# 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 8 questions on 21 pages (including this page). The value of each part of each question is indicated. The total value of all questions is 125.

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: | _____ | (16) |
| Q2: | _____ | (13) |
| Q3: | _____ | (20) |
| Q4: | _____ | (16) |
| Q5: | _____ | (18) |
| Q6: | _____ | (18) |
| Q7: | _____ | (14) |
| Q8: | _____ | (10) |
| BONUS MARKS: | _____ | 2 |
| TOTAL: | _____ | (125) |

## Question 1. Short Questions [16 MARKS]

**Part (a)** [2 MARKS] After the `fork()` system call, either the parent or the child process may run first. If you change this system call to always run the parent first, explain whether it could be beneficial in any way or not.

It would not be beneficial since the application does not control scheduling. Thus, we could not ensure that any user code in the parent would definitely run before the child was scheduled to run again.

**Part (b)** [2 MARKS] You decide to use Unix-style feedback scheduling for scheduling the network packets of different sockets. To do so, you measure network usage by counting the number of packets sent or received on each socket. Describe two benefits of using this scheme.

The benefits would be similar to feedback-based cpu scheduling. Socket connections that use the network heavily will share the network fairly, while socket connections that send packets occasionally will get higher priority, thus speeding up interactive sessions, e.g., remote ssh.

**Part (c)** [4 MARKS] Your friend tells you that using locks, semaphores, condition variables are a thing of the past! They are not required to correctly implement multi-threaded application code if the application is run using a non-preemptive, user-level thread scheduler (e.g., think about Lab 2). Is your friend correct or not (or perhaps both!)? Clearly explain your answer.

Your friend is correct when the user-level thread scheduler uses a single kernel thread. In this case, the multi-threaded code can fully control scheduling and thus it can avoid yielding within a critical section. Similarly, for synchronization, a thread does not need to worry about the lost wakeup problem, since it is the only one running before it goes to sleep.

Your friend is incorrect when the thread scheduler uses multiple kernel threads. In this case, the user threads can run concurrently or in parallel, and so mutual exclusion and synchronization operations are required.

**Part (d)** [2 MARKS] In a multiprocessor operating system, describe one benefit of using per-core data structures (e.g., a per-core run queue), and one benefit of using a global data structure (e.g., a global run queue).

per-core data structures reduce contention due to synchronization operations (e.g., atomic operations), thus enabling better scaling of operations (e.g., thread scheduling) with increasing number of cores.

per-core data structures can lead to load imbalance: some cores stay idle while others are overloaded. global data structures enable better load balancing across cores.

**Part (e)** [2 MARKS] Does a traditional Unix file system use access control lists or capabilities or both? Briefly explain your answer.

In a Unix file system, ACLs are used to store user permissions with each file or directory. These permissions are checked when a user opens a file to access the file. After that, Unix uses a file descriptor, which behaves like a capability, to read and write the file more efficiently (since each read or write does not require the permission check).

**Part (f)** [2 MARKS] In the map-reduce framework, the map and reduce functions need to be deterministic. What are deterministic functions and why are they needed?

Deterministic functions return the same output for the same input. In map-reduce they are needed to handle worker failure. When a function executing a deterministic map or reduce function fails, another worker can be restarted to perform the same function to produce the same correct output.

**Part (g)** [2 MARKS] Which ECE344 lab did you like the most and why? If you had to replace one of the labs, which one would you replace and why?

## Question 2. Synchronization [13 MARKS]

The Niagara Rainbow bridge is used to cross the Canada-US border. Unfortunately, it is undergoing repairs and only one lane is open to 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.

```
1
2   sem fair = 1;
3
4   CA_car(int num)                          US_car(int num)
5   {                                        {
6         sem_down(fair);                     sem_down(fair);
7         print(CAnum);                       print(USnum);
8
9         lock(ca_lock);                           lock(us_lock);
10
11        if (CA == 0) {                           if (US == 0) {
12
13            lock(bridge_lock);                       lock(bridge_lock);
14
15        }                                        }
16        CA++;                                    US++;
17
18        unlock(ca_lock);                         unlock(us_lock);
19        sem_up(fair);                            sem_up(fair);
20        // multiple waiting Canadian            // multiple waiting US
21        // cars can cross the bridge            // cars can cross the bridge
22
23        print(CAnum crosses);               print(USnum crosses);
24
25        lock(ca_lock);                           lock(us_lock);
26
27        CA--;                                    US--;
28        if (CA == 0) {                           if (US == 0) {
29
30            unlock(bridge_lock);                     unlock(bridge_lock);
31
32        }                                        }
33
34        unlock(ca_lock);                         unlock(us_lock);
35
36  }                                        }
```

**Part (a)** [10 MARKS] The traffic light code shown in the previous page is used to synchronize the cars. Each Canadian car is a separate thread that runs the `CA_car` function, and similarly, each US car runs the `US_car` function. Assume that the `print()` function runs atomically. You find that the code works correctly, so there are no car crashes. However, sometimes all the US cars keep crossing the bridge, while the Canadian cars keep waiting for a long time. You are tired of this behavior and want to fix this problem.

