

Department of Electrical Engineering and Computer Science

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

6.1810 Fall 2026

Quiz I

You have 10 minutes to set up your iPad, and then 70 minutes to finish this quiz.

Write your name on this cover sheet AND at the bottom of each page of this booklet.

Some problems are multiple-choice or true/false; we subtract points for incorrect answers.

Some problems are open-ended questions. In order to receive credit you must answer each question as precisely as possible.

Some questions may be harder than others. Read them all first and attack them in the order that allows you to make the most progress. If you find a question ambiguous, write down any assumptions you make. Be neat. If we can't understand your answer, we can't give you credit!

To connect your staff-provided iPad to the internet, press the power button in the top-right corner for about 5 seconds to power it up, then open the Settings app, tap the **`MIT`** wifi network, and enter the following password:

`t*YR!8%!xU`

This exam is open book.
You can use the staff-provided iPad.
You cannot use your own electronic devices.

Gradescope E-Mail Address:

Name:

I System calls

You run the following xv6 program:

```
#include "kernel/types.h"
#include "user/user.h"

int
main()
{
  int x = 0;
  if(fork() == 0){
    x = 1;
    printf("%d\n", x);
    exit(0);
  }
  x = 2;
  printf("%d\n", x);
  exit(0);
}
```

The first time you run it, it prints 2, then 1:

```
2
1
```

1. [6 points]: Which of the following outputs are also possible? (True = possible.)
(Circle True or False for each choice; we subtract points for incorrect answers.)

A. True / False 1, then 1

B. True / False 1, then 2

C. True / False 2, then 2

II Syscall entry/exit

xv6 maps each process's trapframe page into its user page table without `PTE_U`. Ben would like user programs to be able to read their saved registers, so he adds `PTE_U` to the trapframe mapping in `proc_pagetable()`. (Ben also sets the `SUM` bit in `sstatus` so that kernel code can access `PTE_U` pages; you can safely ignore this technicality.)

2. [10 points]: Explain how a malicious user program could take advantage of Ben's change to cause the kernel to invoke any kernel function the malicious user chooses (e.g., the kernel's `main` function).

III Locking

Your classmate Ben is using standard xv6 (with none of the labs, in particular not the COW lab). Ben notes that in `kfree()`, the page being freed cannot possibly be in use by any other RISC-V core or any other kernel thread. Ben decides that, therefore, it is not necessary for `kfree()` to hold a lock around its update of `r->next`. He changes his `kfree()` so that it looks like this (with some debugging code omitted):

```
void
kfree(void *pa)
{
  struct run *r;

  r = (struct run *)pa;

  // the acquire(&kmem.lock) used to be here.

  r->next = kmem.freelist;

  acquire(&kmem.lock); // Ben moved the acquire() to here.
  kmem.freelist = r;
  release(&kmem.lock);
}
```

Ben makes no other changes to xv6.

3. [10 points]: Which of the following problems could arise from Ben's change?
(Circle True or False for each choice; we subtract points for incorrect answers.)

A. True / False There could be a deadlock.

B. True / False The same page could appear twice on `kmem.freelist`.

C. True / False `kalloc()` could return a page that has already been allocated for some other use.

D. True / False Two simultaneous calls to `kfree()` could end up putting only one of the two pages on the free list.

E. True / False `kfree()` could leave `kmem.freelist` pointing to an invalid address (an address with no kernel page table mapping).

IV System call interposition

Recall the `interpose(mask, path)` system call from lab `syscall`: the kernel rejects any system call from the calling process whose bit is set in `mask`, except that a masked `open` or `exec` is allowed if its pathname argument is the same string as `path`. The `sandbox` program forks, calls `interpose()` in the child, and then execs a program in the child. Ben wants to let an untrusted program, called `untrusted`, read the file `README`, but not any other file, so he runs it as follows:

```
$ sandbox 32768 README untrusted
```

where 32768 is `1 << SYS_open`. The file system also contains a file `/secret` that Ben does not want `untrusted` to read. `untrusted` and `README` are both in the root directory, which is the current directory.

