Duration: 2 hour 30 min

Examiner: A. Goel

Please fill your student number and name below and then read the instructions below carefully.

Student Number: _____

First Name: _____

Last Name: _____

MARKING GUIDE

Q1:	_____	(8)
Q2:	_____	(6)
Q3:	_____	(12)
Q4:	_____	(12)
Q5:	_____	(12)
Q6:	_____	(12)
Q7:	_____	(8)
Q8:	_____	(8)
Q9:	_____	(8)
Q10:	_____	(14)
BONUS MARKS:	_____	1
TOTAL:	_____	(100)

Instructions

Examination Aids: No examination aids are allowed.

Do not turn this page until you have received the signal to start.

Do not remove any sheets from this test book. Answer all questions in the space provided. No additional sheets are permitted. Use the last blank page as scratch space. This page will not be marked.

This exam consists of 10 questions on 15 pages (including this page). The value of each part of each question is indicated. The total value of all questions is 100.

For the written answers, explain your reasoning clearly. Be as brief and specific as possible. Clear, concise answers will be given higher marks than vague, wordy answers. Marks will be deducted for incorrect statements in an answer. Please write legibly!

Work independently.

Question 1. True / False [8 MARKS]

TRUE (FALSE)

A thread of a process can read but not modify the local variables of another thread in the same process.

A thread has access to the entire address space of a process and can read or modify any part of it.

TRUE (FALSE)

The atomic test-and-set instruction requires that interrupts are disabled when the instruction executes.

The programmer doesn't have to disable interrupts when issuing atomic instructions

TRUE (FALSE)

The shorter the time slice, the more the round-robin scheduler gives results that are similar to a FIFO scheduler.

The *longer* the time slice, the more the round-robin scheduler is similar to a FIFO scheduler.

(TRUE) FALSE

In a paging system, a context switch requires modifying the page table base register.

True when h/w manages TLB. Even with software-managed TLB, the OS needs to change the page table base address (the page table that is used).

TRUE (FALSE)

A MMU would not be useful if the machine has more physical memory than the sum total of the virtual address space size of all running programs.

A MMU is useful because it allows virtualizing addresses. For example, program addresses can start at 0, which simplifies compiling, loading and debugging programs.

(TRUE) FALSE

The page fault frequency algorithm helps decide how pages should be allocated across processes but not within processes.

TRUE (FALSE)

A program compiled with dynamically linked libraries has a smaller working set than the same program compiled with statically linked libraries.

The working set is based on the pages used by a process. This doesn't change, whether dynamic or static libraries are being used. Dynamic libraries enable sharing reducing the amount of physical memory frames that need to be allocated.

(TRUE) FALSE

Compared to a regular file system, the log-structured file system improves the performance of file writes, but reduces the performance of file reads.

LFS improves write performance because it mostly writes blocks sequentially. It reduces read performance because file data can get spread over the disk, as parts of it are overwritten.

Question 2. Atomic or Not [6 MARKS]

The fetch-and-add hardware instruction *atomically* increments a value while returning the old value at a particular address. The C pseudocode for the fetch-and-add instruction looks like this:

```
int fetch_and_add(int *ptr)
{
    int old = *ptr;
    *ptr = old + 1;
    return old;
}
```

The fetch-and-add instruction can be used to build a *ticket lock*, as shown below. When a thread wishes to acquire a lock, it first does an atomic fetch-and-add on the ticket value. The returned value is considered this thread's "turn" (myturn). The shared value `l->turn` is then used to determine whether this thread can acquire the lock. When `myturn == l->turn` for a given thread, it is that thread's turn to enter the critical section. The lock is released by incrementing `l->turn` so that the next waiting thread (if there is one) can now enter the critical section.

```
struct lock {
    int ticket;
    int turn;
};

void lock_init(struct lock *l)
{
    l->ticket = 0;
    l->turn = 0;
}

void acquire(struct lock *l)
{
    int myturn = fetch_and_add(&l->ticket);
    while (myturn != l->turn)
        ; // spin
}

void release(struct lock *l)
{
    l->turn = l->turn + 1;
}
```

Part (a) [4 MARKS] Alice and Bob look at this code for a long time. Bob is convinced that the `release` code has a race (he suggests using `fetch_and_add` to increment `l->turn` to fix the race). Alice is convinced that the ticket lock code shown above is correct. Who is correct? Why?