To understand this problem further, you observe the cars as they arrive. Say the cars arrive in the following order, as output by the `print()` function (from left to right):

US1 US2 US3 CA1 CA2 US4 US5 CA3

You find that the US4 car can cross before the CA1 or CA2 cars. To ensure fairness, you would like CA1 and CA2 to cross before US4, and similarly US4 and US5 should cross before CA3.

Modify the code shown in the previous page to ensure such fairness (you do not need to remove any existing lines of code). Make sure that multiple cars (going one way) can cross the bridge together. For example, US1, US2 and US3 (or CA1 and CA2) should be able to be on the bridge at the same time, and can cross in any order.

Hint: You need one semaphore, and just calls to `sem_down()` and `sem_up()` operations to implement this behavior. Show the semaphore operations, and make sure to initialize the semaphore.

See the answer in the previous page. sem_up() needs to be placed after the bridge is locked. For example, if it is placed on line 10 then the ordering we require cannot be guaranteed. It cannot be placed at line 14 or else there will not be enough sem_up() issued. It can be placed after line 15 (e.g., line 17, or 19). It should not be placed after line 19 or else there will be much less concurrency (fewer cars will be able to cross the bridge together).

**Part (b)** [1 MARK] Does your code make any assumptions? If so, state them.

sem_down implements FIFO ordering.

**Part (c)** [2 MARKS] Does your solution increase or decrease the bridge throughput, i.e., the total number of cars crossing the bridge per unit time? Circle your answer and explain why.

Increases throughput | Decreases throughput

Cars going in the same direction will no longer be able to go immediately due to another car arriving from the opposite direction. Thus the total number of cars crossing the bridge at a time will decrease, reducing overall throughput.

## Question 3. **Paging** [20 MARKS]

A processor uses 48 bit virtual addresses. Its memory management unit uses a four-level page table scheme with a page size of 4 KB. The page table entries are 32 bits wide, and all page tables (except the top-level page table) require exactly a single page frame. The memory management unit is designed to be able to address a maximum of 1TB of physical memory. The frame number is stored in the lowest order bits in the page table entry as shown below:

Page table entry:

| Control Bits | Frame number |
|---|---|

**Part (a)** [2 MARKS] How many page table entries are stored at each level of the page table?

Top level:

Second level:

Third level:

Fourth level:

page size: 4KB
PTE size: 32 bits = 4 bytes
# of PTEs per page: 4KB / 4 = 2^10

top level: 2^6 = 64
second level: 2^10 = 1024
third level: 2^10 = 1024
fourth level: 2^10 = 1024
(page): 2^12

6 + 10 + 10 + 10 + 12 = 48 bits in virtual address

**Part (b)** [1 MARK] The processor manufacturer is interested in updating the paging MMU and has hired you to consider alternative paging designs. You know that programs are getting larger and so you propose to change the paging MMU to use a three-level page table scheme with 16 KB pages. You do not make any other changes to the processor or the paging scheme. How many page table entries would be stored at each level of the new page table scheme?

Top level:

Second level:

Third level:

page size: 16KB
PTE size: 32 bits = 4 bytes
# of PTEs per page: 16KB / 4 = 2^12

top level: 2^10 = 1024
second level: 2^12 = 4096
third level: 2^12 = 4096
(page): 2^14

10 + 12 + 12 + 14 = 48 bits in virtual address

**Part (c)** [8 MARKS] The manufacturer is concerned that your three-level scheme may have significant memory overheads. To make a reasoned argument about the benefits or drawbacks of the three-level scheme, you do some back of the envelope calculations of memory overheads of paging for the program stack. Assume that the program stack starts at the top of the 48-bit virtual address space of the program and grows downwards. In the table below, show the memory requirements (in KB) for all the page tables that are needed to access the stack in the four and the three-level paging scheme as the stack grows over time. You may provide your answer as a formula or as a final value. If you make any assumptions, write them down.