4. [10 points]: Write a sequence of system calls, as C code, that `untrusted` could execute to read the contents of `/secret`. Briefly explain why it works.

V Page tables

Ben adds `vmprint()` from the pgtbl lab to an otherwise unmodified xv6 kernel (i.e., without any other lab solutions). He types `echo hello world` to the shell. The shell forks, and the child calls `exec("echo", argv)`. Just before `echo` starts running in user space, `vmprint()` prints `echo`'s page table as follows:

```
page table 0x0000000087f23000
 ..0x0000000000000000: pte 0x0000000021fc7c01 pa 0x0000000087f1f000
 .. ..0x0000000000000000: pte 0x0000000021fc7801 pa 0x0000000087f1e000
 .. .. ..0x0000000000000000: pte 0x0000000021fc801b pa 0x0000000087f20000 RXU
 .. .. ..0x0000000000001000: pte 0x0000000021fc7417 pa 0x0000000087f1d000 RWU
 .. .. ..0x0000000000002000: pte 0x0000000021fc7007 pa 0x0000000087f1c000 RW
 .. .. ..0x0000000000003000: pte 0x0000000021fc6c17 pa 0x0000000087f1b000 RWU
 ..0x0000003fc0000000: pte 0x0000000021fc8801 pa 0x0000000087f22000
 .. ..0x0000003fffe00000: pte 0x0000000021fc8401 pa 0x0000000087f21000
 .. .. ..0x0000003fffffe000: pte 0x0000000021fd0807 pa 0x0000000087f42000 RW
 .. .. ..0x0000003ffffff000: pte 0x0000000020001c0b pa 0x0000000080007000 RX
```

Here is an abridged `vmprint()` of the kernel page table; a line with "`...`" stands for omitted consecutive entries of the same kind (device registers, the rest of physical memory, and the kernel stacks):

```
page table 0x0000000087fff000
 ..0x0000000000000000: pte 0x0000000021fff801 pa 0x0000000087ffe000
 .. ..0x000000000c000000: pte 0x0000000021fff001 pa 0x0000000087ffc000
 .. .. ..0x000000000c000000: pte 0x00000000030000c7 pa 0x000000000c000000 RW
 .. .. ..  ...                                                 (PLIC)
 .. ..  ...
 .. ..0x0000000010000000: pte 0x0000000021fff401 pa 0x0000000087ffd000
 .. .. ..0x0000000010000000: pte 0x00000000040000c7 pa 0x0000000010000000 RW
 .. .. ..0x0000000010001000: pte 0x00000000040004c7 pa 0x0000000010001000 RW
 ..0x0000000080000000: pte 0x0000000021ff7001 pa 0x0000000087fdc000
 .. ..0x0000000080000000: pte 0x0000000021ff6c01 pa 0x0000000087fdb000
 .. .. ..0x0000000080000000: pte 0x000000002000004b pa 0x0000000080000000 RX
 .. .. ..  ...
 .. .. ..0x0000000080007000: pte 0x0000000020001c0b pa 0x0000000080007000 RX
 .. .. ..0x0000000080008000: pte 0x0000000020002047 pa 0x0000000080008000 RW
 .. .. ..  ...
 .. ..  ...
 .. ..0x0000000087e00000: pte 0x0000000021fe7001 pa 0x0000000087f9c000
 .. .. ..0x0000000087e00000: pte 0x0000000021f80007 pa 0x0000000087e00000 RW
 .. .. ..  ...
 .. .. ..0x0000000087f1b000: pte 0x0000000021fc6cc7 pa 0x0000000087f1b000 RW
 .. .. ..  ...
 .. .. ..0x0000000087fff000: pte 0x0000000021fffc47 pa 0x0000000087fff000 RW
 ..0x0000003fc0000000: pte 0x0000000021fe6c01 pa 0x0000000087f9b000
 .. ..0x0000003fffe00000: pte 0x0000000021fe6801 pa 0x0000000087f9a000
 .. .. ..0x0000003ffff7e000: pte 0x0000000021fd6807 pa 0x0000000087f5a000 RW
 .. .. ..  ...                                                 (kernel stacks)
 .. .. ..0x0000003ffffff000: pte 0x0000000020001c4b pa 0x0000000080007000 RX
```