Alice is correct. There is no race because l->turn is only updated by release(), and only the thread that calls acquire() calls release(). The acquire() function only reads the value of l->turn. It will either get the old or the new value of l->turn but this will not affect the correctness of the code.

Part (b) [2 MARKS] Assuming that Alice and Bob figure out how to implement the ticket lock correctly, would there be any benefit to using a ticket lock over a spin lock?

The ticket lock ensures fairness since each thread gets a ticket on arrival, unlike spinlocks. Also, ticket locks can be more efficient because they perform a regular read instruction instead of an atomic instruction while spinning in the acquire() code.

Question 3. My Way or the Highway [12 MARKS]

The Bloor Viaduct Bridge is undergoing repairs and only one lane is open for traffic. To prevent accidents, traffic lights have been installed at either end of the bridge to synchronize the traffic going in different directions. A car can only cross the bridge if there are no cars going in the opposite direction on the bridge. Sensors at either end of the bridge detect when cars arrive and depart from the bridge, and these sensors control the traffic lights. A skeleton implementation of two routines, `Arrive()` and `Depart()` is shown below. You may assume that each car is represented by a thread, and threads call `Arrive()` when they arrive at the bridge and `Depart()` when the leave the bridge. Threads pass in their direction of travel as input to the `Arrive()` routine.

```
int num_cars = 0;
enum dir = {open, east, west};
dir cur_dir = open;

   void
   Arrive(dir my_dir)
   {

        lock(lock);
A1:     while (cur_dir != my_dir &&
               cur_dir != open) {
               wait(cv, lock);

        }


A2:     num_cars++;


A3:     cur_dir = my_dir;
        unlock(lock);

   }
```

```
Lock *lock = new Lock();
Condition *cv = new Condition();


void
Depart()
{

     lock(lock);
D1:   num_cars--;


D2:  if (num_cars == 0) {


D3:       cur_dir = open;
          broadcast(cv, lock);

     }
     unlock(lock);

}
```

Note that a broadcast(), not a signal() needs to be placed within the "if" statement, or else cars will get blocked.

Alternatively, a signal can be placed after the "if" statement, just before the "unlock"

Part (a) [4 MARKS] The code shown above does not do any synchronization. Show how two cars may travel in opposite directions at the same time. Use T1:E:A1 (or T1:W:A1) to indicate Thread T1 going East (or West) while executing statement A1.

T1:E:A1
T1:E:A2 (now num_cars has been incremented, but cur_dir has not been updated)
T2:W:A1 (another thread can enter because cur_dir is still "open")
T2:W:A2 ...

Part (b) [6 MARKS] Show how a lock and condition variable can be used to correctly synchronize the cars. Annotate the code above with calls to `lock()`, `unlock()`, `wait()`, `signal()`, and `broadcast()` operations (together with their arguments). Use only the operations that are needed or your answer will be penalized.

Part (c) [2 MARKS] Is there a problem with your solution? If so, give an example.

Even though the solution above synchronizes cars correctly, it can cause starvation. Cars going one way can starve cars going the other way (similar to reader-writer problem).

Question 4. Trick the Scheduler [12 MARKS]

A (very) clever student in your class, called Trickster, has figured out a way to guess when timer interrupts are generated by hardware (the student knows that the interrupts are generated periodically, and has somehow determined this period). Trickster knows that the UG machines run the Unix feedback scheduler. Trickster runs a program called Sneaky that repeatedly performs a computation (uses the CPU), puts itself to sleep for the short duration during which the timer interrupt fires, and then wakes up to continue performing the computation.

Part (a) [4 MARKS] Clearly explain why the Sneaky program is able to trick the feedback scheduler. How does the program benefit from this trick? Hint: It will help to think about each of the steps in the scheduler.

The feedback scheduler updates usage on each timer interrupt for the currently running thread. Sneaky will appear to not be running at all, so its usage will remain 0. This will give Sneaky the highest priority among all processes and it will get to run most of the time.

Part (b) [4 MARKS] Clearly explain how you would change the feedback scheduler so that it can still achieve its goals. Explain, as precise as possible, why your change will fix the problem above.