| Stack Size | Four-level Page Tables (in KB) | Three-level Page Tables (in KB) |
|---|---|---|
| 4KB | (1 + 1 + 1 + 1) * 4KB = 16 KB | (1 + 1 + 1) * 16KB = 48 KB |
| 16KB | (1 + 1 + 1 + 1) * 4KB = 16 KB | (1 + 1 + 1) * 16KB = 48 KB |
| 4MB | (1 + 1 + 1 + 1) * 4KB = 16 KB | (1 + 1 + 1) * 16KB = 48 KB |
| 64MB | (1 + 1 + 1 + 16) * 4KB = 76 KB | (1 + 1 + 1) * 16KB = 48 KB |

For four-level page table, the third level holds 2^10 pages = 2^10 * 2^12 = 2^22 = 4MB. With 64MB, we need 64/4 = 16 third-level page tables.

For three-level page table, the third level holds 2^12 pages = 2^12 * 2^12 = 2^26 = 64MB.

**Part (d)** [2 MARKS] Based on the above analysis, is there a crossover point (in terms of stack size) below which one scheme requires less memory for the page tables than the other, but above which it requires more memory than the other? If so, what is the crossover point? Write your answer in KB or MB. If you make any assumptions, write them down.

Notice from the answers above that the crossover point occurs between 4MB and 64MB stack size. The actual crosspoint can be calculated as follows: (1 + 1 + 1 + x) * 4KB = 48KB, so x = 9 third-level page tables. Since each third-level page table supports 4MB, the crossover point is 36MB. Below this stack size, the four-level page table design is more efficient. Above this stack size, the three-level page table design is more efficient.

**Part (e)** [2 MARKS] Consider a program that has larger memory requirements over time. Assume that it can be larger because its regions (segments) are larger, or because it has more regions (e.g., mmapped regions), but not both. What would you suggest to the manufacturer in the two cases shown below. Briefly explain your answer.

| | Use Four or Three-Level Page Table |
|---|---|
| Larger regions | Based on part d, use three-level page table. |
| More regions | Based on part d, use four-level page table. |

**Part (f)** [4 MARKS] A partial last level page table, with page table entries on each row, is shown below. The start of the table is shown at the top and memory addresses grow downwards, as shown in the first column.

| Page Offset | | | | |
|---|---|---|---|---|
| 00 | e1 | 7a | d9 | 12 |
| 04 | 00 | 03 | 80 | 00 |
| 08 | 89 | 2d | c6 | d5 |
| 0c | b2 | d3 | 53 | 58 |
| 10 | 8a | be | 9b | 09 |
| 14 | 80 | 05 | 00 | 09 |
| 18 | 22 | 9c | 18 | 67 |
| 1c | b5 | 37 | 23 | 8f |
| 20 | b6 | 94 | c6 | d8 |
| 24 | cc | f7 | 3f | e8 |
| 28 | ab | 13 | 22 | 40 |
| 2c | b3 | 81 | 11 | 00 |
| 30 | 80 | d6 | 7b | e0 |
| 34 | 0f | 33 | b4 | 56 |
| 38 | d9 | 3b | 78 | 98 |
| 3c | 9c | 31 | 76 | b2 |

The MMU will use this table to translate the virtual address 0xf23d0c00974c. What is the physical address corresponding to this virtual address using the four and the three level paging schemes? Show your calculations.

| | Physical Address |
|---|---|
| Four-Level Page Table | 0xcf73fe874c |
| Three-Level Page Table | 0x4b71b557fc |

Four level: page size = 4KB, so 12 bits. so VA = (0xf23d0c) 00(00 0000 1001) (0x74c)
Last-level page table has 10 bits = 0x9
Corresponding page table entry is at page offset 0x24 (0x9 * 4 = 36 = 0x24) = ccf73fe8
Since the hardware support 1TB physical memory (2^40), and frame size = 4KB (2^12), the number of bits for storing frames = 28 (40 - 12). So the frame number is 0xcf73fe8 (the highest 4 bits = 0xc are the control bits).
Physical address = 0xcf73fe874c

Three level: page size = 16KB, so 14 bits. VA = (0xf23d0) 11(00 0000 0000 10)01 (0x74c)
Last-level page table has 12 bits = 0x2
Corresponding page table entry is at page offset 0x8 (0x2* 4 = 0x8) = 892dc6d5
Since the hardware support 1TB physical memory (2^40), and frame size = 16KB (2^14), the number of bits for storing frames = 26 (40 - 14). So the frame number is (01)2dc6d5 (the highest 6 bits are the control bits).
Physical address = 01 0010 1101 1100 0110 1101 0101 01 (0x74c) = 0100 1011 0111 0001 1011 0101 0101 (74c) = 4b71b5574c

**Part (g)** [1 MARK] Based on the page table design, which paging scheme (four or three level) provides more flexibility for the OS to implement virtual memory management? Briefly explain why.