Name:

exec() placed the string argv[1] on echo's user stack at virtual address 0x3fe0. echo calls write(1, argv[1], strlen(argv[1])), and the kernel eventually reaches copyin() with srcva equal to 0x3fe0:

```
int
copyin(pagetable_t pagetable, uint64 psz, char *dst, uint64 srcva, uint64 len)
{
  uint64 n, va0, pa0;

  while (len > 0) {
    va0 = PGROUNDDOWN(srcva);
    pa0 = walkaddr(pagetable, va0);
    if (pa0 == 0) {
      if ((pa0 = vmfault(pagetable, psz, va0, 1)) == 0) {
        return -1;
      }
    }
    n = PGSIZE - (srcva - va0);
    if (n > len)
      n = len;
    memmove(dst, (void *)(pa0 + (srcva - va0)), n);

    len -= n;
    dst += n;
    srcva = va0 + PGSIZE;
  }
  return 0;
}
```

5. [8 points]: What value does copyin() compute for pa0 when it copies argv[1]?
(Circle the one best choice; we subtract points for incorrect answers.)

A. 0x3000

B. 0x21fc6c17

C. 0x87f1b000

D. 0x87f1bfe0

6. [12 points]: Ben notices that the kernel page table maps all of physical memory, and concludes that `walkaddr()` is unnecessary. He changes the `memmove` line to the following, leaving the rest of `copyin()` unchanged:

```
memmove(dst, (void *)srcva, n);
```

What happens when `echo` runs `write(1, argv[1], ...)` on Ben's kernel?

(Circle the one best choice; we subtract points for incorrect answers.)

A. The `write` works correctly, because the kernel page table maps all of physical memory, including the user stack page at `0x87f1b000`.

B. `memmove()` reads from physical address `0x3fe0`, which does not hold `argv[1]`, so `echo` prints garbage.

C. `memmove()` causes a page fault in supervisor mode; `kerneltrap()` then panics.

D. `memmove()` causes a page fault; `vmfault()` maps a page at `0x3000` in the kernel page table, the load is retried, and the `write` works correctly.

VI Interrupts

Ben wants to find out the process IDs of the programs that read from the UART. He adds some lines to `uartgetc()` in `kernel/uart.c`:

```
static int
uartgetc(void)
{
  if (ReadReg(LSR) & LSR_RX_READY) {
    struct proc *p = myproc();            // added by Ben
    if(p != 0)                            // added by Ben
      printk("%d is reading\n", p->pid);  // added by Ben
    return ReadReg(RHR);
  } else {
    return -1;
  }
}
```

Recall that `uartgetc()` is called from `uartintr()`, the UART device interrupt handler. Ben has not made any other changes to xv6.

Ben boots xv6 and spends a lot of time typing and running various commands.

7. [10 points]: Which one of the following statements is true about Ben's changes to `uartgetc()`? **(Circle the one best choice; we subtract points for incorrect answers.)**

A. Ben's new code will always crash because `mycpu()`, which `myproc()` calls, will return 0 instead of a valid `struct cpu *`.

B. Ben's new code might crash because `p` has a good chance of being a non-zero but invalid pointer.

C. Ben's new code will never print anything because `p` is guaranteed always to be 0.

D. Ben's new code may print process IDs of processes that are **not** actually trying to read from the UART.

E. Ben's new code will act as Ben intends: if it prints anything, it will print the process ID of a process that is reading from the UART.

VII 6.1810

8. [1 points]: What lab so far has been your *least* favorite, and why?

- ☐ util — **Why:**
- ☐ syscall
- ☐ pgtbl
- ☐ traps
- ☐ cow
- ☐ lock

9. [1 points]: Which of the papers so far do you think we should delete next year, and why?

- ☐ Syscall interposition — **Why:**
- ☐ Transparent superpages
- ☐ VM primitives for user programs
- ☐ Microkernels

10. [1 points]: What do you like or dislike about lab checkoff meetings?

11. [1 points]: What should we improve about the staff-supplied iPad plan for the final exam?

End of Quiz I

Name: