CS 162 Spring 2021

Operating Systems and Systems Programming

Midterm 2

INSTRUCTIONS

This is your exam. Complete it either at exam.cs61a.org or, if that doesn't work, by emailing course staff with your solutions before the exam deadline.

This exam is intended for the student with email address `<EMAILADDRESS>`. If this is not your email address, notify course staff immediately, as each exam is different. Do not distribute this exam PDF even after the exam ends, as some students may be taking the exam in a different time zone.

For questions with **circular bubbles**, you should select exactly *one* choice.

◯ You must choose either this option

◯ Or this one, but not both!

For questions with **square checkboxes**, you may select *multiple* choices.

☐ You could select this choice.

☐ You could select this one too!

You may start your exam now. Your exam is due at <DEADLINE> Pacific Time. Go to the next page to begin.

This is a **proctored, closed-book exam**. During the exam, you may **not** communicate with other people regarding the exam questions or answers in any capacity.

When answering questions, make no assumptions except as stated in the problems. If there that you believe is open to interpretation, please use the "Clarifications" button to request a clarification. We will issue an announcement if we believe your question merits one.

This exam has 4 parts of varying difficulty and length. Make sure you read through the exam completely before starting to work on the exam. Answering a question instead of leaving it blank will **NOT** lower your score. We will overlook minor syntax errors in grading coding questions.

(a)

Name

(b)

Student ID

(c)

Please read the following honor code: "I understand that this is a closed book exam. I hereby promise that the answers that I give on the following exam are exclusively my own. I understand that I am allowed to use two 8.5x11, double-sided, handwritten cheat-sheets of my own making, but otherwise promise not to consult other people, physical resources (e.g. textbooks), or internet sources in constructing my answers." **Type your full name below to acknowledge that you've read and agreed to this statement.**

1. **(18.0 points) True/False**

Please **EXPLAIN** your answer in **TWO SENTENCES OR LESS**. Answers without any explanation **GET NO CREDIT.**

(a) **(3.0 points)**

i.

In a page table-based addressing scheme, every virtual address in a process' virtual address space always corresponds to a valid page table entry.

○ True

● False

ii.

Explain.

Virtual addresses may not be mapped to a physical address (and hence, have an invalid PTE) due to multiple reasons. For example, the size of virtual memory may be greater than physical memory, which requires pages to be swapped out for other pages on disk. This leads to invalidating the PTE's of pages being evict from memory. Another example is if a page corresponding to a requested virtual address hasn't been accessed prior and fetched into memory yet, the PTE would be invalid.

(b) (3.0 points)

i.

In a page table-based addressing scheme, the TLB is a cache for page table entries.

○ True

● False

ii.

Explain.

The TLB stores a mapping of the virtual page numbers in a virtual address to the physical page number in the physical address. Therefore, the TLB acts as a cache for address translations.

(c) (3.0 points)

i.

Page faults always cause a user program to terminate.

○ True

● False

ii.

Explain.

Page faults normally just trigger a read from disk.

(d) i. (3.0 points)

In PintOS with priority donation, immediately after a thread releases a lock, the thread's effective priority may be different from the thread's base priority.

- ● True
- ○ False

B.

Explain.

There could be multiple locks and other threads with a priority higher than the base priority may have donated their priority. Example: Let's say T1 is holding L1 and L2. T2 with higher priority waits on L1. T3 with even higher priority waits on L2. T1 releases L2.

(3.0 points)

In PintOS with priority donation, immediately after a thread releases a lock, the thread's effective priority must be the same as the thread's base priority.

ii. Ⓐ True

● False

B.

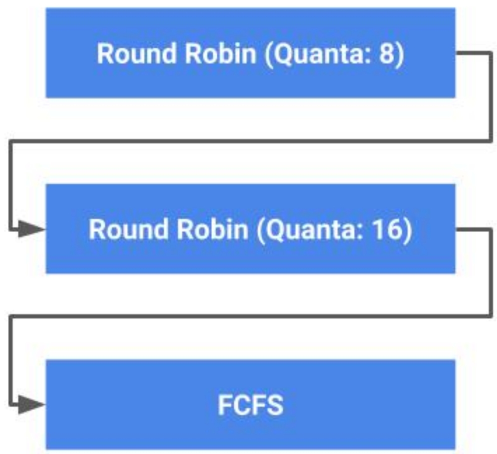
Explain.