Three level, since it provide more control bits for OS to implement its VM management.

## Question 4. Page Replacement [16 MARKS]

Two processes, P1 and P2 are executing on a system with 8 pages of physical memory. The OS runs the page replacement algorithm on demand, i.e., when a frame needs to be allocated and no free frames are available. At that time, the replacement algorithm uses the global clock algorithm to make one frame available. The clock algorithm tracks the status of pages and frames using the per-process page tables and the coremap shown below.

### Initial State

P1 Page Table

| Page | Valid | Reference | Frame |
|---|---|---|---|
| 0 | 1 | 1 | 5 |
| 1 | 1 | 0 | 6 |
| 2 | 0 | 0 | - |
| 3 | 0 | 0 | - |
| 4 | 1 | 1 | 2 |

P2 Page Table

| Page | Valid | Reference | Frame |
|---|---|---|---|
| 0 | 0 | 0 | - |
| 1 | 1 | 0 | 1 |
| 2 | 1 | 1 | 7 |
| 3 | 1 | 1 | 3 |
| 4 | 1 | 1 | 4 |

Coremap

| Frame | Allocated | ID | Page |
|---|---|---|---|
| 0 | 0 | - | - |
| 1 | 1 | 2 | 1 |
| 2 | 1 | 1 | 4 |
| 3 | 1 | 2 | 3 |
| 4 | 1 | 2 | 4 |
| 5 | 1 | 1 | 0 |
| 6 | 1 | 1 | 1 |
| 7 | 1 | 2 | 2 |

### Step 1

| V | R | F |
|---|---|---|
| | | |
| 1 | 1 | 6 |
| | | |
| | | |
| | | |

| V | R | F |
|---|---|---|
| | | |
| | | |
| | | |
| | | |
| | | |

| A | I | P |
|---|---|---|
| | | |
| | | |
| | | |
| | | |
| | | |
| | | |
| | | |
| | | |

### Step 2

| V | R | F |
|---|---|---|
| | | |
| | | |
| 1 | 1 | 0 |
| | | |
| | | |

| V | R | F |
|---|---|---|
| | | |
| | | |
| | | |
| | | |
| | | |

| A | I | P |
|---|---|---|
| 1 | 1 | 2 |
| | | |
| | | |
| | | |
| | | |
| | | |
| | | |
| | | |

### Step 3

| V | R | F |
|---|---|---|
| | | |
| | | |
| 1 | 0 | 0 |
| 1 | 1 | 1 |
| | | |

| V | R | F |
|---|---|---|
| | | |
| 0 | 0 | - |
| | | |
| | | |
| | | |

| A | I | P |
|---|---|---|
| | | |
| 1 | 1 | 3 |
| | | |
| | | |
| | | |
| | | |
| | | |
| | | |

### Step 4

| V | R | F |
|---|---|---|
| 1 | 0 | 5 |
| 1 | 0 | 6 |
| 0 | 0 | - |
| | | |
| 1 | 0 | 2 |

| V | R | F |
|---|---|---|
| 1 | 1 | 0 |
| | | |
| 1 | 0 | 7 |
| 1 | 0 | 3 |
| 1 | 0 | 4 |

| A | I | P |
|---|---|---|
| 1 | 2 | 0 |
| | | |
| | | |
| | | |
| | | |
| | | |
| | | |
| | | |

**Part (a)** [2 MARKS] Fill in the initial state of the coremap using the page tables for P1 and P2. The Allocated column (already filled), is the allocation status of each page. The ID is the process ID.

**Part (b)** [14 MARKS] Suppose the two processes run and access 4 pages as shown below.

Step 1: P1: Page 1
Step 2: P1: Page 2
Step 3: P1: Page 3

Context switch

Step 4: P2: Page 0

Show the changes to the page tables and the coremap after each step on the previous page. Assume that after each step, the process can access the corresponding page (e.g., P1 can access Page 1 after Step 1). When a page table or coremap entry changes, update the entire entry. Assume that when the clock replacement algorithm runs for the first time, the clock hand is pointing at Frame 0.

## Question 5. TLB and Memory Management [18 MARKS]

Your are designing a high-performance, CPU-bound application called FAST that processes lots of data in (DRAM) memory. When you measure the performance of FAST using a testing framework, you find that FAST behaves erratically, performing well initially but then it becomes slower when it runs for a long time (have we seen this before?). Using profiling tools, you find that the problem is caused by the heap allocator (i.e., `malloc` library).

