Department of Electrical Engineering and Computer Science

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

6.1810 Fall 2024

Quiz II

You have 120 minutes to finish this quiz.

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

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

THIS EXAM IS OPEN BOOK AND OPEN LAPTOP, but CLOSED NETWORK.

Please do not write in the boxes below.

I (xx/15)	II (xx/10)	III (xx/10)	IV (xx/5)	V (xx/5)	VI (xx/10)	VII (xx/5)	VIII (xx/10)	IX (xx/2)	(xx/72)

Gradescope E-Mail Address:

Name:

I The xv6 file system and lab fs

Ben makes a fresh `fs.img`, boots xv6, and runs the following commands:

```
$ mkdir a
$ mkdir a/b
```

1. [5 points]: How many inodes will xv6 allocate while executing these two commands? (Circle the one best answer.)

A. 0

B. 1

C. 2

D. 3

Alyssa adds the statement:

```
printf("write: %d\n", b->blockno);
```

to xv6's `log_write` in `log.c`. She then makes a fresh `fs.img`, boots xv6, and runs the following command:

```
$ mkdir a
write: 33
write: 33
write: 45
write: 770
write: 770
write: 33
write: 770
write: 33
write: 46
write: 32
write: 32
```

2. [5 points]: What does block 770 contain? (Circle the one best answer.)

A. directory entries

B. inodes

C. file data

D. a bitmap

Ben makes a fresh `fs.img`, boots xv6, and runs a program that makes the following system call:

```
  symlink("b", "b");
```

From the shell he then runs:

```
$ cat b
```

3. [5 points]: What will the result of the `cat` be? (Circle the one best answer.)

A. "b"

B. an error because "b" doesn't exist

C. an error because "b" points to itself

D. nothing because xv6 will panic

II EXT3

Recall the Linux EXT3 journaling file system from *Journaling the Linux ext2fs Filesystem* and Lecture 15. The paper's "ext2fs" is the same as EXT3.

Suppose that the current compound transaction has just closed (see step 1 on the paper's page 6) and is starting to commit.

4. [5 points]: How long must new file-system system calls wait until they can start executing? (Circle the one best answer.)

A. New system calls can start immediately.

B. New system calls must wait until all system calls in the just-closed transaction have completed.

C. New system calls must wait until the just-closed transaction has started to write journal blocks to the journal.

D. New system calls cannot start until the just-closed transaction has finished committing to the journal.

E. New system calls cannot start until all updated buffers from the just-closed transaction have been synced to their homes on disk.

Hatshepsut is building an application on Linux that creates a set of directories, and she would like the set of creations to be atomic with respect to crashes. She's using the EXT3 file system. She experiments with this application code:

```
int main() {
  mkdir("/aaa", 0777);
  mkdir("/zzz", 0777);
  exit(0);
}
```

(The 0777 is needed for Linux, though not for xv6; it does not affect this question.)

Hatshepsut runs this program. Both calls to `mkdir()` return success. Hatshepsut causes her computer to crash just after the program exits. Then she re-starts the computer, which runs the EXT3 recovery program.

5. [5 points]: What could Hatshepsut see after recovery? (Circle all that apply.)

A. She might see neither `/aaa` nor `/zzz`.

B. She might see `/aaa` but not `/zzz`.

C. She might see `/zzz` but not `/aaa`.

D. She might see both `/zzz` and `/aaa`.

E. None of the above.

III VM primitives

Below is a code fragment illustrating how a user program can implement a large table of square roots with Linux VM primitives while using little physical memory. (The full code presented in lecture is in the appendix of this quiz.)

```c
static size_t page_size;
#define MAX_SQRTS       (1 << 27) // Maximum limit on sqrt table entries

static double *sqrts;

// The page handler catching page faults
static void
handle_sigsegv(int sig, siginfo_t *si, void *ctx)
{
  uintptr_t fault_addr = (uintptr_t)si->si_addr;
  double *page_base = (double *)align_down(fault_addr, page_size);
  static double *last_page_base = NULL;

  if (last_page_base && munmap(last_page_base, page_size) == -1) {
    fprintf(stderr, "Couldn’t␣munmap();␣%s\n", strerror(errno));
    exit(EXIT_FAILURE);
  }

  if (mmap(page_base, page_size, PROT_READ | PROT_WRITE,
           MAP_PRIVATE | MAP_ANONYMOUS | MAP_FIXED, -1, 0) == MAP_FAILED) {
    fprintf(stderr, "Couldn’t␣mmap();␣%s\n", strerror(errno));
    exit(EXIT_FAILURE);
  }

  calculate_sqrts(page_base, page_base - sqrts, page_size / sizeof(double));
  last_page_base = page_base;
}

// Simplified version of the test function
static void
test_sqrt_region(void)
{
  int i, pos;
  double s;

  // Find a sufficiently-large unused range of virtual addresses, and
  // sets sqrts to the start.
  setup_sqrt_region();

  // look up some numbers in the sqrt table
  for (i = 0; i < 8192; i++) {
    s = sqrts[i];
    printf("sqrt␣%f", s);
  }
}
```

Assume size of `double` is 8 bytes and `page_size` is 4096 bytes.

