Started on	Tuesday, 22 March 2022, 2:00:46 PM
State	Finished
Completed on	Tuesday, 22 March 2022, 5:11:58 PM
Time taken	3 hours 11 mins
Grade	**22.83** out of 40.00 (**57**%)

Question **1**

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 **2**

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. Exhibit Balady's anomaly between 3 and 4 frames
- 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 **3**

Partially correct

Mark 0.25 out of 1.00

Select all the correct statements about signals

Select one or more:

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

Your answer is partially correct.

You have selected too many options.
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 **4**

Incorrect

Mark 0.00 out of 1.00

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

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

Your answer is incorrect.

The correct answer is: Singly linked NULL terminated list

Question **5**

Partially correct

Mark 0.17 out of 1.00

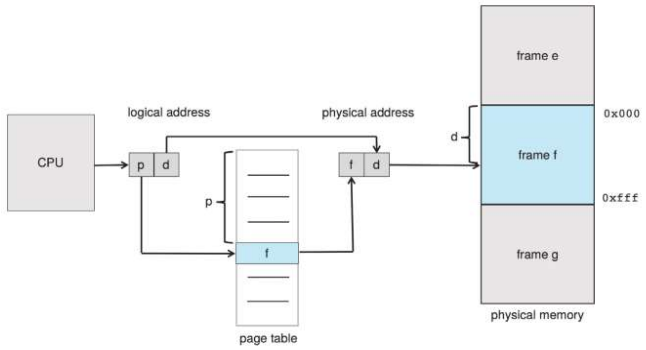

Figure 9.8 Paging hardware.

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

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

Question **6**

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.

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

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

Question **7**

Partially correct

Mark 0.56 out of 1.00

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

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

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
MAP_SHARED leads to a mapping that is copy-on-write: False
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
mmap() results in changes to buffer-cache of the kernel.: False
on failure mmap() returns NULL: False
on failure mmap() returns (void *)-1: True
mmap() is a system call: True

Question **8**

Partially correct

Mark 0.17 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 all processors	✗
bootasm.S	gdt setup with 3 entries, at start32 symbol of bootasm.S	✓
bootmain()	gdt setup with 5 entries (0 to 4) on one processor	✗
after startothers() in main()	gdt setup with 5 entries (0 to 4) on one processor	✗
entry.S	gdt setup with 3 entries, right from first line of code of bootloader	✗
kvmalloc() in main()	gdt setup with 3 entries, right from first line of code of bootloader	✗

Your answer is partially correct.

You have correctly selected 1.
The correct answer is: 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, 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

Question **9**

Partially correct

Mark 0.30 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. A process becomes ZOMBIE when another process bites into it's memory
- [ ] b. The scheduler can change state of a process from RUNNALBE to RUNNING and vice-versa
- [x] c. Process state is changed only by interrupt handlers ✖ -> no exit() also changes it, and so does fork() during it's execution
- [ ] d. A process becomes ZOMBIE when it calls exit()
- [ ] e. Process state is implemented as a string
- [ ] f. Process state is stored in the processor
- [x] g. The scheduler can change state of a process from RUNNALBE to RUNNING ✔
- [x] h. Process state is stored in the PCB ✔
- [ ] i. Process state can be implemented as just a number

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 **10**

Partially correct

Mark 0.57 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		
◉ ☑	○ ✖	Logical address space could be larger than physical address space	✔
○ ☑	◉ ✖	Cumulative size of all programs can be larger than physical memory size	✖
○ ☑	◉ ✖	One Program's size can be larger than physical memory size	✖
◉ ☑	○ ✖	Relatively less I/O may be possible during process execution	✔
○ ✖	◉ ☑	Virtual access to memory is granted to all processes	✔
◉ ☑	○ ✖	Code need not be completely in memory	✔
◉ ✖	○ ☑	Virtual addresses become available to executing process	✖

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

Question **11**

Partially correct

Mark 0.70 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 can be used for communication between only "related" processes, if the common ancestor created it.	✔
☑	☐	Named pipes can exist beyond the life-time of processes using them.	✔
☐	☑	A named pipe has a name decided by the kernel.	✖
☐	☑	Both types of pipes provide FIFO communication.	✖
☐	☑	Named pipes can be used for communication between only "related" processes.	✔
☑	☐	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".	✔
☐	☑	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
Named pipes can exist beyond the life-time of processes using them.: True
A named pipe has a name decided by the kernel.: False
Both types of pipes provide FIFO communication.: True
Named pipes can be used for communication between only "related" processes.: False
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
The pipe() system call can be used to create either a named or un-named pipe.: False

Question **12**

Partially correct

Mark 0.40 out of 1.00

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

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

Question **13**

Partially correct

Mark 1.00 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: 9

#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 **14**

Partially correct

Mark 0.43 out of 1.00

Select the correct statements about interrupt handling in xv6 code

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

Your answer is partially correct.