To understand this problem further, you look at the code of the `malloc` library, and find that it uses linked list allocation as shown below. On each call to `malloc(size)`, the library allocates a chunk of `8 + size` bytes. The 8 bytes are for the header (shown with a dotted pattern). The header tracks whether a chunk is allocated or free, and it has a pointer to the next chunk.

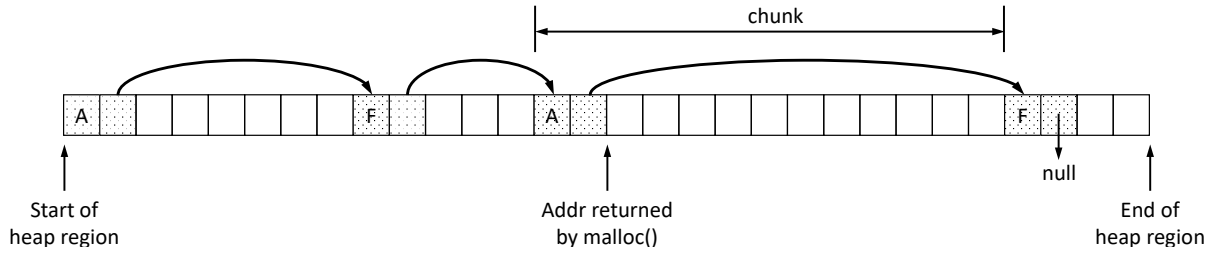



Figure 1: Malloc Linked List Implementation

You instrument FAST with a debugging function of the `malloc` library called `total_heap_size()` that prints the amount of memory currently allocated to the heap region of the program. FAST is implemented as a multi-threaded program. You configure FAST to run with 4 threads on a 4 core UG lab machine. The different threads allocate memory dynamically using `malloc()` from the same heap region of the program.

**Part (a)** [4 MARKS] You find that as long as `total_heap_size()` returns less than roughly 480 KB, FAST works well. Otherwise, the performance of FAST degrades significantly. You suspect the problem is with TLB usage. You look at the processor specifications, and find that each core has its own TLB, and the page size of the paging MMU is 4 KB. Based on your calculations, how many entries does each TLB have? Briefly explain your answer. Hint: In each of these questions, think about how the linked-list `malloc` implements allocation and freeing of memory.

480KB / 4 KB = 120 pages

Each core has its own TLB, but a thread running on each core is accessing all the 120 pages (since allocation requires following the linked list that is likely spread across all the pages). Thus the TLB on each core has copies of the 120 page table entries.

Since the TLB also likely has other entries for code and data pages, each TLB likely has 128 TLB entries.

**Part (b)** [4 MARKS] While debugging the performance problem in FAST, you get side tracked because you recently learned about the `mmap()` system call and would like to make FAST a more reliable program. So you replace each FAST thread with a process. Then you use `mmap()` to map a large file in the address space of each of the processes. The file is mapped at the same virtual address range in the different processes. You know that this new mapped region is shared by the different processes, and you reimplement a new `process_malloc` library using this mapped region. Your new implementation still uses the linked list implementation. Does the performance of the process-based FAST application degrade similar to thread-based FAST? If so, does it degrade when the `total_heap_size()` is 1) much less than 480 KB, 2) roughly 480 KB, or 3) much more than 480 KB? Briefly explain your answer in terms of TLB usage. You can ignore the cost of reading or writing the file on disk.

The process-based FAST suffers the same degradation as the thread-based FAST program at roughly 480KB. While the process-based FAST will have different page table entries for each process (unlike thread-based FAST, where the threads share the page table), the TLB on each core is storing page table entries for the same 120 frames, so the performance degradation will happen at the same heap size.

**Part (c)** [4 MARKS] Coming back to thread-based FAST, you decide to reimplement a new `thread_malloc` library in which each thread uses its own heap area. To do so, you split the heap region into roughly four parts (called t-regions), and each thread allocates memory from its own t-region. Your new implementation still uses the linked list implementation within a t-region. You can assume that each t-region is sized on program startup and does not need to grow. Does the t-region based FAST application perform similar to the original thread-based FAST? If not, how does its performance change? Briefly explain your answer in terms of TLB usage and `total_heap_size()`.

The t-region based FAST will perform much better than the original thread-based FAST, because the TLB entries will contain page table entries for different pages. In this case, the performance of t-region FAST should degrade once each thread is accessing 480KB, so the performance should degrade when total_heap_size() returns 480 * 4 = 1920KB.