The problem above is that we are updating the usage value periodically. Instead, if we kept track of the time that a process actually ran, the problem above would not happen. To do so, we can track the time when a process is run (on context switch) and when it stops running, i.e., it is put back in the ready/sleep queue (again on context switch).

Part (c) [4 MARKS] Outline a way that Trickster may have been able to figure out the period of the timer interrupt. You may assume that Trickster works in the night, when no one else is running their programs. non-hint: making the program sleep will likely not be helpful.

Trickster could write a program that read the computer clock (gettimeofday) in a tight loop. It would notice that the loop executes each iteration every few nanoseconds (assuming a GHz machine), but when the timer interrupt fires, the time difference between successive calls to gettimeofday would be much higher (since interrupt processing takes significant time). Plotting these results would show clear periodic behavior.

Question 5. Missing the TLB [12 MARKS]

TLB misses can be nasty. The following code can cause a lot of TLB misses, depending on the values of STRIDE and MAX. Assume that your system has a 32-entry TLB with a 8KB (2^{13} bytes) page size. Assume that the code runs on a 32-bit machine architecture. Also, assume that `malloc` below returns a page-aligned address.

```
int value = 0;
int *data = malloc(sizeof(int) * MAX); // an array of MAX integers
for (int j = 0; j < 1000; j++) {
    for (int i = 0; i < MAX; i += STRIDE) {
        value = value + data[i];
    }
}
```

Part (a) [6 MARKS] Choose the minimum value for `MAX` and for `STRIDE` so that every access to the `data` array causes a TLB miss (100% TLB miss rate). Use the space below to show any calculations. If you make any assumptions, state them below.

Assuming that the TLB entries are replaced using LRU (or randomly, or any reasonably replacement algorithm), if the array is 33 pages long, and a new page is accessed when data[i] is accessed, then each such access will likely cause a TLB miss. This will not cause a page fault because 33 pages is just 33 * 8KB = 264KB of memory, which is pretty small on modern machines.

`STRIDE =` STRIDE >= 8KB/sizeof(int) = 8KB/4 = 2048

We divide by sizeof(int) above because data is an int array. With a stride value of 2048, each read of data[i] will access a new page.

`MAX =` MAX >= 33 * STRIDE = 33 * 2048 = 67584

If the answer mentioned that the local variables and code will require 1 page each, we also accepted 31 * STRIDE.

Part (b) [2 MARKS] Would the *page fault* rate 1) increase, 2) remain the same, or 3) decrease, if the MAX value is made 10 times the value of MAX you have chosen in Part(a) (STRIDE is unchanged)? Explain your answer.

Assuming any contention (other programs are running), page fault will increase.
Page fault will remain the same, only if a lot of physical memory is available.

Part (c) [2 MARKS] Would the *page fault* rate 1) increase, 2) remain the same, or 3) decrease, if the MAX and STRIDE values are both made 10 times the values of MAX and STRIDE you have chosen in Part(a)? Explain.

Page fault will remain the same because the number of unique pages that are accessed compared to Part(a) is the same.

Part (d) [2 MARKS] Say we doubled the TLB size (64 entries). With the MAX and STRIDE values you have chosen in Part(a), would the *TLB miss* rate be 1) less than 10%, 2) around 50%, or 3) greater than 90%?

The TLB miss rate will be less than 10%, likely close to 0, because the number of TLB entries is larger than 33, and we are accessing 33 pages repeatedly.

Question 6. **Paging Redone** [12 MARKS]

An 16-bit machine with a paging MMU has a virtual address space size of 256 bytes, and a page size of 16 bytes. The machine uses a linear page table and the page table entry (PTE) uses 8 bits, with the format shown below. Note that the valid bit is the least significant bit in the PTE.

	7 bits	1 bit
PTE:	frame number	valid bit

Part (a) [4 MARKS] The memory dump of the page table, located at physical address 0, shows the following:

```
00: 0xa9 0xb4 0xbf 0xc2 0xff 0xd7 0xfe 0x03 0xfd 0x09 0x10 0x87 0x39 0x7f 0x6e 0x05
```

For the following virtual addresses, say whether it is a *valid* virtual address or will cause a *fault*. For those that are valid, write the physical address (in hex) that results from the translation.

