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: _____

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 9 questions on 16 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.

MARKING GUIDE

Q1:	_____	(20)
Q2:	_____	(15)
Q3:	_____	(10)
Q4:	_____	(10)
Q5:	_____	(10)
Q6:	_____	(10)
Q7:	_____	(8)
Q8:	_____	(7)
Q9:	_____	(10)
TOTAL:	_____	(100)

Question 1. Short Answers [20 MARKS]

Part (a) [4 MARKS] **Threads:** Consider the user-level threads system that you implemented in the OS labs. Which of the following operations require support from the operating system (e.g., require issuing system calls) in their implementation? For those that do, write Yes, and provide a reason. Do **not** write Yes, if an operation does not directly issue system calls, but simply uses another operation that does.

Operation	Yes	System Call Explanation
Thread switching	no:	getcontext and setcontext are not system calls
Creating a thread	yes:	need to allocate stack dynamically, requiring sbrk
Disabling interrupts	yes:	disables Unix signals, requires system calls
Enabling interrupts	yes:	enables Unix signals, requires system calls
Thread exits itself	no:	disable/enables interrupts, switches thread
Thread preemption	yes:	requires signal support from OS
Thread sleep	no:	disable/enables interrupts, switches thread
Releasing a mutex lock	no:	disable/enables interrupts

1/2 mark for each yes, 1/2 mark for correct explanation, no explanation needed for 'no' answers

Part (b) [6 MARKS] **File Systems:** A program issues the following system call on a Unix file system:

```
ret = mkdir("/a/b", 755);
```

This system call creates directory "b" with Unix permission 755. Assume that the directory "a" exists and directory "b" is created successfully as a result of this call. Assume that the super block has already been read at boot time, but otherwise the buffer cache is empty, and all directories use one data block. Also, assume that the file system does not use journaling.

Show the types of blocks that are read and written when the system call is performed. Add a brief explanation (e.g., read inode of "a") for why the block is read or written beside each block type.

Read	Block Type	Explanation
1	inode block	read inode of /
2	dir block	find inode of a
3	inode block	read inode of a
4	dir block	check if b exists in a
5	block bitmap	find empty block for b
6	inode bitmap	find empty inode for b
7	inode block	read inode of b *
8		
9		

Write	Block Type	Explanation
1	block bitmap	allocate block for b
2	inode bitmap	allocate inode for b
3	dir data block	write data for b
4	inode block	write inode for b
5	dir data block	update data for a*
6	inode block	update inode of a**
7		
8		
9		

* this is to ensure the rest of the inodes in inode block are not overwritten in step 4 on the right side.

*create b dir in a
**e.g., update time stamp

This question continues on the following page.

Part (c) [4 MARKS] **Scheduling:** If a thread is CPU-bound, would it make sense to give it higher priority for disk I/O than an I/O bound thread?

Yes, for the same reason that I/O-bound threads should get higher priority for the CPU. If a CPU-bound thread must wait for I/O behind I/O-bound threads, we could end up in a situation where all of the threads are waiting for I/O and the CPU is idle. Better to finish the I/O for the CPU-bound threads as quickly as possible so they can get keep the CPU busy; this increases our chances of keeping all of the system components busy at all times.

Part (d) [6 MARKS] **Virtual Memory:** Answer the following for a demand paging system. Assume that the physical memory size is fixed.

a) Can doubling the page size reduce the number of page faults? If so, describe how. If not, describe why.

Doubling the page size behaves similar to prefetching pages. If a program accesses contiguous pages (e.g., pairs of pages), then increasing the page size will reduce page faults.

b) Can halving the page size reduce the number of page faults? If so, describe how. If not, describe why.

Paging systems cause internal fragmentation. Reducing the page size reduces internal fragmentation, potentially making more physical memory available (e.g., if a program was using a small part of a page, the smaller page size would still suffice, but the rest of the memory would be available to others). This will lead to fewer page faults and demand paging requests.

c) Suppose a friend told you: If a system has lots of runnable threads, it can use them to hide the cost of page faults. Explain whether your friend is right, wrong, or both.

Both. If the working sets of all the runnable threads fit in memory, then while one thread is waiting for a page fault another thread can execute, effectively hiding the cost of page faults. However, if the working sets do not fit in memory, then the system will quickly end up in a state where all threads are waiting for page faults; as soon as a thread resumes after a page fault, it will touch another page that is not in memory and generate another page fault to wait for. The more runnable threads there are, the worst things will get.

Question 2. Fill in the Blanks [15 MARKS]

Part (a) [3 MARKS] **TLB:** Consider a system with a software-managed tagged TLB. The tag uses 6 bits. The OS supports running a maximum of 1024 processes at a time. The processor provides the following options, in increasing cost, for programming the TLB: 1) invalidate a specific TLB entry, 2) invalidate the entries with a given tag, 3) flush the TLB (invalidate all entries). For each of the following events, indicate whether the OS software should take no action with respect to the TLB (**N**), invalidate an entry in the TLB (**I**), invalidate all entries associated with a tag (**T**) or flush the TLB (**F**). Use the most efficient option available. You may use the space on the right to explain your answer (optional).

T	Context switch	note that multiple processes can map to the same tag, so we will need to invalidate all entries if another process with same tag ran the last time
I	Page eviction	update mapping to invalid page
N	Return from a recursive function call	no change to address space
T	Process exit	invalidate all entries associated with this process
I	Copy-on-write fault	update mapping to writeable page
N	Fork system call, without copy-on-write	no change to address space of current process

Part (b) [4 MARKS] **OS Concepts:** Hardware support for operating systems includes privileged instructions (**P**), MMU support for virtual memory (**M**), interrupts (**I**), and the trap instruction (**T**). Indicate how the OS takes advantage of (one or more of) this hardware to ensure or support the following:

M	a program doesn't modify the memory of another	
M, P	a program doesn't access hardware directly	The P is for i/o operations
I	a program doesn't run forever	
M	a program doesn't jump to an arbitrary OS address	
M, P	a program doesn't modify its virtual memory mapping	The P is for MMU operations
T	a program updates a block on disk	Need system call
T	a program sends a message to another process	Need system call
T	a program waits to receive a message from another process	Need system call

This question continues on the following page.

Part (c) [5 MARKS] **Disk scheduling:** Your OS *only* allows reading and writing whole files (similar to the web server implementation in your lab that read entire files). When files are accessed, the OS issues disk requests for all the blocks of the file. These requests can be scheduled using various disk scheduling algorithms.

We have discussed the First-Come-First-Served (FCFS), Shortest-Seek-First (SSF) and the Elevator scheduling algorithms in class. In addition, let's consider two other scheduling algorithms. The Shortest-File-First (SFF) is a non-preemptive algorithm that schedules all block requests for a file at a time, prioritizing the shortest file first. The Round-Robin (RR) algorithm is a preemptive algorithm that schedules requests one block at a time for all files that are being accessed concurrently.

These disk scheduling algorithms have various desirable properties shown below.

Exploit Locality: whether the algorithm exploits request locality.

Bounded Waiting: whether the algorithm bounds waiting time for all block requests it has received.

Fairness: whether the algorithm provides fairness, meaning that each block request is treated equally, regardless of what position on the disk the request is referencing.

Fill in the cells of the table below, using a check mark to indicate that the algorithm has the desired property. If the algorithm does not have the desired property, leave the table cell blank.

	Exploits locality	**Bounded waiting**	**Fairness**
FCFS		X	X
SSF	X		
Elevator	X	X	
SFF	X*		
RR		X	X

* assumes that small files are placed mostly sequentially

Part (d) [3 MARKS] **Page Replacement:** You are familiar with the FIFO, Clock and LRU page replacement algorithms. The MRU page replacement algorithm replaces the *most* recently accessed page. Simply state **True** or **False** for the following statements regarding these four page replacement algorithms. You may use the space below each statement to explain your answer (optional).

	True or False
Clock and FIFO are easier to implement that MRU.	True
A program accesses an array that is slightly larger than physical memory size repeatedly. MRU is the best choice for this program. MRU page will be accessed furthest in the future.	True
A program accesses a large array just once. LRU is the best choice for this program. MRU is better because the data should be replaced right away since it will not be used in the future.	False
A program accesses the elements of a large array randomly. All the four page replacement algorithms will work equally well for this program.	True
A program makes a deeply recursive function call. FIFO is the best choice for this program. FIFO and LRU will make the same page replacement decisions, but FIFO is more efficient.	True
All the four page replacement algorithm benefit from hardware support for tracking page accesses. FIFO doesn't need reference bits.	False

Question 3. OS Design [10 MARKS]

In this question, you will add signals to the threads library that you implemented in the OS labs.

