Started on	Friday, 17 March 2023, 2:29 PM
State	Finished
Completed on	Friday, 17 March 2023, 4:43 PM
Time taken	2 hours 14 mins
Grade	**6.25** out of 10.00 (**62.52**%)

Question **1**

Partially correct

Mark 0.19 out of 0.50

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

True	False		
◉ ✗	○ ✓	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.	✓
○ ✓	◉ ✗	'chan' is used only by the sleep() and wakeup1() functions.	✗
◉ ✗	○ ✓	Changing the state of a process automatically changes the value of 'chan'	✗
◉ ✗	○ ✓	when chan is NULL, the 'state' in proc must be RUNNABLE.	✗
○ ✓	◉ ✗	The value of 'chan' is changed only in sleep()	✗
◉ ✓	○ ✗	chan stores the address of the variable, representing a condition, for which the process is waiting.	✓

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
'chan' is used only by the sleep() and wakeup1() functions.: True
Changing the state of a process automatically changes the value of 'chan': False
when chan is NULL, the 'state' in proc must be RUNNABLE.: False
The value of 'chan' is changed only in sleep(): True
chan stores the address of the variable, representing a condition, for which the process is waiting.: True

Question **2**

Partially correct

Mark 0.66 out of 0.75

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

True	False			
◉ ✔	○ ✖	ptable.lock protects the proc[] array and all struct proc in the array	✔	
○ ✖	◉ ✔	ptable.lock can be held by different processes on different processors at the same time	✔	No lock can be held like this!
◉ ✔	○ ✖	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 is acquired but never released	✖	how is that possible?
○ ✖	◉ ✔	A process can sleep on ptable.lock if it can't aquire it.	✔	It's a spinlock!
◉ ✔	○ ✖	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
○ ✖	◉ ✔	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.
◉ ✔	○ ✖	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: True
ptable.lock can be held by different processes on different processors at the same time: False
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 is acquired but never released: False
A process can sleep on ptable.lock if it can't aquire it.: False
the rule of "never block holding a spinlock" does not apply to ptable.lock in xv6: True
The swtch() in scheduler() is called without holding the ptable.lock when control jumps to it from sched(): False
It is taken by one process but released by another process, running on same processor: True

Question **3**

Partially correct

Mark 0.32 out of 0.75

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

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

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

Question **4**

Incorrect

Mark 0.00 out of 0.25

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 **5**

Partially correct

Mark 0.33 out of 0.50

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

True	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.	✓	No. kinit2() calls kfree() and then initializes use_lock.
○ ✓	◉ ✗	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	✓	
○ ✗	◉ ✓	free page list is a singly circular linked list.	✓	it's singly linked NULL terminated list.
◉ ✓	○ ✗	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	✓	
○ ✓	◉ ✗	the kmem.lock is used by kfree() and kalloc() only.	✗	

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
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
free page list is a singly circular linked list.: False
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
the kmem.lock is used by kfree() and kalloc() only.: True

Question **6**

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.

- [ ] 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 not read as it is user programs.
- [ ] b. The bootmain() code does not read the kernel completely in memory
- [ ] c. The kernel.asm file is the final kernel file
- [x] d. The kernel.ld file contains instructions to the linker to link the kernel properly ✔
- [x] 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. readseg() reads first 4k bytes of kernel in memory ✔
- [x] 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 read using program headers in bootmain(). ✔
- [ ] i. 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.

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 **7**

Correct

Mark 0.50 out of 0.50

Which of the following is DONE by allocproc() ?

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

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**

Incorrect

Mark 0.00 out of 0.25

Which of the following call sequence is impossible in xv6?

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

Your answer is incorrect.

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

Question **9**

Partially correct

Mark 0.64 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		
◉ ✔	○ ✘	sleep() is the function which blocks a process.	✔
◉ ✘	○ ✔	the 'spinlock lk' protects 'locked' variable, but not the 'name' nor the 'pid'	✘
◉ ✔	○ ✘	Sleeplock() will ensure that either the process gets the lock or the process gets blocked.	✔
◉ ✔	○ ✘	The spinlock lk->lk is held when the process comes out of sleep()	✔
◉ ✔	○ ✘	the 'spinlock lk' is needed in a sleeplock, because access to the sleeplock for locking/unlocking itself creates a critical section	✔
○ ✘	◉ ✔	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()	✔

True	False			
◉ ✗	○ ✓	Wakeup() will wakeup the first process waiting for the lock	✗	Wakeup() will wakeup all processes waiting for the lock
○ ✓	◉ ✗	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()
◉ ✗	○ ✓	A process has acquired the sleeplock when it comes out of sleep()	✗	
◉ ✓	○ ✗	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.
○ ✗	◉ ✓	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().