	Fault/Valid	Physical Address (in hex)
virtual address 0x65:	0x65: '6' maps to 6th PTE = 0xfe. Lowest bit is unset => invalid => fault.	
virtual address 0x0c:	0x0c: '0' maps to 0th PTE = 0xa9. Lowest bit is set => valid frame nr = 1010100 = 0x54, paddr =	0x54c
virtual address 0x26:	0x26: '2' maps to 2nd PTE = 0xbf. Lowest bit is set => valid frame nr = 1011111 = 0x5f, paddr =	0x5f6
virtual address 0xa8:	0xa8: 'a' maps to 10th PTE = 0x10. Lowest bit is unset => invalid => fault.	

This question continues on the following page.

Question 6. **Paging Redone** (CONTINUED)

Part (b) [8 MARKS] The designers of the 16-bit machine wanted to run larger programs. They decided to update the paging MMU to use a two-level page table, supporting a virtual address space of 4096 bytes. The designers carefully ensured that the top and the second-level page tables have the same size, and the format of the page table entry (PTE) in both of them is the same as shown in the previous page. No other changes were made to the architecture.

Consider the partial memory dump of the page table, located at physical address 0:

```
00:0xa9 0xb4 0xbf 0xc2 0xff 0xd7 0xfe 0x03 0xfd 0x09 0x10 0x87 0x39 0x7f 0x6e 0x05
16:0xb2 0xb7 0x7e 0xbe 0x93 0xeb 0xa2 0xd2 0x97 0x96 0xd9 0xc7 0xb5 0xa7 0xea 0x9d
32:0x15 0x0e 0x14 0x17 0x31 0x0b 0xa0 0xdb 0x07 0x04 0x13 0x0f 0x1d 0x4f 0xle 0x47
```

The page table register has the value 0. For the following virtual addresses, say whether 1) it is a *valid* virtual address, 2) it will cause a *top fault*, i.e., a fault in the top-level page table, 3) it will cause a *second fault*, i.e., a fault in a second-level page table, or 4) you *can't tell* because the memory dump doesn't provide enough information. For valid addresses, write the corresponding physical address (in hex).

	Top Fault/Second Fault/Valid/Can't Tell	Physical Address (in hex)
virtual address 0x143:	0x143: '1' maps to 1st PTE in top-level page table = 0xb4. Lowest bit is unset => invalid => top fault.	
virtual address 0x70c:	0x70c: '7' maps to 7th PTE in top-level page table = 0x03 = 1|1. Lowest bit is set => valid second level page table at frame 1. Since each frame is 16 bytes, this page table starts at address 16. '0' maps to 0th PTE = 0xb2. Lowest bit is unset => invalid => second fault.	
virtual address 0xf43:	0xf43: 'f' maps to 15th PTE in top-level page table = 0x05 = 10|1. Lowest bit is set => valid second level page table at frame 2. Since each frame is 16 bytes, this page table starts at address 32. '4' maps to 4th PTE = 0x31 = 11000|1. Lowest bit is set => valid. frame number = 11000, paddr = 11000|0011 = 0x183.	
virtual address 0x9a8:	0x9a8: '9' maps to 9th PTE in top-level page table = 0x9 = 100|1. Lowest bit is set => valid second level page table at frame 4. Frame 4 is not shown. Can't tell.	

Question 7. RAID Regained [8 MARKS]

You have a RAID-4 device (parity-based RAID + a single parity disk) with 5 disks and a 4KB chunk size, as shown below:

```
Disk-0    Disk-1    Disk-2    Disk-3    Disk-4
block0    block1    block2    block3    parity(0..3)
block4    block5    block6    block7    parity(4..7)
```

You are not sure that writes to the RAID device are working correctly. You write the following skeleton code for writing to the RAID device. Fill in the rest of the code in the empty boxes to ensure that writes work correctly.

```
/*
 * This write() routine takes a logical block number (block), as shown above,
 * and writes 4KB (data) to the device.
 *
 * It uses the underlying primitives:
 * read(int disk, int block_nr, char *data)
 * write(int disk, int block_nr, char *data)
 * xor(char *d1, char *d2, char *d3) ;; xor d1 and d2, place result in d3
 */

void write(int block, char *data)
{
  char buf[4096];
  char parity[4096];
  int disk = [block % 4];
  int phy_block_nr = [block / 4];

  [read](disk, phy_block_nr, [buf]);
  read(4, phy_block_nr, parity);
  xor([parity], [buf], [parity]);
  xor(parity, data, parity);
  [write](disk, phy_block_nr, [data]);
  write(4, phy_block_nr, parity);
}
```

Question 8. Inode Lost [8 MARKS]

In this question, we consider a variant of the Unix file system, which instead of using inodes, instead stores relevant information about a file in the directory entry for that file, so no inode table exists in the file system. We call our new system the Inode-Lost File System or ILFS.