You have correctly selected 5.
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 **15**

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. : No ✓

c. Increase the degree of multiprogramming. : Yes ✗

d. Decrease the degree of multiprogramming. : No ✗

e. Install more main memory.: No ✗

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. : Yes ✓

Question **16**

Partially correct

Mark 0.50 out of 1.00

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

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

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 **17**

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:

- [x] a. all three models, that is many-one, one-one, many-many , require a user level thread library ✔
- [ ] b. one-one model can be implemented even if there are no kernel threads
- [x] c. A process blocks in many-one model even if a single thread makes a blocking system call ✔
- [ ] d. many-one model can be implemented even if there are no kernel threads
- [x] e. one-one model increases kernel's scheduling load ✔
- [ ] f. many-one model gives no speedup on multicore processors
- [ ] 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 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 **18**

Correct

Mark 1.00 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

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

Question **19**

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 **20**

Partially correct

Mark 1.56 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() ✔
Load contents from ELF into pages after allocating the pages first	loaduvm() ✖
Setup and load the user page table for initcode process	inituvm() ✔
Copy the code pages of a process	copyuvm() ✖
Switch to user page table	switchuvm() ✔
Switch to kernel page table	switchkvm() ✔
setup the kernel part in the page table	setupkvm() ✔
Create a copy of the page table of a process	copyuvm() ✔

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

Question **21**

Incorrect

Mark 0.00 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. Exhibit Balady's anomaly between 2 and 3 frames ✖
- c. Do not exhibit Balady's anomaly

Your answer is incorrect.

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

Question **22**

Correct

Mark 1.00 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 same process can also read from that file ✔
- c. none of these
- d. other threads in the another process can also read from that file

Your answer is correct.

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

Question **23**

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

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 **24**

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		
☐	◉	Global replacement may give highly variable per process completion time because number of page faults become un-predictable.	✗
◉	☐	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).	✓
☐	◉	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 results in more predictable per-process completion time because number of page faults can be better predicted.	✓

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
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 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
Local replacement results in more predictable per-process completion time because number of page faults can be better predicted.: Consistent

Question **25**

Partially correct

Mark 0.10 out of 1.00

Select all the correct statements about linking and loading.

Select one or more:

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

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.

Question **26**

Incorrect

Mark 0.00 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: 0 0 1 0 0 0 0 ✖

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

Question **27**

Partially correct

Mark 0.50 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	✓
◉	☐	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.	✓
☐	◉	The arguments are accessed in the kernel code using esp on the trapframe.	✗
◉	☐	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.	✓
◉	☐	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
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
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
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

Question **28**

Partially correct

Mark 0.50 out of 1.00

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

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

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 **29**

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() ✔
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[] ✔
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() ✔

Your answer is correct.

The correct answer is: Setup kernel part of a page table, and switch to that page table → kvmalloc(), 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[], 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()

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		
◉ ☑	○ ✖	The kernel code and data take up less than 2 MB space	✔
◉ ☑	○ ✖	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 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	✔
◉ ☑	○ ✖	The process's address space gets mapped on frames, obtained from ~2MB:224MB range	✔
◉ ☑	○ ✖	xv6 uses physical memory upto 224 MB only	✔
◉ ☑	○ ✖	PHYSTOP can be increased to some extent, simply by editing memlayout.h	✔
○ ✖	◉ ☑	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 switchkvm() call in scheduler() changes CR3 to use page directory of new process	✔

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

Question **31**

Partially correct

Mark 0.30 out of 2.00

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

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

Your answer is partially correct.

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

Grade details: 3 / 20 = 15%

Here are the scores for each item in this response:

1. 1 / 1 = 100%
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. 1 / 1 = 100%
18. 1 / 1 = 100%
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 **32**

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!

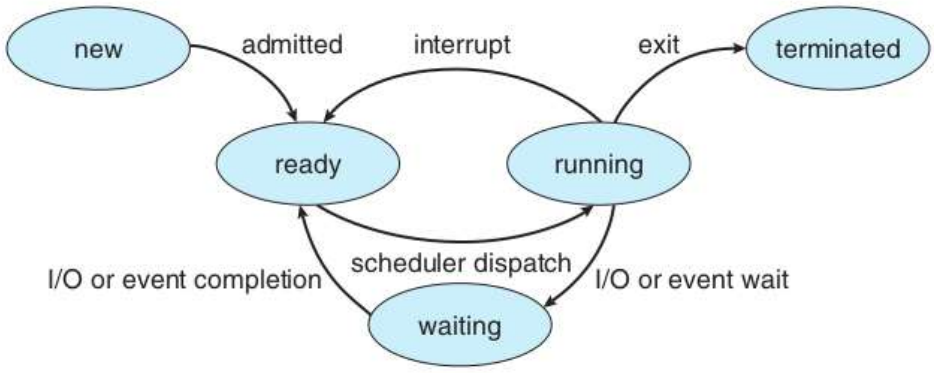

Figure 3.2 Diagram of process state.

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	✔
○☑	◉✖	Every forked process has to go through ZOMBIE state, at least for a small duration.	✖
◉☑	○✖	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.	✖

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
Every forked process has to go through ZOMBIE state, at least for a small duration.: 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 **33**

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

Your answer is correct.

The correct answer is: end, P2V(PHYSTOP)

Question **34**

Correct

Mark 1.00 out of 1.00

Select the most common causes of use of IPC by processes

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

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 **35**

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

EMBRYO	fork()->allocproc() before setting up the UVM	✔
SLEEPING	sleep(), called by any process blocking itself	✔
UNUSED	wait() called by the exiting process itself	✖
ZOMBIE	exit(), called by process itself	✔
RUNNING	scheduler()	✔
RUNNABLE	fork()->allocproc() before setting up the UVM	✖

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

◄ (Optional Assignment) lseek system call in xv6

Jump to...

Feedback on Quiz-2 ►