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

```
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
```

Question **10**

Partially correct

Mark 0.50 out of 1.00

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

True	False			
◉ ✗	○ ✓	swtch() called from scheduler() changes the stack from the process's kernel stack to the scheduler's kernel stack.	✗	it does reverse!
◉ ✗	○ ✓	movl %esp, (%eax) means, *(c->scheduler) = contents of esp When swtch() is called from scheduler()	✗	No. it means c->scheduler = contents of esp.
◉ ✓	○ ✗	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.
◉ ✗	○ ✓	switch stores the old context on new stack, and restores new context from old stack.	✗	old goes on old, new comes from new stack
○ ✓	◉ ✗	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.
○ ✗	◉ ✓	sched() is the only place when p->context is set	✓	no. allocproc() also sets it.
◉ ✓	○ ✗	swtch() changes the context from "old" to "new"	✓	yeah, that's the definition
◉ ✓	○ ✗	push in swtch() happens on old stack, while pop happens from new stack	✓	
◉ ✓	○ ✗	swtch() is called only from sched() or scheduler()	✓	

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

Question **11**

Correct

Mark 0.25 out of 0.25

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

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

The correct answer is: 801154a8

Question **12**

Correct

Mark 0.50 out of 0.50

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

- a. struct buf is used for disk I/o which takes lot of time, so sleeping/blocking is the only option available.
- b. It could be a spinlock, but xv6 has chosen sleeplock for purpose of demonstrating how to use a sleeplock.
- c. 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
- d. sleeplock is preferable because it is used in interrupt context and spinlock can not be used in interrupt context
- 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 correct.

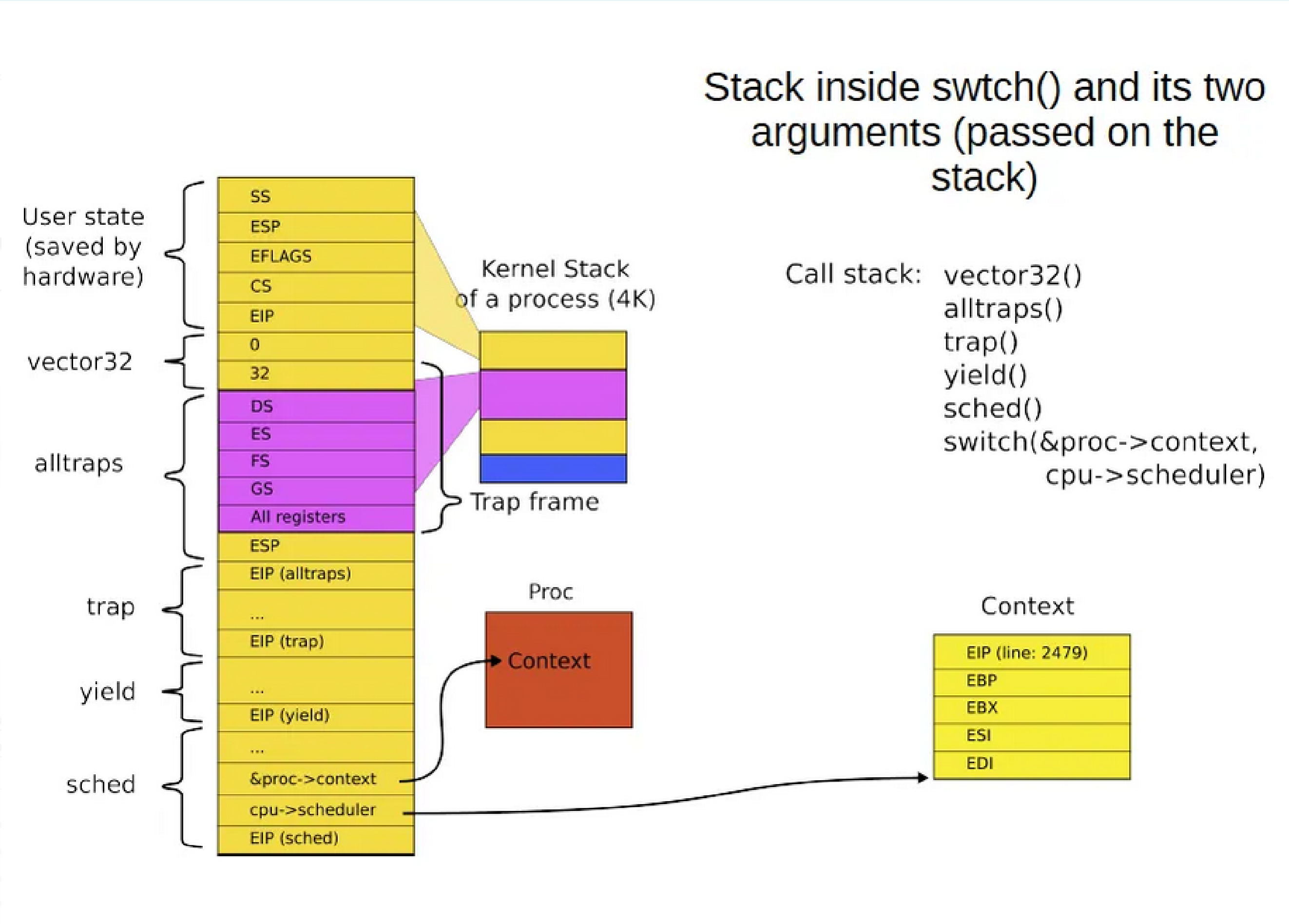
The correct answer is: struct buf is used for disk I/o which takes lot of time, so sleeping/blocking is preferred to spinning/busy-wait for the desired buf.