Part (a) [2 MARKS] Which of the following are found in a standard Unix inode? Circle all that apply:

1) Direct pointers to data blocks
2) The name of the file
3) Some statistics about when the file has been accessed, updated, etc.
4) The inode number of the file's parent directory
5) The current position of the file pointer

Part (b) [2 MARKS] In a standard Unix file system, what is the minimum number of disk reads it will take to read a single block from the file `/this/path/is/tooshort` from disk? Other than the superblock, you should assume nothing is cached, i.e., everything starts on disk. Indicate the number of reads of each type of block (e.g., # of directory blocks, etc.).

4 inode blocks, 3 directory blocks, 1 data block

Part (c) [2 MARKS] Now compare this to the minimum number of reads it would take to read a single block from the same file `/this/path/is/tooshort` in ILFS. Again, other than the superblock, assume nothing is cached, i.e., everything starts on disk. Indicate the number of reads of each type of block.

3 directory blocks, 1 data block

Part (d) [2 MARKS] Why do you think ILFS does not support hard links?

The inode allows the Unix file system to keep a reference count of the number of directory entries that point to the inode. This reference count is used to ensure that file contents are deleted only when the last name of the file is deleted. Without the inode, there is no area in which ILFS can keep the reference count, so it cannot support hard links.

Part (e) [2 MARKS] Say the contents of a directory are corrupted due to some errors on the disk. In which file system, ILFS or the Unix file system, would it be *easier* to recover the *contents* of the files in that directory? Why?

It would be easier in the Unix file system because the inodes are located in a well-known location on disk, and each inode helps locate the contents of a unique file or directory. In the Unix file system, a file system recovery program could traverse the file system and find out which inodes are not referenced by any directories. These are the lost files. In ILFS, the directory is corrupted, so we wouldn't know where to start looking for the contents of files.

Question 9. 2^{Files} [8 MARKS]

In a Unix-like file system called 4FS, the block size is 4K, and the block numbers are stored as 4 byte integers in the inode and in the indirect blocks.

Part (a) [2 MARKS] With three levels of indirect blocks, what is the maximum size of a file that can be supported in 4FS. Circle the closest answer below. Show your calculations.

1) 1 GB (2^{30} bytes) 2) 4 GB 3) 1 TB (2^{40} bytes) 4) 4 TB 5) 16 TB

Each block contains 2^10 pointers (4KB/4B). The third-level block allows accessing (2^10)^3 = 2^30 blocks. Each block is 2^12 bytes. So we can access 2^42 bytes = 4TB.

Part (b) [2 MARKS] Suggest a way by which you can increase the maximum file size without changing the number of levels of indirect blocks.

increase block size, e.g., 8KB, which increases number of pointers and block size.

Part (c) [2 MARKS] Now let's go back to the original 4FS file system. Say we store block numbers as 8 byte integers. Would that increase or decrease the maximum file size? Show your calculation.

Decreases maximum file size. Each block will contain 2^9 pointers. So maximum file size is 2^9^3 * 2^12 = 2^39.

Part (d) [2 MARKS] Now let's go back to the original 4FS file system. Say we use four levels of indirect blocks. What is the maximum size of a file that can be supported. Circle the closest answer below. Show your calculations.

1) 1 TB (2^{40} bytes) 2) 4 TB 3) 16 TB 4) 1 PB (2^{50} bytes) 5) 4 PB

This is a little tricky. Four levels of indirect pointers allows 2^40 blocks but the block numbers are stored in 4 bytes, so we cannot address more than 2^32 blocks. Hence, the maximum file size would be limited to 2^32 * block_size = 2^32 * 2^12 = 2^44 = 16TB. Using four levels of indirect pointers fully requires using 8 byte block numbers.