There could be multiple locks and other threads with a priority higher than the base priority may have donated their priority. Example: Let's say T1 is holding L1 and L2. T2 with higher priority waits on L1. T3 with even higher priority waits on L2. T1 releases L2.

(e) (3.0 points)

i.

Assume that Job A always bursts for 10 ticks at a time, and runs forever. Also assume that our system uses a MLFQS (Multi-Level Feedback Queue Scheduler), with the following structure:



(Round-Robin (Q=8), Round-Robin (Q=16), FCFS)

If Job A is initially placed on the top queue when it enters the system, it will eventually be placed on the middle queue; after this point, when the job is not running, it will always be placed on the middle queue.

- ○ True
- ● False

ii.

Explain.

Job A bursts for 10 ticks, which means it runs for 8 ticks in the top queue before expiring and being moved to the middle queue. In the middle queue, it runs for another 2 ticks then finishes the burst and yields, thereby moving back up to the top queue. The job will continually move between the top two queues.

(f) (3.0 points)

i.

Assume that, in our system, there are 10 jobs that will run for 100 ticks total (and never voluntarily yield the CPU or block). If our system uses a round robin scheduler, increasing the quantum from 10 ticks to 20 ticks will always result in higher throughput.

- ● True
- ○ False

ii.

Explain.

This is true because the system will preempt and context switch between threads about half as many times. Since each switch takes some non-zero amount of time, the total time to run the 10 jobs will be less with a higher quantum, implying a higher throughput.

2. (18.0 points) Multiple Choice

In the following multiple choice questions, **please select all options that apply**.

(a) (3.0 pt)

Which of the following scheduling algorithms may cause starvation?

- [x] First Come First Serve (FCFS)
- [ ] Round-Robin (RR)
- [x] Shortest Remaining Time First (SRTF)
- [ ] Completely Fair Scheduler (CFS)
- [ ] None of the above

Starvation may occur in FCFS when a thread runs indefinitely, and in SRTF when systems have many bursty threads, thus long-running threads never get run. In RR and CFS, preeemption can occur, and all threads will run.

(b) (3.0 pt)

Which of the following descriptions of Banker's Algorithm are correct?

- [ ] Banker's Algorithm is used in deadlock recovery.
- [ ] Banker's Algorithm can be used to determine whether or not a system is guaranteed to deadlock.
- [x] Using Banker's Algorithm results in an execution that is guaranteed to avoid deadlock.
- [x] Banker's Algorithm relies on having known upper limits for resource requests per thread.
- [ ] None of the above.

Banker's Algorithm is used in deadlock avoidance and determines whether or not a system is in an unsafe state. Being in an unsafe state does not mean a system is guaranteed to deadlock; however, Banker's Algorithm does give us a sequence of safe serial executions that, while inefficient, avoids deadlock.

(c) (3.0 pt)

Which of the following scheduling algorithms are (1) deterministic and (2) will eventually preempt any infinitely looping thread in favor of a thread with finite run-time?

- [ ] First Come First Serve (FCFS)
- [x] Round-Robin (RR)
- [x] Multi-Level Feedback Queue Scheduling (MLFQS)
- [ ] Lottery Scheduling
- [ ] Strict Priority Scheduling
- [x] Shortest Remaining-Time First Scheduling (SRTF)
- [ ] None of the above

Lottery Scheduling is not deterministic. FCFS will run a infinitely looping thread indefinitely. SPS will run a high-priority infinitely looping thread indefinitely.

(d) (3.0 pt)

Which of the following could be components of a page table entry (PTE)?

- [x] Physical page number
- [ ] Virtual page number
- [x] Valid bit
- [ ] Offset
- [ ] None of the above

A PTE in a system of multi-level page tables may contain the PPN of the physical address or of the next page table. Metadata bits are also part of PTEs. The VPN is used to index into page tables, and the offset is used to index into a page; both are part of virtual addresses, not PTEs.

(e) (3.0 pt)

Assume that the goal for our page replacement policy is to minimize total page replacements. Which of the following statements are accurate?

- [x] FIFO can have the same performance as MIN.
- [ ] Clock Algorithm always has the same performance as Second-Chance List Algorithm.
- [ ] LRU performs at least as good as any Nth-Chance Clock Algorithm.
- [ ] An optimal page replacement policy prevents thrashing.
- [ ] None of the above.

FIFO can have the same performance as MIN – for example, in the trivial case where the working set is less than the size of physical memory. Second-Chance List Algorithm uses half the pages of physical memory for a second-chance list, while [Second-Chance] Clock Algorithm uses a separate use bit to access all pages of physical memory at full speed. Nth-Chance Clock Algorithm is an approximation of LRU, so there is no guarantee of better or worse. (For example, consider LRU vs. Clock for the memory access pattern ACABC in a system with 2 pages of physical memory). Thrashing is dependent on memory size, not replacement policy.

(f) (3.0 pt)

Assume we make a single memory access on a processor which has no caches and no TLB. All else being equal, which of the following are true?

- [x] Base-and-bound will on average be faster than a single-level page table.
- [ ] An inverted page table will on average be faster than a single-level page table.
- [x] A single-level page table will on average be faster than a multi-level page table.
- [ ] A multi-level page table will on average be faster than an inverted page table.
- [ ] None of the above.

To translate the virtual address to a physical address, base-and-bound would require 0 memory accesses (only arithmetic operations), a single-level page table and an inverted page table would both require 1 memory access (to access the page table), and a multi-level page table would require multiple memory accesses, depending on the number of page tables to traverse.

3. (38.0 points) Short Answer

(a) (11.0 points) Virtual Memory Bits Pieces

Suppose we have virtual memory mapped with a two-level page table scheme, where each page table fits *exactly* inside a page, and each page table entry is exactly 4 (2^2) bytes in size. We also have a physical memory size of 16 MiB (2^{24} B), and a page size of 4 KiB (2^{12} B). Additionally, we have a fully-associative tagged TLB, where each entry includes 1 byte of metadata containing information including a valid bit, dirty bit, and a process ID.

i. (1.0 pt)

How many bits are in the offset of an address in this scheme?

12

log(4 KiB) = 12 bits.

ii. (1.0 pt)

How many bits are in the first-level page table index in this scheme?

10

log(4 KiB / 4 B) = 10 bits.

iii. (1.0 pt)

How many bits are in the second-level page table index in this scheme?

10

log(4 KiB / 4 B) = 10 bits.

iv. (1.0 pt)

How many bits are in the physical page number in this scheme?

12

log(16 MiB / 4 KiB) = 12 bits.

v. (1.0 pt)

How large of a virtual address space does this scheme address, in bytes? You may leave your answer as a power of 2.

2^32

Two-level page table: 2^10 * 2^10 * 4 KiB = 2^32 bytes = 4 GiB.

vi. (2.0 pt)

If our TLB had 8 entries, how big is it in bytes?

40

TLB entry size = VPN size (2 * 10 bits) + PPN size (12 bits) + 1 byte of metadata. Each entry is 40 bits. 40 * 8 entries / 8 bits per byte = 40 bytes.

vii. (2.0 pt)

Assume we start with an empty TLB and make the following memory accesses:

- Process 1 accesses `0x10000000`
- Process 1 accesses `0x10001000`
- Process 2 accesses `0x10000000`

How many entries of our TLB will now be filled?

3

We track the process ID in our TLB metadata, so multiple processes' memory mappings can live in our TLB at once.

viii. (2.0 pt)

How many different pages will now be in memory, assuming there were 0 valid pages in physical memory before these accesses?

3, or 7

A clarification was made during the exam to also consider the pages in the page table, but we decided to accept both answers that consider page tables and those that don't. If we do NOT consider the PT, since Process 2's `0x10000000` is different from Process 1's `0x10000000`, we will have 3 valid physical pages. If we DO consider the PT, then there will be 7 pages in total:

- Process 1: because the first level index is the same for both accesses, we have 1 first level PT + 1 second level PT + 2 data pages
- Process 2: separate processes have separate page tables, so we have another 1 first level PT + 1 second level PT + 1 data page

(b) (9.0 points) Rocky Wrench's Replacements

Monty Mole and Rocky Wrench disagree about the best page replacement policies for demand paging – each is trying to throw a wrench into the other's arguments! In the following problems, assume that the system has 3 pages of physical memory (1-3), and the program they run uses 4 pages of virtual memory (A-D). Provide lists as a comma-separated list of virtual memory accesses (e.g. `A,B,C,D,A`).

i. (3.0 pt)

Rocky Wrench is convinced that the best page replacement policy for demand paging is FIFO (First In First Out). Demonstrate the problems with FIFO by providing a list of 6 additional accesses (after the first two accesses that are given) that would cause every access to page fault.

Page \ Access	A	B						
1	A							
2		B						
3								

Possible Solutions:
C,D,A,B,C,D
D,C,A,B,D,C

ii. (3.0 pt)

Monty Mole is convinced that the best page replacement policy for demand paging is MRU (Most Recently Used). Demonstrate the problems with MRU by providing a list of 6 additional accesses (after the first two accesses that are given) that would cause every access to page fault.

Page \ Access	A	B						
1	A							
2		B						
3								

Possible Solutions:
C,D,C,D,C,D
D,C,D,C,D,C

iii. (3.0 pt)

After putting their disagreements aside, they come up with an optimal page replacement policy. Monty Mole and Rocky Wrench use that replacement policy with a memory access pattern that contains 3 A's, 2 B's, 2 C's, and 1 D (not necessarily in that order; you should not assume it begins with A and B). What is the maximum total number of page faults that could occur when using that optimal page replacement policy on such a memory access pattern?

5

The optimal page replacement policy is MIN (and one could construct an arbitrary heuristic that has the same behavior as MIN for any workload under the above constraints). With 4 pages of virtual memory, the minimum number of page faults with MIN is 4, once per page. To find the maximum number of page faults, we should space the memory accesses out as far as possible, therefore we have to maximize the sum of differences between accesses of the same virtual page. With the above constraints, all memory patterns have either 4 or 5 page faults. One example of 5 is shown below.

Page \ Access	A	B	C	D	A	B	C	A
1	A							
2		B					C	
3			C	D				

(c) (4.0 points) Rosalina's Special Scheduler

Rosalina's Round-Robin (RR) scheduler is a special implementation of RR that considers thread priorities. More specifically, the quanta of a thread (in ticks) is equal to its priority. Say there is an IO-bound thread, Thread A, with a priority of 64 that always bursts for 1 tick in between I/O. Another CPU-bound thread, Thread B, has a priority of 2 and always bursts for 100 ticks in between I/O. Assume each I/O operation takes 1 tick.

i. (2.0 pt)

Assuming the two threads only yield for I/O, for what fraction of time will Thread A run on the CPU on average?

1/3

ii. (2.0 pt)

Why doesn't the higher priority thread (Thread A), with a higher quanta, run for a majority of the time?

Thread A does not use up its quanta, which means that it cannot take advantage of its higher quanta to gain more time on the CPU.

(d) (14.0 points) Mario's Moves

Mario wrote a program that synchronizes three of his special moves as threads.

```
sem_t semaphore;
pthread_mutex_t lock;

void threadA() {
  sem_wait(&semaphore);       // line 1
  printf("threadA!");
  sem_post(&semaphore)
}

void threadB() {
  sem_wait(&semaphore);       // line 1
  pthread_mutex_lock(&lock);  // line 2
  printf("threadB!");
  pthread_mutex_unlock(&lock);
  sem_post(&semaphore);
}

void threadC() {
  sem_wait(&semaphore);       // line 1
  pthread_mutex_lock(&lock);  // line 2
  sem_wait(&semaphore);       // line 3
  printf("threadC!");
  sem_post(&semaphore);
  pthread_mutex_unlock(&lock);
  sem_post(&semaphore);
}

int main() {
  sem_init(&semaphore, 0, /*initial value*/ 2);
  pthread_mutex_init(&lock, NULL);
  pthread_t threads[3];
  pthread_create(&threads[0], NULL, threadA, NULL);
  pthread_create(&threads[1], NULL, threadB, NULL);
  pthread_create(&threads[2], NULL, threadC, NULL);
}
```

i. (4.0 pt)

What order of execution will cause this program to print nothing? Provide your answer as a list of lines of `thread:N`, specifying the order of execution of the lines of code of each thread.
NOTE: include any lines/expressions that start executing but may not finish executing.

Example:

```
threadA:1
threadC:1
threadC:2
threadC:3
threadB:1
```

Possible solutions (brackets indicate any order is acceptable):
threadC:1 threadC:2 threadB:1 {threadA:1 threadB:2 threadC:3}
threadB:1 threadC:1 threadC:2 {threadA:1 threadB:2 threadC:3}
{threadB:1 threadC:1} threadA:1 threadC:2 {threadB:2 threadC:3}
threadC:1 threadB:1 threadC:2 {threadA:1 threadB:2 threadC:3}

ii. (2.0 pt)

What is name of the condition that causes the program to print nothing?

Deadlock

iii. (4.0 points)

Mario realizes his mistake, and decides to implement a new function to replace `pthread_mutex_lock` to prevent this from happening. Complete the function below to ensure that all three print statements are executed regardless of how the threads are scheduled (assume that every `pthread_mutex_lock(&lock)` line is replaced with `pthread_mutex_lock_safe()`). You may only use one statement/expression in each of the provided blanks.

```
void pthread_mutex_lock_safe() {
  ___[A]___;
  pthread_mutex_lock(&lock);
  ___[B]___;
}
```

A.

[A]

sem_post(&semaphore);

B.

[B]

sem_wait(&semaphore);

iv. (4.0 pt)

Explain how your solution above changes the ordering of resource allocation to prevent the condition you noted in 4.2.

This new implementation ensures that every thread acquires the lock before "acquiring" (downing) the resource represented by the semaphore. Since resources are requested in a consistent/particular order, there cannot be deadlock.

4. (26.0 points) Monty Mole's Memory Management

In order to reduce the latency of L1 cache hits, many modern CPU architectures employ a type known as Virtually-Indexed, Physically-Tagged (VIPT) caches. In this system, rather than having to perform address translation to be able to look up an entry in the cache, the CPU instead selects the *cache set* based on the *virtual* address in parallel with address translation, only afterwards storing or comparing bits from the now-translated *physical* address for the tag. You have been contracted to help Monty Mole implement such a design for the L1 data cache of an application-specific operating system.

After much deliberation, you and your team of Koopalings have decided on a 256 byte, 2-way set associative cache with LRU replacement and 16-byte cache lines. To test the effectiveness of this layout, you decide to simulate a series of cache accesses on your new design.

Given the list of memory addresses, the initial state of the cache, and the layout of physical memory below, fill out the the final state of the cache once all memory accesses are complete. An LRU bit value of 0 indicates that that set's Entry 0 is up for eviction, while an LRU bit value of 1 indicates the same for Entry 1.

Assume the following details hold for the system throughout the problem:

- The operating system uses single-level paging for address translation.
- Pages are 256 bytes in size.
- The Page Table Base Register contains the value `0x10000000`.
- All relevant virtual addresses are present within the system's TLB before execution begins, and remain there throughout.
- All memory contents are displayed in big-endian order.
- Each Page Table Entry is 4 bytes long: bits [31:8] contain the PPN, bit 0 contains the valid bit, and bits [7:1] are unspecified.
- All pages are fully accessible by the running process.

You may find it helpful to use the emailed exam PDF as a reference in this problem. The physical memory table below is also available as a separate PDF. **Some answers may be related, so use shortcuts wherever possible!**