Recall that Unix signals allow the OS to notify a process (or thread) of some event of interest. For instance, the kernel delivers the SIGTERM signal to a process when a human user types `Ctrl-C`. If a process doesn't handle this signal, the process is killed by the kernel. Alternatively, the process can choose to ignore the signal, or perform some action (e.g., print "I am unkillable") upon receiving this signal.

A process registers a signal handler function that is invoked when a signal is to be delivered to the process. The signal can be delivered either by the kernel (e.g., when a process accesses bad memory), or at the request of another process (e.g., bash shell sends `Ctrl-C` to child process). Note that signal delivery interrupts the process. For example, in your labs, the SIGALRM signal was used to implement preemptive threading.

Given this information, describe how you would implement signal support in your threads library. In particular, your library should allow a thread to signal another thread. Make sure to describe the minimal, but complete, set of changes, including to data structures, interfaces, etc., that you would make to your current threads implementation. Write your answer as a set of bullet points, preferably with a short caption for each point. Be as precise as possible. Hint: think about the entire lifetime of the signaling mechanism.

1. Add an interface function (e.g., thread_register_signal) that allows registering (and deregistering) a signal function.

2. Add an interface function (e.g., thread_signal) to signal another thread.

3. Your implementation of thread_exit already implements exiting another thread. You can simply reuse that implementation for implementing signals (this is what we were looking for in your answer). A call to thread_signal should be very similar to thread_exit, except that it should set a "signaled bit" (rather than the "exited bit") in the target thread.

4. When a signaled thread is run the next time and the signal function has been registered then the signal function should be run. (If the signal function is not registered, the signaled thread should exit.) The signaled bit should be unset.

5. When the signal function returns, the original code of the thread should be run again. To detect function return, a signal stub (similar to the thread_stub function) should be used to run the signal function.

Question 4. Red-Blue Race [10 MARKS]

Car owners are trying to decide whether red cars or blue cars are faster. They device an experiment in which all the red cars line up behind each other in a red line, and similarly all the blue cars line up behind each other in a blue line. Your job is to ensure that a red car and a blue car at the head of the two lines start the race at about the same time, i.e., either car should not start the race much before the other. You also need to ensure that the experiment makes progress as long as cars are present in both lines.

Part (a) [2 MARKS] You ask your friend John for help, and the pseudocode shown below is the best that he can come up with. A red car, upon arriving in the red line, invokes the function `red_lineup()`. When the function returns, the car starts racing. The blue solution is symmetric with the red solution (swap all occurrences of red and blue), and is not shown. Assume that `signal()` wakes up threads in the order they issued the `wait()`.

```
red_lineup()
{
        lock(lock);
        red_waiting++;
        while (blue_waiting == 0) {
           wait(red, lock);
        }
        signal(blue, lock);
        red_waiting--;
        unlock(lock);
}
```

You think about this solution and realize that it doesn't work correctly. Explain why.

red arrives and waits
blue arrives and signals, reduces blue_waiting to 0
red still waits because of "while"

This question continues on the following page.

Part (b) [3 MARKS] Now you ask your friend Mary for help, and the pseudocode shown below is the best that she can come up with.

```
red_lineup()
{
        lock(lock);
        red_waiting++;
        if (blue_waiting == 0) {
           wait(red, lock);
        } else {
           signal(blue, lock);
        }
        red_waiting--;
        unlock(lock);
}
```

You think about this solution and realize that it still doesn't work correctly. Explain why.

red arrives and waits
blue arrives, signals, exits
second blue arrives, should wait, but continues because red_waiting is still 1

Part (c) [5 MARKS] You think hard about why John and Mary's solutions didn't work and realize that a small change will make the solution work. Show that solution below.

the problem is that in part b), when the first blue signals, it should have reduced red_waiting as well.

```
red()
{
    lock(lock);
    red_waiting++;
    if (blue_waiting == 0) {
        wait(red, lock);
    } else {
        red_waiting--;
        blue_waiting--;
        signal(blue, lock);
    }
    unlock(lock);
}
```

Question 5. Deadlocks [10 MARKS]

Part (a) [5 MARKS] Alice, Bob, and Carol go to a Chinese restaurant at a busy time of the day. The waiter apologetically explains that the restaurant can provide only two pairs of chopsticks (for a total of four chopsticks) to be shared among the three people. Alice proposes that all four chopsticks be placed in an empty glass at the center of the table and that each diner should obey the following protocol:

```
while (!had_enough_to_eat()) {
    acquire_one_chopstick(); /* May block. */
    acquire_one_chopstick(); /* May block. */
    eat();
    release_one_chopstick(); /* Does not block. */
    release_one_chopstick(); /* Does not block. */
}
```

Can this dining plan lead to deadlock? Explain your answer.