**Part (d)** [2 MARKS] Can you think of any other reason (besides TLB usage) why t-region based FAST may perform slower/faster than the original FAST?

Since each thread uses its own heap area, there is no need for synchronization (using atomic instructions, blocking locks, etc.), when allocating or freeing memory, so t-region FAST should be much faster/scalable that original threaded FAST.

**Part (e)** [4 MARKS] You would like to improve the performance of FAST so that it can handle much larger amounts of heap memory, without suffering as much from TLB-based performance degradation. Suggest a way in which you would change the `malloc` library implementation to improve its performance.

List-based allocation is not efficient in terms of TLB usage since it requires potentially traversing the entire heap. This problem is similar to list-based allocation in file systems, where accessing the index information required traversing the blocks in a linked list. The solution there was to collect and store all the index information together (FAT file system). We could use a similar scheme, where the chunk header information is kept in one area (say in the beginning of the heap), and the rest of the heap will have the chunk data. In this case, allocation will only require accessing the chunk headers at the beginning of the heap, reducing the number pages in the heap that need to be traversed, thus reducing TLB usage.

## Bonus. [2 MARKS]

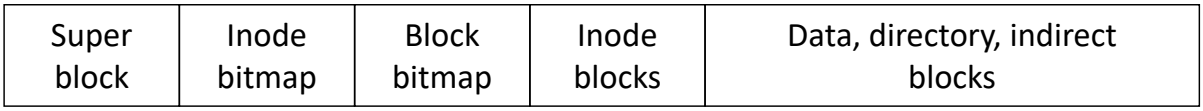
As a bonus, suggest a second, different way to change the `malloc` library implementation to improve its TLB performance. If your changes require any changes to the callers of the library (e.g., the FAST program), mention what changes are needed.

It is possible to optimize the malloc implementation by defragmenting the chunks so that allocated chunks are located close together. This will reduce the memory footprint of the malloc code, reducing the number of pages that need to be accessed to allocate or free heap memory. In practice, it is hard to move allocated chunks since any program pointer may point to allocated memory, especially in low-level languages like C. In managed languages, a garbage collector frees chunks that are no longer accessible, thus allowing allocation from freed memory, which helps reduces fragmentation.

We were looking for terms like garbage collection or defragmentation in your answer. Since we haven't talked about this in class, this was a Bonus question.

## Question 6. File System Design [18 MARKS]

The `testfs` file system that you have been working on in the lab has an annoying limitation. It only allows you to create a fixed maximum number of files. If you try to create any additional files, you are returned an ENOSPC error, and the file (or directory) creation fails. The `testfs` file system layout is shown below:




**Part (a)** [2 MARKS] Based on the file system layout, clearly explain why the ENOSPC error is returned.

There are a fixed number of inodes in the inode blocks regions, and similarly, there are a fixed number of bits in the inode bitmap that track the allocation status of the inodes. Once all inodes are allocated, and we try to create a file, the ENOSPC error is returned.

**Part (b)** [8 MARKS] You modify `testfs` so that it can create any number of files, as long as there is enough space on disk. You call this file system `cxfs` (c - creates lots of files, x - because it sounds good). To ensure that you don't spend too much time writing and testing new code, you should make as **few changes** as possible to `testfs`. Hint: Think about other resizeable data structures in `testfs`.

Show the format of the `cxfs` file system. Using this format, describe the changes you have made to `testfs` to implement `cxfs`. Clearly explain how it imposes no limit on the number of files in the file system (other than when there is not enough space on disk).

There are many options here. One of the simplest is to use a dedicated file for the inode bitmap (inode bitmap file), and a dedicated file for the inode blocks (inode file). Since files can grow, we can grow the number of bits in the inode bitmap file, or the number of inodes in the inode file. This approach is similar to storing directory data using inodes. We can store 2 inodes in the super block for the inode bitmap and inode files. Then using these inodes, we can find these files.

**Part (c)** [2 MARKS] Does the `cxfs` file system perform better, worse or the same as the `testfs` file system? Briefly explain your answer.

cxfs performs slightly worse because we need to access a file for the inode bitmap and a file for the inodes, so some extra blocks may need to be read, e.g., the indirect blocks of these files. Also, the blocks of these files may not be stored contiguously, which can induce seeks.

If we assume that the superblock is cached, then there is no extra cost to accessing the inode of these files.