Question **13**

Partially correct

Mark 0.17 out of 0.50

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



True	False			
◉ ✗	○ ✓	This is a diagram of swtch() called from scheduler()	✗	No. diagram of swtch() called from sched()
◉ ✗	○ ✓	The "context" yellow coloured box, pointed to by cpu->scheduler is on the kernel stack of the scheduler.	✗	
◉ ✓	○ ✗	The "ESP" (second entry from top) is stack pointer of user-stack of process, while the "ESP" (first entry below pink region) is the trapframe pointer on kernel stack of process.	✓	
◉ ✗	○ ✓	The diagram is wrong because it shows the user stack and kernel stack together (continuous), but in practice they are separate	✗	diagram shows only kernel stack
◉ ✓	○ ✗	The blue shaded part in "kernel stack of a process(4k)" refers to remaining part of stack (not used yet)	✓	
○ ✓	◉ ✗	The diagram is correct	✗	

This is a diagram of swtch() called from scheduler(): False
The "context" yellow coloured box, pointed to by cpu->scheduler is on the kernel stack of the scheduler.: False
The "ESP" (second entry from top) is stack pointer of user-stack of process, while the "ESP" (first entry below pink region) is the trapframe pointer on kernel stack of process.: True
The diagram is wrong because it shows the user stack and kernel stack together (continuous), but in practice they are separate: False

The blue shaded part in "kernel stack of a process(4k)" refers to remaining part of stack (not used yet): True
The diagram is correct: True

Question **14**
Correct
Mark 0.50 out of 0.50

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

```
cli
```

from bootasm.S

Why?

- a. "cli" clears all registers and makes them zero, so that processor is as good as "new"
- b. "cli" disables interrupts. It is required because as of now there are no interrupt handlers available
- c. "cli" clears the pipeline of the CPU so that it is as good as "fresh" CPU
- d. "cli" stands for clear screen and the screen should be cleared before OS boots.
- e. "cli" enables interrupts, it is required because the kernel supports interrupts.
- f. It disables interrupts. It is required because the interrupt handlers of kernel are not yet installed. ✔
- g. "cli" that is Command Line Interface needs to be enabled first
- h. "cli" enables interrupts, it is required because the kernel must handle interrupts.

Your answer is correct.

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

Question **15**
Partially correct
Mark 0.20 out of 0.25

Match function with it's meaning

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

Your answer is partially correct.

You have correctly selected 4.
The correct answer is: idewait → Wait for disc controller to be ready, ideinit → Initialize the disc controller, iderw → Issue a disk read/write for a buffer, block the issuing process, idestart → tell disc controller to start I/O for the first buffer on idequeue, ideintr → disk interrupt handler, transfer data from conroller to buffer, wake up processes waiting for this buffer, start I/O for next buffer

Question **16**

Correct

Mark 0.25 out of 0.25

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

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

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

Question **17**

Correct

Mark 0.25 out of 0.25

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

- (•) 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 **18**

Incorrect

Mark 0.00 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 which is currently used as stack by processor
- ( ) d. The "stack" variable declared in "stack.S" in xv6
- ( ) e. The ss:esp pair
- (•) f. The region of memory allocated by kernel for storing the parameters of functions
- ( ) g. The stack segment

Your answer is incorrect.