Deadlock **cannot** occur because there are enough chopsticks to guarantee that at least one of the 3 diners will be able to get the 2 chopsticks that he/she needs. Once that diner finishes, he/she will release their chopsticks for someone else to use, so eventually everyone finishes.

Part (b) [5 MARKS] Suppose now that instead of three diners there will be an arbitrary number, `D`. Furthermore, each diner may require a different number of chopsticks to eat. For example, it is possible that one of the diners is an octopus, who for some reason refuses to begin eating before acquiring eight chopsticks. The second parameter of this scenario is `C`, the number of chopsticks that would simultaneously satisfy the needs of all diners at the table. For example, Alice, Bob, Carol, and one octopus would result in `C = 14`. Each diner's eating protocol is shown:

```
int s;
int num_sticks = my_chopstick_requirement();
while (!had_enough_to_eat()) {
    for (s = 0; s < num_sticks; ++s) {
        acquire_one_chopstick(); /* May block. */
    }
    eat();
    for (s = 0; s < num_sticks; ++s) {
        release_one_chopstick(); /* Does not block. */
    }
}
```

What is the smallest number of chopsticks (in terms of D and C) needed to ensure that deadlock cannot occur? Explain your answer.

C – D + 1. This guarantees that every diner can get all but one of the chopsticks it needs (C – D), with one additional chopstick to guarantee that at least one diner gets all of the chopsticks it needs.

Question 6. **Paging** [10 MARKS]

You have been hired by the MobARM Corporation to develop the virtual memory architecture on their next generation mobile processor. The virtual memory design has the following parameters:

- Virtual addresses are 54 bits.
- The page size is 64K byte.
- The architecture allows a maximum of 128 Terabytes (TB) of physical memory.
- The first- and second-level page tables are stored in physical memory.
- All page tables can start only on a page boundary.
- Each second-level page table fits exactly in a single page frame, but the first-level page table may be larger than a page frame.
- There are only valid bits and no other extra permission, or dirty bits.

Draw a figure showing how a virtual address gets mapped into a physical address. Your design should be as efficient as possible. You should list how the various fields of the virtual and the physical address are interpreted, including the size of each field (in bits). Show the format of the page table entry, the total number of entries each table holds, and the total size of each table (in bytes).

page size 64KB => offset is 16 bits.

max physical memory = 128 TB => 47 bits.

number of bits in the frame = 47 - 16 = 31

with 1 valid bit, the page table entry can nicely fit in 4 bytes (31 + 1) bits.

second level page table is 64KB (one page, 16 bits), so it can have 2^14 entries (each entry is 4 bytes).

virtual address size = 54 bits

so top level page table should be able to address (54 - 14 - 16) = 24 bits

top-level page table size = 2^24 * 4 (bytes per entry) = 2^26 = 64MB.

Question 7. Sparse Files [8 MARKS]

A file system stores files that are mostly empty as *sparse files* on disk. For example, if the length of a file is 1 MB, but the file only contains data in its first and last blocks, then the file system only stores these two data blocks. When an application reads a sparse file, the file system converts the empty blocks into ”real” blocks filled with zero bytes, without the application being aware of this conversion. Answer the following questions.

Part (a) [1 MARK] Briefly explain how sparse files might be implemented in a Unix file system.

An easy way to implement sparse files is to ensure that a block pointer is null/zero for an empty block (block is not allocated).

Part (b) [3 MARKS] Say you wanted to upload the sparse files on your local disk to a file hosting service (such as Dropbox), and the file service supports sparse files as well. The problem will applications being unaware of sparse files is that your file synchronization client needs to read the entire sparse file, to determine whether any block contains all zeroes, before sending the sparse file to the file service. Show the function prototype of a new system call that you would implement that will not require reading the entire sparse file but just the blocks that are non-empty. How is the system call used? Are there any restrictions or limitations on the usage of this system call?

```
char buffer[4096];
int offset, next_offset;
loop:
  error = read_sparse(fd, offset, buffer, &next_offset);
  offset = next_offset;
```

In this interface, reads must be at block granularity. offset is the block or byte offset in the file at which to read a block. If the block at that offset is empty, the call should return an error. next_offset returns the next offset in the file at which a block is allocated. The main limitation of this interface is that the application must know about the file system block size (e.g., 4096 bytes).