6. [5 points]: Assume the `sqrts` table occupies 0 pages of physical memory after the return from `setup_sqrt_region`. How many pages of physical memory does the `sqrts` table occupy when `test_sqrt_region` returns? (You can ignore physical memory pages used for the page table itself.) (Circle the one best answer.)

A. 0

B. 1

C. 1000

D. $((1 \ll 27) * 8) / 4096$

7. [5 points]: How many total page faults will the repeated execution of line 42 cause? (Circle the one best answer.)

A. 0

B. 1

C. 2

D. 16

E. 8192

IV L^{4}Linux

Consider *The Performance of μ-Kernel-Based Systems*, by Härtig *et al.*, along with Lecture 17.

Suppose that an `sh` Linux process running under L^{4}Linux performs a `fork()`.

8. [5 points]: Which of the following are true? (Circle all that apply.)

A. The L4 kernel's `fork()` system call copies the `sh` process's memory.

B. When the Linux kernel server task has finished executing the system call implementation, it executes the x86 equivalent of RISC-V `sret` to return to the `sh` process.

C. When the Linux kernel server task returns to the newly created child process, the Linux kernel changes the hardware page table register (equivalent of RISC-V satp) to point to the child process's page table.

D. Copy-on-write `fork()` is not possble for L^{4}Linux because the CPU delivers page faults to the L4 kernel, not to the Linux kernel task.

E. None of the above.

V RedLeaf

Consider *RedLeaf: Isolation and Communication in a Safe Operating System* by Narayanan *et al*.

9. [5 points]: Which of the following are true statements about RedLeaf's design? (Circle all that apply.)

A. Because the RedLeaf microkernel and domains run in a single address space, a domain can read any kernel memory by dereferencing a Rust pointer.

B. User programs can avoid data copies by passing pointers to their private memory to other user programs.

C. Two domains can have a Rust pointer to an object on the shared heap at the same time.

D. The rv6 file system can be modified to support memory-mapped files using the same ideas as in the mmap lab without modifying the RedLeaf microkernel.

E. A divide-by-zero error in the network domain won't crash the rv6 file system.

F. None of the above.

VI Networking lecture/reading

Consider *Eliminating Receive Livelock in an Interrupt-driven Kernel*, by Mogul *et al.*, and Lecture 20.

Ben implements the paper's polling design (Section 6.4), in which the NIC interrupt handler just wakes up the polling thread. However, Ben's implementation leaves NIC interrupts enabled (in contrast to Section 6.4, which specifies that they be disabled until the polling thread is done).

Ben's computer has just one CPU (i.e. just a single core).

10. [5 points]: What will Ben observe as the rate of packet arrivals increases? (Circle the one best answer.)

A. He won't see livelock, because the interrupt handler doesn't process the packets; only the polling thread handles the packets.

B. He won't see livelock, because the polling design eliminates the IP input queue, which was the point at which packets were discarded in the old design.

C. He will see livelock, because at high enough arrival rates the CPU will spend all its time in the polling thread.

D. He will see livelock, because at high enough arrival rates the CPU will spend all its time in the interrupt handler.

E. He will see livelock, because the polling thread can only process packets at some finite rate, and the input rate could be higher than that.

Name:

Zoe's xv6 computer has a UART that has no limit on how fast it can transmit bytes. The UART interrupts once per byte transmitted, to indicate that it has finished transmitting the byte. Zoe has a program whose standard output (file descriptor 1) is connected to the xv6 console, which uses the UART; the program sends bytes as fast as it can:

```
while(1){
  char c = 'x';
  write(1, &c, 1);
}
```

Zoe's computer has just one CPU (i.e. just a single core).

11. [5 points]: Could this program cause interrupt livelock due to the CPU spending all its time in the UART interrupt handler, and thus no time executing Zoe's program? Explain briefly.

VII Meltdown

Below is Listing 2 of the paper *Meltdown: reading kernel memory from user space* by Lipp *et al.*, written in a C-like notation instead of x86 assembly.

```
 1  char buf[8192]
 2
 3  // The flush part of Flush+Reload
 4  cflush buf[0]
 5  cflush buf[4096]
 6
 7  // The core attack from listing 2
 8  r1 = 0x79cbcc0   // a kernel virtual address
 9  r2 = *r1
10  r2 = r2 & 1
11  r2 = r2 * 4096
12  r3 = buf[r2]
```

12. [5 points]: Which of the following are true statements? (Circle all that apply.)

A. In Linux as described in the paper, page tables of user programs map all of kernel memory.

B. Loading the value at kernel address 0x79cbcc0 on line 9 will lead to an exception.

C. If the attack succeeds, then `buf[0]` will be in the L1 cache if the low bit of the value at address 0x79cbcc0 is a 0.

D. One reason why one run of Meltdown might not succeed is because `buf[0]` maybe evicted from the L1 cache before the attack can measure its presence using Reload.

E. The Meltdown attack on xv6 wouldn't be able to dump all of xv6 kernel memory because like KAISER the xv6 kernel and user processes have separate page tables.

F. None of the above.

Name:

VIII RCU

Ben has a Linux kernel that uses RCU as described in *RCU Usage In the Linux Kernel: One Decade Later*, by McKenney *et al.* He modifies `udp_sendmsg()` in the paper's Figure 6, adding a call to `new_function()` on line 8, so that the code now reads:

```
 1 void udp_sendmsg(sock_t *sock, msg_t *msg)
 2 {
 3   ip_options_t *opts;
 4   char packet[];
 5   copy_msg(packet, msg);
 6   rcu_read_lock();
 7   opts = rcu_dereference(sock->opts);
 8   new_function(); // *** Ben adds this line. ***
 9   if (opts != NULL)
10     copy_opts(packet, opts);
11   rcu_read_unlock();
12   queue_packet(packet);
13 }
14 void setsockopt(sock_t *sock, int opt, void *arg)
15 {
16   if (opt == IP_OPTIONS) {
17     ip_options_t *old = sock->opts;
18     ip_options_t *new = arg;
19     rcu_assign_pointer(&sock->opts, new);
20     if (old != NULL)
21       call_rcu(kfree, old);
22     return;
23   }
24 }
```

This code is otherwise identical to the paper's Figure 6.

`new_function()` performs a context switch (i.e., it calls the Linux equivalent of xv6's `sleep()` or `yield()`).

Name:

13. [5 points]: Ben has made a mistake. Explain a scenario in which something goes wrong with the Figure 6 code as a result of Ben's change.

Now Ben is working on the code in the RCU paper's Figure 7. He reasons that the `kfree(local_table)` in `retract_table()` really belongs inside the critical section, so that the entire sequence is atomic. He moves that line, resulting in this code:

```
...;
spin_lock(&table_lock);
local_table = table;
rcu_assign_pointer(&table, NULL);
kfree(local_table); // *** Ben moved this line. ***
spin_unlock(&table_lock);
...;
```

14. [5 points]: What problem is Ben's change likely to cause? (Circle the one best answer.)

A. Ben's change could cause a deadlock.

B. Ben's change could allow a context switch to occur just before the `kfree()` call, which would be illegal.

C. Ben's change could cause `invoke_syscall()` to dereference a pointer to freed memory.

D. Ben's change could cause `retract_table()` to dereference a pointer to freed memory.

IX 6.1810

15. [1 points]: Please indicate which of the labs you found to be the most helpful, and which the least.

- Networking
- Locking
- File system
- mmap

16. [1 points]: Which paper is the best candidate for deletion in future years?

End of Quiz II — Happy holidays!

X Appendix: mmap.c

```c
#include <stdlib.h>
#include <stdio.h>
#include <errno.h>
#include <unistd.h>
#include <string.h>
#include <signal.h>
#include <stdint.h>
#include <sys/mman.h>
#include <sys/resource.h>
#include <sys/time.h>
#include <math.h>

static size_t page_size;

// align_down - rounds a value down to an alignment
// @x: the value
// @a: the alignment (must be power of 2)
//
// Returns an aligned value.
#define align_down(x, a) ((x) & ~((typeof(x))(a) - 1))

#define AS_LIMIT        (1 << 23) // Maximum limit on virtual memory bytes
#define MAX_SQRTS       (1 << 27) // Maximum limit on sqrt table entries
static double *sqrts;

static int nfault;

// Use this helper function as an oracle for square root values.
static void
calculate_sqrts(double *sqrt_pos, int start, int nr)
{
  int i;

  for (i = 0; i < nr; i++)
    sqrt_pos[i] = sqrt((double)(start + i));
}

static void
handle_sigsegv(int sig, siginfo_t *si, void *ctx)
{
  uintptr_t fault_addr = (uintptr_t)si->si_addr;
  double *page_base = (double *)align_down(fault_addr, page_size);
  static double *last_page_base = NULL;

  if (last_page_base && munmap(last_page_base, page_size) == -1) {
    fprintf(stderr, "Couldn't␣munmap();␣%s\n", strerror(errno));
    exit(EXIT_FAILURE);
  }
  if (mmap(page_base, page_size, PROT_READ | PROT_WRITE,
           MAP_PRIVATE | MAP_ANONYMOUS | MAP_FIXED, -1, 0) == MAP_FAILED) {
    fprintf(stderr, "Couldn’t␣mmap();␣%s\n", strerror(errno));
    exit(EXIT_FAILURE);
  }

  nfault++;

  calculate_sqrts(page_base, page_base - sqrts, page_size / sizeof(double));
  last_page_base = page_base;
}

static void
setup_sqrt_region(void)
{
  struct rlimit lim = {AS_LIMIT, AS_LIMIT};
  struct sigaction act;

  // Only mapping to find a safe location for the table.
  sqrts = mmap(NULL, MAX_SQRTS * sizeof(double) + AS_LIMIT, PROT_NONE,
               MAP_PRIVATE | MAP_ANONYMOUS, -1, 0);
  if (sqrts == MAP_FAILED) {
    fprintf(stderr, "Couldn’t␣mmap()␣region␣for␣sqrt␣table;␣%s\n",
            strerror(errno));
    exit(EXIT_FAILURE);
  }

  // Now release the virtual memory to remain under the rlimit.
  if (munmap(sqrts, MAX_SQRTS * sizeof(double) + AS_LIMIT) == -1) {
    fprintf(stderr, "Couldn’t␣munmap()␣region␣for␣sqrt␣table;␣%s\n",
            strerror(errno));
    exit(EXIT_FAILURE);
  }

  // Set a soft rlimit on virtual address-space bytes.
  if (setrlimit(RLIMIT_AS, &lim) == -1) {
    fprintf(stderr, "Couldn’t␣set␣rlimit␣on␣RLIMIT_AS;␣%s\n", strerror(errno));
    exit(EXIT_FAILURE);
  }

  // Register a signal handler to capture SIGSEGV.
  act.sa_sigaction = handle_sigsegv;
  act.sa_flags = SA_SIGINFO;
  sigemptyset(&act.sa_mask);
  if (sigaction(SIGSEGV, &act, NULL) == -1) {
    fprintf(stderr, "Couldn’t␣set␣up␣SIGSEGV␣handler;,␣%s\n", strerror(errno));
    exit(EXIT_FAILURE);
  }
}

static void
```

```c
test_sqrt_region(void)
{
  int i, pos = rand() % (MAX_SQRTS - 1);
  double correct_sqrt;
  struct timeval start, end;
  long secs_used,micros_used;

  printf("Validating␣square␣root␣table␣contents...\n");
  srand(0xDEADBEEF);

  gettimeofday(&start, NULL);

  for (i = 0; i < 1000; i++) {
    pos = rand() % (MAX_SQRTS - 1);
    calculate_sqrts(&correct_sqrt, pos, 1);
    if (sqrts[pos] != correct_sqrt) {
      fprintf(stderr, "Square␣root␣is␣incorrect.␣Expected␣%f,␣got␣%f.\n",
              correct_sqrt, sqrts[pos]);
      exit(EXIT_FAILURE);
    }
  }
  gettimeofday(&end, NULL);

  printf("start:␣%d␣secs,␣%d␣usecs\n",start.tv_sec,start.tv_usec);
  printf("end:␣%d␣secs,␣%d␣usecs\n",end.tv_sec,end.tv_usec);

  secs_used=(end.tv_sec - start.tv_sec); //avoid overflow by subtracting first
  micros_used= ((secs_used*1000000) + end.tv_usec) - (start.tv_usec);

  printf("micros_used:␣%d␣for␣nfault␣%d\n",micros_used, nfault);
}

int
main(int argc, char *argv[])
{
  page_size = sysconf(_SC_PAGESIZE);
  printf("page_size␣is␣%ld\n", page_size);
  setup_sqrt_region();
  test_sqrt_region();
  return 0;
}
```