Address	0x0	0x4	0x8	0xC
0x10000000	0x10840111	0x330CD003	0x00000000	0x10010000
0x10000010	0xB33A0001	0xA55A0200	0x01A05203	0x00000000
...				
0x1003C000	0xA55A0201	0x5500BE47	0xA55A0001	0xA55A0101
0x1003C010	0x3300C081	0x00000000	0x00000000	0xA55A0200
...				
0x12100440	0x3300BE47	0x33000000	0x3300C001	0xB55A0201
0x12100450	0x730F1101	0x730F1201	0x730F1301	0xB55A0200
...				
0x3300C000	0xFEEEEEED	0xD0600000	0xFA1AFE15	0x5EAF00D5
0x3300C010	0xCAFEF00D	0x444417F0	0xF865179C	0x00000000
0x3300C020	0x00000000	0x00000000	0x00000000	0x00000000
0x3300C030	0xBEAD2000	0x60001147	0x51EFA1AF	0x9114F000
...				
0x3300C0C0	0x5EAF00D5	0x00000000	0x00000000	0x0001FFFF
0x3300C0D0	0xBEEEEEEF	0xFFFFFFFF	0xFFFFFFFF	0xFFFFFFFF
0x3300C0E0	0xFFFFFFFF	0xBEEEEEEE	0xEEEEEEEE	0xEEEEEEEF

Address	0x0	0x4	0x8	0xC
0x3300C0F0	0x10101010	0x20202020	0x30303030	0x40900000
...				
0xA55A0120	0xE32BFFFE	0xE32C0000	0xE32C0002	0xE32C0004
0xA55A0130	0xE32C0006	0xE32C0008	0xE32C000A	0xE32C000C
0xA55A0140	0x007A19E1	0x0024CE00	0x04S93C70	0x0BADBEEF
0xA55A0150	0xACACACAC	0xACACACAC	0x00020004	0x00010004
...				
0xA55A01A0	0xF000000D	0x00000000	0xC0CAC01A	0x00011000
0xA55A01B0	0xABABABAB	0xABABABAB	0x0000F045	0x01F0FC35
0xA55A01C0	0xA55A01C0	0xA55A01C4	0xA55A01C8	0xA55A01CC
0xA55A01D0	0x00000000	0x0000000C	0x08C48348	0x0005A1AD
...				
0xA55A0240	0xE32C0000	0x30EC8348	0xE5894855	0x48D07589
0xA55A0250	0x48DC7D89	0x89480000	0x0AAA058D	0x000ACB05
0xA55A0260	0x8D48F845	0x558B48F0	0xBEEFBEEF	0xD60948F8
0xA55A0270	0x48C68948	0xE8000000	0x00000000	0x0BADBEEF
...				
0xB33A00C0	0x01331330	0x00000000	0xBEEFBEEF	0x1A1A1A1A
0xB33A00D0	0x00000000	0xE32C0000	0x70000000	0x00000002
0xB33A00E0	0x00010000	0x00030012	0x5EAF000D	0xBEEFBEEF
0xB33A00F0	0x00000000	0x00000000	0x00000000	0xACACACAC

Set	**LRU Bit**	**Tag for Entry 0**	**Tag for Entry 1**
0	1	0x3300C0	
1	0	0xA55A01	0x800230
2	1	0xB55A03	0xB55A04
3	0		
4	1	0xA55A02	0x800113
5	0	0x800010	
6	0	0x3300C0	0x840112
7	1	0x70F3F0	

You may also find this table of hex to binary conversions helpful:

Hex	0x0	0x1	0x2	0x3	0x4	0x5	0x6	0x7	0x8	0x9	0xA	0xB	0xC	0xD	0xE	0xF
Binary	0000	0001	0010	0011	0100	0101	0110	0111	1000	1001	1010	1011	1100	1101	1110	1111

(a) (15.0 points)

Virtual Address	**Physical Address** (1 pt ea.)	**Set** (0.5 pt ea.)	**Hit?** (0.5 pt ea.)	**Value** (1 pt ea.)
`0x840110C0`	(a)	**6**	**Hit**	`0xBEEFBEEF`
`0x00F003DC`	(b)	**6**	**Miss**	`0x0005A1AD`
`0x00F00324`	(c)	(d)	(e)	(f)
`0x840110C0`	(g)	**6**	**Hit**	`0xBEEFBEEF`
`0x84011208`	`0x3300C008`	(h)	(i)	(j)
`0x840112D0`	`0x3300C0D0`	(k)	(l)	(m)
`0x00F00040`	`0xA55A0240`	(n)	(o)	(p)
`0x00F00070`	`0xA55A0270`	(q)	(r)	(s)
`0x00000164`	(t)	**3**	**Miss**	`0xC001C142`

i.