**Part (d)** [2 MARKS] You decide to improve the performance of both `testfs` and `cxfs` file systems by implementing a block-based caching scheme. The block cache uses LRU replacement. You find that the `cxfs` file system performs slight worse than the `testfs` file system. What could be the reason?

The reason is the same as Part (c). Some additional file blocks need to be accessed for the inode bitmap and inode file.

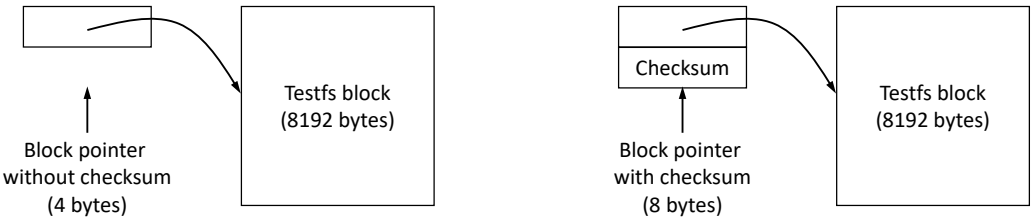
**Part (e)** [4 MARKS] Can you think of any way to modify the block replacement policy in `cxfs` so that its performance is close to `testfs`?

We could pin or prioritize the additional blocks (e.g., indirect blocks) of the inode bitmap and inode files in the block cache, so that these blocks are unlikely to be evicted. Thus the number of block accesses for accessing the inode bitmap and inode blocks will be comparable to testfs. Prioritizing these additional blocks reduces the effective number of blocks available in the block cache, which may still impact the performance of cxfs, but it should be neglible if the cache is large.

## Question 7. File System Reliability [14 MARKS]

You have recently heard that disks are not reliable. After a sector is written, a *latent sector error* may occur where the contents of the sector may become incorrect, and further reads of the sector will then return incorrect contents. Worse, the disk may not know or tell the file system that such an error has occurred. As a result, the file system reads the sector and may return the incorrect data in the sector to the user application. Even worse, if the sector is part of file system metadata, then the file system may crash or cause further corruption when interpreting the data in the sector.

You decide to handle this serious data reliability problem in the `testfs` file system that you have been working on in the lab. You have heard that *checksums* are small fixed-sized data that are derived from some arbitrary-size data for the purpose of detecting errors. You decide to add a checksum to each file-system block pointer to help detect corruption of the pointed-to block. You call the resulting file system `sumfs`.




The picture above shows how a checksum can be added to a block pointer. The checksum is a value that is derived from the contents of the `testfs` block. You apply the string hash function (from Lab 1) to the contents of the `testfs` block by treating the entire (8192 byte) contents as a string (you can ignore the fact that the `testfs` block may contain the c-string termination character). The hash value that is generated is the checksum value: `checksum = Hash(block_data)`.

Checksums are used as follows: when a block is written, the checksum value is generated based on the updated block contents, and stored with the block pointer that points to the block. When a block is read (via the block pointer), the checksum is regenerated using the block contents, and this generated checksum is compared with the checksum stored in the block pointer. If there is a mismatch, the data that is read is not the same as the data that was written to the block, and hence a latent sector error has been detected. Note that checksums may lead to certain errors being undetectable due to hash collisions, i.e., `Hash(block_data_with_error)` is the same as `Hash(block_data)`, but we will ignore that issue.

The program listing below shows the inode structure of `testfs` on disk. If the questions below ask for a calculation, you may provide your answer as a formula or as a final value.

```
struct dinode {
  s32 i_type;           /* s32 is 4 bytes, i_type is file type */
  s32 i_block_nr[10];
  s32 i_indirect;
  s32 i_dindirect;
  s32 block_count;
  off_t i_size;         /* off_t is 8 bytes, i_size is file size */
};
```

**Part (a)** [2 MARKS] What is the maximum size of a file in the original `testfs` file system? Provide your answer in terms of number of `testfs` blocks.

Nr of pointers in an indirect block = 8192/4 = 2048
10 + 2048 + 2048^2 blocks = 4196362

**Part (b)** [2 MARKS] What is the maximum size of a file in the `sumfs` file system? Assume that the number of block pointers in the inode structure of `sumfs` are the same as in `testfs`. Provide your answer in terms of number of `sumfs` blocks.

Nr of pointers in an indirect block = 8192/8 = 1024
10 + 1024 + 1024^2 blocks = 1049610

**Part (c)** [1 MARK] Roughly, what is the ratio of the maximum size of a file in `sumfs` to the maximum size of a file in `testfs`?

1024^2 / 2048^2 = 1/4

**Part (d)** [2 MARKS] Suppose `testfs` allows creating a maximum of roughly 14000 files. Roughly, how many files can be created in `sumfs`? Show your calculations.

inode size of testfs = 64

inode size of sumfs = 112 (only the 12 block pointers are doubled, so 12 * 4 = 48 extra bytes)

So sumfs can store roughly 64/112 * 14000 = 4/7 * 14000 = 8000 files. We accepted 7000 as an answer as well (inode size doubles).

**Part (e)** [3 MARKS] Are all block corruptions detectable in `sumfs`? If so, explain why. Otherwise, explain where else you would add checksums to detect all block corruptions.

Not all block corruptions are detectable because the statically allocated blocks (super block, bitmaps, inode blocks) are not checksummed. Note that these blocks are not accessed via block pointers, and hence we would need to embed the checksum in the block. We can add a checksum to the super block since it has extra space. We would need to create a header for each of the bitmap blocks and inode blocks to store the checksum information.

**Part (f)** [4 MARKS] With all the changes above to `sumfs`, suppose that you create a new file of 8 KB size on both file systems. Are the number of blocks that are modified the same or different in `testfs` and `sumfs`. Clearly explain your answer.

To create a file, we need to update the inode bitmap (a new bit is allocated), block bitmap (a new bit is allocated), inode block (an inode for the new file is initialized), a data block (which stores the contents of the file), the directory data block (to add a directory entry for the new file), and the directory inode block (to update timestamp, size of the directory). Both testfs and sumfs would update exactly these same set of 6 blocks because these blocks will contain the checksums for these 6 blocks.

## Question 8. Block-Level Crash Consistency [10 MARKS]

After gaining much experience with designing `sumfs`, a reliable, checksum-based file system, you realize that checksums can also help with ensuring crash consistency.[1]

You know that disks update a sector atomically, but they provide no such guarantees when updating multi-sector blocks. Assume that you need to update a single disk Block B consisting of 8 sectors. You have a spare Block S, and another Sector D, available to you. You design the following scheme for updating Block B atomically:

1. Write old version of Block B in memory to Block S on disk.
2. Write the checksum of the old version of Block B to Sector D on disk, flush blocks.
3. Write new version of Block B from memory to Block B to disk, flush blocks.
4. Write a special, invalid checksum value to Sector D on disk, flush blocks.

**Part (a)** [5 MARKS] Does this scheme update Block B atomically? If not, explain why there may be a problem. If it is correct, describe the block recovery procedure you would use after a crash, and also mention the step number at which the updated Block B is committed.

This scheme updates Block B atomically.

Recovery:

Case 1: If the checksum written in Sector D matches the checksum of spare Block S, then we must be at stage 3. In this case, we do undo recovery by copying the contents of Block S (old version of Block B) to Block B.

Case 2: Otherwise, we are at step 1, 2 or 4. In this case, Block B was never written (old version), or was fully written (new version), and we don't need to do anything.

**Part (b)** [2 MARKS] The design above flushes blocks in Steps 2, 3, and 4. Briefly explain what happens on a block flush.

On a block flush, all block writes that have been issued and acknowledged before the block flush is issued are written to disk durably by the time the block flush finishes. This is used to ensure both durability of writes, and ordering of writes. In particular, the code sequence above requires (1, 2) to occur before 3, and 3 to occur before 4, and 4 to occur before (1, 2).

[1] If you haven't done so yet, you should read about checksums on the first page of the File System Reliability question before reading this question any further. In particular, you can ignore checksum collisions for this question.

**Part (c)** [1 MARK] Describe a potential problem with the scheme if blocks are not flushed in Step 2.

A new version of the block could be partially written before the old version or its checksum are updated. On a crash, since the checksum will not match, we will apply Case 2 for recovery, and do nothing, but the new version is not written fully.

**Part (d)** [1 MARK] Describe a potential problem with the scheme if blocks are not flushed in Step 3.

We could invalidate the checksum before a new version of the block is fully written. On a crash, since the checksum will not match, we will apply Case 2 for recovery, and do nothing, but the new version is not written fully.

**Part (e)** [1 MARK] Describe a potential problem with the scheme if blocks are not flushed in Step 4.

Let's say a crash happens long after Step 3 is performed but before the checksum block is invalidated. On a crash, since the checksum will match, we will apply Case 1 of recovery, and undo the correctly updated block. This doesn't violate block atomicity, but it does cause a durability problem (e.g., if the user is told that the block has been written after Step 4, on crash recovery, the updated version will not be found -- this happens because we didn't flush the checksum invalidation).

*[Use the space below for rough work.]*