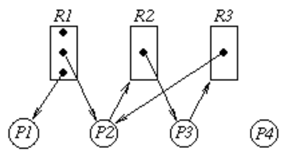
For each of the resource allocation diagram shown,

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

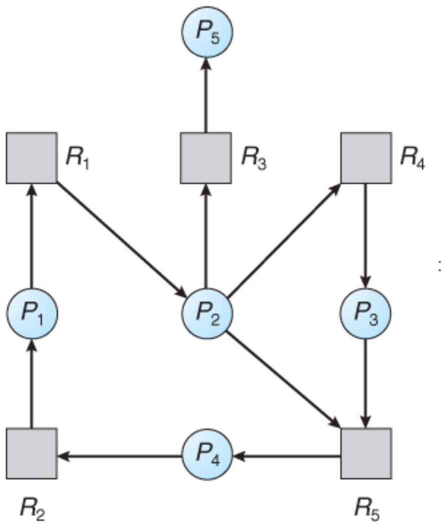
Yes	No	
☑	☐ ✗	 ✓
☑	☐ ✗	 ✓
☑	☐ ✗	 ✓



: Yes



: Yes



: Yes

Question **32**

Correct

Mark 1.00 out of 1.00

Map each signal with it's meaning

SIGPIPE	Broken Pipe
SIGUSR1	User Defined Signal
SIGSEGV	Invalid Memory Reference
SIGALRM	Timer Signal from alarm()
SIGCHLD	Child Stopped or Terminated

The correct answer is: SIGPIPE → Broken Pipe, SIGUSR1 → User Defined Signal, SIGSEGV → Invalid Memory Reference, SIGALRM → Timer Signal from alarm(), SIGCHLD → Child Stopped or Terminated

Question **33**

Correct

Mark 1.00 out of 1.00

Not a FOSS: Which of the following is/are not a FOSS operating system ?

- [ ] a. Minix
- [ ] b. Open Solaris
- [ ] c. xv6
- [ ] d. GNU/Linux
- [ ] e. Darwin (core kernel of Mac-OS)
- [ ] f. FreeBSD
- [x] g. Windows ✔
- [ ] h. BSD Unix

The correct answer is: Windows

Question **34**

Correct

Mark 1.00 out of 1.00

Will this code work for a spinlock() operation? The intention here is to call compare-and-swap() only if the lock is not held (the if condition checks for the same).

```
void spinlock(int *lock) {
{
    while (true) {
        if (*lock == 0) {
            /* lock appears to be available */
            if (!compare_and_swap(lock, 0, 1))
            break
        }
    }
}
```

- ○ a. No, because this breaks the atomicity requirement of compare-and-test.
- ○ b. Yes, because there is no race to update the lock variable
- ○ c. No, because in the case of both processes succeeding in the "if" condition, both may end up acquiring the lock.
- ◉ d. Yes, because no matter in which order the if-check and compare-and-swap run in multiple processes, only one process will succeed in compare-and-swap() and others will keep looping in while-loop. ✔

Your answer is correct.

The correct answer is: Yes, because no matter in which order the if-check and compare-and-swap run in multiple processes, only one process will succeed in compare-and-swap() and others will keep looping in while-loop.

Question **35**

Partially correct

Mark 0.75 out of 1.00

Mark the statements as True/False with respect to Mobile systems and swapping.

True	False		
◉ ✔	○ ✘	Mobile systems generally do not support swapping	✔
○ ✔	◉ ✘	Size of flash memory is one reason for restriction on use of swap on mobile devices.	✘
◉ ✔	○ ✘	Limited number of write operations that flash memory can tolerate, is one reason for limitations of swapping on mobile devices.	✔
◉ ✔	○ ✘	Poor throughput between main memory and flash memory is one reason for restriction on use of swap on mobile devices.	✔

Mobile systems generally do not support swapping: True
Size of flash memory is one reason for restriction on use of swap on mobile devices.: True
Limited number of write operations that flash memory can tolerate, is one reason for limitations of swapping on mobile devices.: True
Poor throughput between main memory and flash memory is one reason for restriction on use of swap on mobile devices.: True

◄ Course Exit Feedback

Jump to...