Question 10. The Lies That Editors Tell [14 MARKS]

A modern, text editor claims that it updates files atomically, so that on a system crash, the file contents will either be the old value of the file, or the new value of the file, but not some garbled combination of the old and the new value.

You look at the following editor code for updating a File `f`. Note that `f.new` is a file with the string `''.new''` appended to the name of File `f`. It looks like the editor is using redo recovery. We will ignore any errors returned by these calls.

```
   update_file(File f)
   {
       char buffer[file_size];
1:     read(f, buffer);          // read File ''f'' into a buffer in memory
2:     update_buf(buffer);       // update buffer in memory
3:     write(f.new, buffer);     // write buffer into new File ''f.new''
4:     create(f.done);           // create a new file ''f.done''
5:     copy(f.new, f);           // copy the new version to File ''f''
6:     remove(f.done);           // remove the ''f.done'' file if it exists
7:     remove(f.new);            // remove the ''f.new'' file if it exists
   }
```

Part (a) [6 MARKS] When the text editor opens File `f` after a crash, it needs to run recovery code to ensure that the file is updated atomically. Unfortunately, there is no such code implemented in the editor. Please implement it. You may assume that a function called `exists(f)` is defined (in the editor code) that returns TRUE if File `f` exists in the file system. Hint: Be as brief as possible. It may help to read the question on the next page before writing your answer.

```
   recover_file(File f)
   {
        if (exists(f.done)) { /* do every thing in update_file after Line 5 */
           copy(f.new, f);
           remove(f.done);
        }
        remove(f.new); /* remove f.new if it exists */




   }
```

This question continues
on the following page.

Question 10. The Lies That Disks and Their Drivers Tell (CONTINUED)

For your convenience, we have listed the `update_file` code again.

```
   update_file(File f)
   {
       char buffer[file_size];
1:     read(f, buffer);          // read File ''f'' into a buffer in memory
2:     update_buf(buffer);       // update buffer in memory
3:     write(f.new, buffer);     // write buffer into new File ''f.new''
4:     create(f.done);           // create a new file ''f.done''
5:     copy(f.new, f);           // copy the new version to File ''f''
6:     remove(f.done);           // remove the ''f.done'' file if it exists
7:     remove(f.new);            // remove the ''f.new'' file if it exists
   }
```

Part (b) [8 MARKS] The code above appears to work but it doesn't have any disk flush commands. Recall that modern disks lie to improve performance. They use a DRAM disk buffer, and when a write is issued, they simply copy the data to the disk buffer, and claim the write is completed. Then, they copy the data in the disk buffer to the disk platters asynchronously (sometime later). The problem is that on a crash, the data in the disk buffer that has not been copied to the disk platters can be lost.

Fortunately, disks provide a flush command that forces all data in the disk buffer to be written durably to the disk platters. In the `update_file` code, *under* which lines would you place the disk flush command to ensure that a file is updated atomically? For example, if you say "Line 1", then you would add a flush between Lines 1 and 2 in the code above. Write the line numbers in the table below. Since disk flushes are expensive, use the minimum number of flush commands that ensure a correct atomic update. With each line, state one of the two problems that could happen if the flush was not issued at that point: 1) *before crash*: File `f` is written partially before a crash occurs and no recovery is performed, or 2) *after recovery*: File `f` is written partially when recovery is performed (after the crash). Hint: consider how your recovery code works.

Line Number	Before crash/After recovery
3	after recovery
4	before crash
5	before crash
6	after recovery

3: If f.new is not fully on disk but f.done is created on disk in Line 5, then recovery will copy a garbage version of f.new to f.

4: If f.done is not on disk but the copy operation in Line 6 has partially succeeded, then we will not perform the copy operation during recovery, and f could have partial contents of f.new.

5: If the copy operation is not complete but f.done is removed in Line 7, then similar to the above reasoning, we will not perform the copy operation during recovery, and f could have partial contents of f.new.

6: If f.done is not removed from disk, but f.new is removed in Line 8, then recovery is not directly affected for a single write to f. However, consider a second update to file f (no crash yet). Say a crash occurs after a partial write to f.new in Line 4. On recovery, since f.done exists, we would copy the partially written f.new to f.

Bonus. [1 MARK]

Let us know what you liked or disliked about the course. Any answer, other than no answer, will get a bonus mark.

[Use the space below for rough work.]