This question continues on the following page.

Part (c) [2 MARKS] Briefly describe how a file system would implement your new system call.

read_sparse would be implemented similar to read. the main difference is that the next_offset would need to be returned. This will require traversing the file metadata blocks, e.g., inode, indirect block, etc., until the next allocated block is found (non-null block pointer).

Part (d) [2 MARKS] State two reasons why your file synchronization client will be more efficient when it uses your new system call.

1) The client will issue fewer read_sparse system calls than read system calls
2) The client does not need to parse a block and see if it has all zeroes to detect empty blocks.

Question 8. File System Extents [7 MARKS]

In an extent-based file system, storage is allocated in variable-sized, contiguous disk sectors called *extents*. Assume that extents are sized in multiples of blocks (e.g., 1 block, 13 blocks, etc.). In a Unix-style file system, suppose the only change you make to the on-disk format is that all block pointers are replaced with extent pointers. Each extent pointer is 8 bytes long, and contains a block location on disk, and the length of the extent in blocks. Answer the following questions. If you make any assumption, state them.

Part (a) [3 MARKS] State three advantages of using an extent-based file system design compared to a block based design.

1) a large file that is allocated sequentially will need to store data for one extent only, so there is lower metadata storage overhead.
2) there will be fewer accesses for metadata blocks for the same reason as above, improving performance.
3) files could be preallocated large extents, allowing for file growth without fragmentation.

Part (b) [3 MARKS] How would you change the block-based Unix file system code so that the extent-based file system will generally perform much better than the block-based design. Clearly explain your answer.

The file system could preallocate large extents when files are created, allowing for file growth. This may however lead to internal fragmentation.

Another option would be for the file system to periodically copy one or more small logically contiguous extents into a large single extent.

Part (c) [1 MARK] Give two examples where the extent-based file system design would be less efficient than a block-based design. Explain why.

The inode size is much bigger (close to twice the size), leading to performance overhead, because more inode blocks have to be read (to read the same number of files).
If each extent is a single block, the file metadata is doubled (8 bytes per extent, versus 4 bytes per block), leading to more storage/performance overhead, and smaller maximum file size.

Question 9. File System Consistency [10 MARKS]

Part (a) [4 MARKS] **Performance Optimization:** Suppose that you decide to improve the performance of the Berkeley fast file system (FFS) by removing all synchronous writes to disk. In your modified file system, all writes to disk are delayed 30 seconds (or until the kernel evicts blocks from the buffer cache). This allows two successive writes to the same inode, directory block, or other metadata structures to only require a single write to disk. Moreover, your kernel uses a disk scheduler that orders the writes to minimize disk seeks. Describe a way in which, after a power failure, a directory block may contain, instead of directory entries, data blocks from an existing user file. Make sure to show the sequence of system calls made before the power failure that cause this problem.

1. delete file f

2. create new directory d

say the kernel reuses a block that used to belong to f to store the directory data for d

3. crash

if all the metadata indicating the f has been deleted (inode bitmap block for f, inode of f, block containing directory entry for f, block bitmap for all blocks of f that are deallocated) did not reach disk before the crash, while all the metadata blocks indicating that d is created (inode bitmap block for d, inode of d, block containing directory entry for d) did reach disk before crash, then both file f and directory d will point to the same blocks

This question continues
on the following page.

Part (b) [6 MARKS] **Journaling:** Recall that journaling file systems are designed to speed up crash recovery. Explain clearly how a journaling file system would avoid or recover from the inconsistency caused by the scenario that you have outlined above. To do so, make sure to describe the types of blocks that are written, allocated or deallocated, and show the precise order of operations using a simple diagram. You do not need to describe any blocks in your scenario that are not relevant to crash recovery. Note that your journaling file system only journals metadata blocks. It may help to write your answer as a set of bullet points.

The reason for the inconsistency is that the file f's deletion does not reach disk before directory d's creation reached disk. We can avoid this inconsistency by using journaling. Two options:

1. Journal "delete f, mkdir d" in one transaction, where all the metadata updates for each operation (described above) are logged to the journal. In this case, write-ahead logging will ensure that either both the operations are performed, or neither will be performed, avoiding any inconsistency.

2. Journal "delete f" in one transaction, "mkdir d" in next transaction. In this case, write-ahead logging will ensure one of the following: 1) none of them are performed, 2) "delete f" is performed, but "mkdir d" is not, 3) both operations are performed. In all cases, there is no inconsistency.

[Use the space below for rough work.]