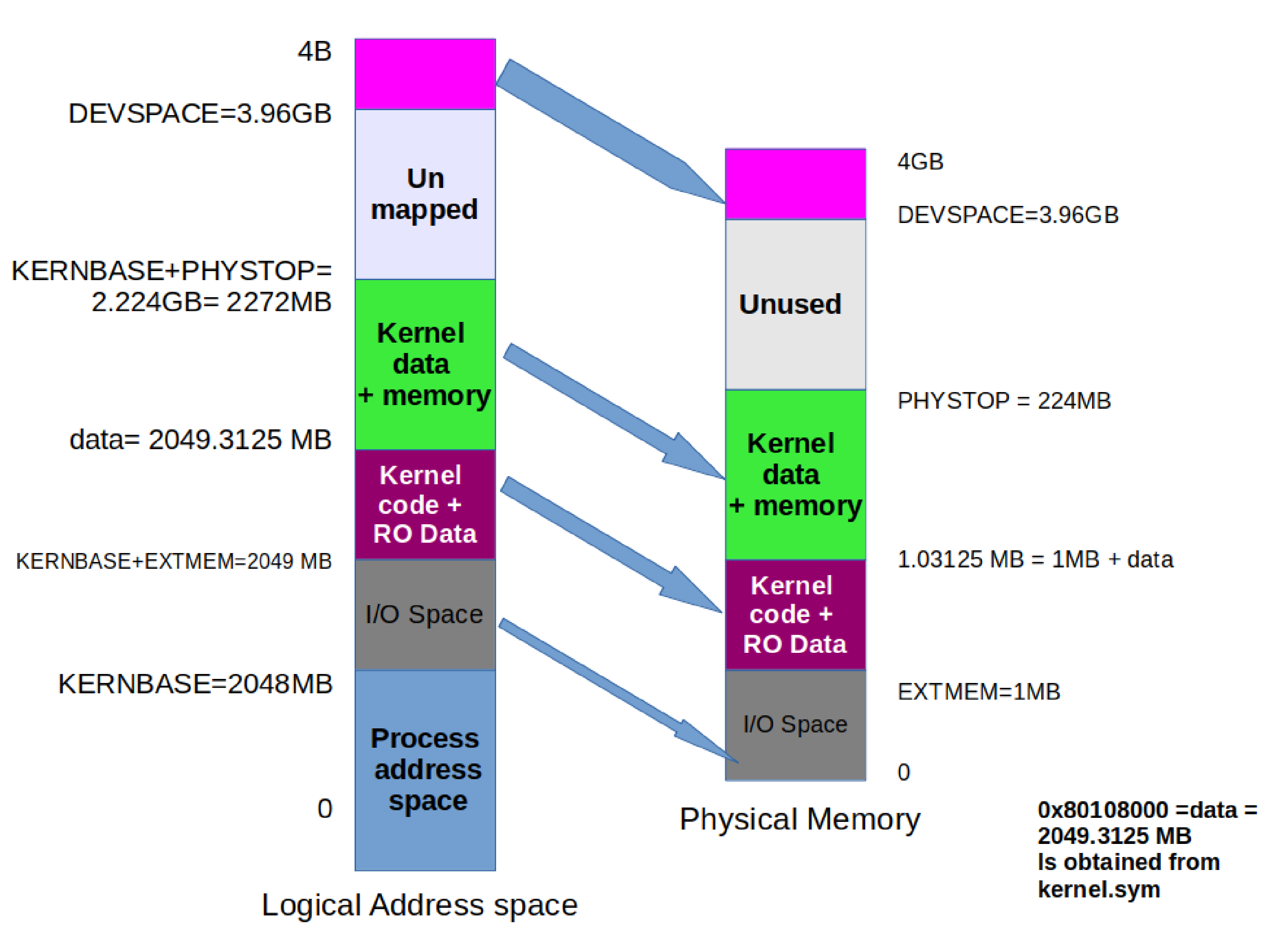
The correct answer is: The ss:esp pair

Question **19**

Correct

Mark 0.50 out of 0.50

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



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.	✔	
◉ ✔	○ ✖	"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 file, after compilation, has maximum virtual address up to "data" as shown in the diagram, which is equal to "end" variable	✔	
◉ ✔	○ ✖	This diagram only shows the absolutely defined virtual->physical mappings, not the mappings defined at run time by kernel.	✔	
◉ ✔	○ ✖	The kernel virtual addresses start from KERNLINK = KERNBASE + EXTMEM	✔	

True	False		
✓		The process's pages are mapped into physical memory from 1.03125 MB to PHYSTOP.	✔

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
"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 file, after compilation, has maximum virtual address up to "data" as shown in the diagram, which is equal to "end" variable: True
This diagram only shows the absolutely defined virtual->physical mappings, not the mappings defined at run time by kernel.: 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

Question **20**

Incorrect

Mark 0.00 out of 0.50

when is each of the following stacks allocated?

kernel stack for scheduler, on first processor	during main()->kinit1()	✖
kernel stack for the scheduler, on other processors	during main()->kinit2()	✖
kernel stack of process	in entry.S	✖
user stack of process	during main()->kinit1()	✖

Your answer is incorrect.

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

◄ Quiz-1(24 Feb 2023)

Jump to...

Pre-requisite Quiz (old) - use it for practice ►