(a)

0x3300BEC0

ii.

(b)

0xA55A01DC

iii.

(c)

0xA55A0124

iv.

(d)

6

v.

(e)

Hit

◯ Hit

◯ Miss

vi.

(f)

0xE32C0000

vii.

(g)

0x3300BEC0

viii.

(h)

6

ix.

(i)

Hit

◯ Hit

◯ Miss

x.

(j)

0xFA1AFE15

xi.

(k)

6

xii.

(l)

Hit

◯ Hit

◯ Miss

xiii.

(m)

0xBEEEEEEF

xiv.

(n)

2

xv.

(o)

Miss

◯ Hit

◯ Miss

xvi.

(p)

0xE32C0000

xvii.

(q)

3

xviii.

(r)

Miss

◯ Hit

◯ Miss

xix.

(s)

0x48C68948

xx.

(t)

0x330CD064

(b) (5.0 points)

Suppose we were to entirely remove the TLB from the system.

i. (2.0 pt)

Would you expect the *total number* (not the rate) of cache misses to increase, decrease, or be unaffected by the change?

- ● Increase
- ○ Decrease
- ○ No Change

ii. (3.0 pt)

Why so?

Without a TLB, every access to main memory will require additional 'surplus' accesses as a result of address translation; these accesses both have a chance to miss on their own as well as a chance to knock existing entries out of the cache, causing misses down the line.

(c) (6.0 points)

Imagine if you were to scale the capacity of the cache by a factor of 4, from 256 bytes to 1 KiB, while leaving all other parameters (associativity, replacement policy, etc.) unchanged. Suppose two processes (different virtual -> physical mappings) execute on the CPU one after another, resulting in the following two virtual/physical address combinations being present in the cache simultaneously:

Process	Virtual Address	Physical Address
A	`0x400000F4`	`0x700000F4`
B	`0x5594FFF0`	`0x700000F0`

i. (3.0 pt)

What problem(s) could this cause?

Even though both `0x400000F4` and `0x5594FFF0` map to the same physical address (via different page tables, in different processes) the former has index bits `0b0111` while the latter has index bits `0b1000`, resulting in the two addresses being stored in different cache ways, even though the underlying physical memory they represent is the same. Any mention of aliasing counts for the "what problems" portion; writing about the effects of aliasing, like changes to memory not properly propagating between processes, is also acceptable.

ii. (3.0 pt)

Could this problem also occur in the 256-byte VIPT cache described above? Why so or why not?

This would not have been possible in the 8-way VIPT cache; in that case, all index bits were selected from within the offset bits of a page, resulting in them correctly coming from the physical address. Reference to how, conversely, some of the index bits come from the VPN in the 16-way VIPT cache works as well.

5. Reference Sheet

```
/********************************** Threads **********************************/
int pthread_create(pthread_t *thread, const pthread_attr_t *attr,
                   void *(*start_routine) (void *), void *arg);
int pthread_join(pthread_t thread, void **retval);
int pthread_mutex_init(pthread_mutex_t *mutex, const pthread_mutexattr_t *attr);
int pthread_mutex_lock(pthread_mutex_t *mutex);
int pthread_mutex_unlock(pthread_mutex_t *mutex);
int sem_init(sem_t *sem, int pshared, unsigned int value);
int sem_post(sem_t *sem); // up
int sem_wait(sem_t *sem); // down
int pthread_cond_init(pthread_cond_t *cond, pthread_condattr_t *cond_attr);
int pthread_cond_signal(pthread_cond_t *cond);
int pthread_cond_broadcast(pthread_cond_t *cond);
int pthread_cond_wait(pthread_cond_t *cond, pthread_mutex_t *mutex);

/********************************* Processes *********************************/
pid_t fork(void);
pid_t wait(int *status);
pid_t waitpid(pid_t pid, int *status, int options);
int execv(const char *path, char *const argv[]);

/******************************* High-Level I/O *******************************/
FILE *fopen(const char *path, const char *mode);
FILE *fdopen(int fd, const char *mode);
size_t fread(void *ptr, size_t size, size_t nmemb, FILE *stream);
size_t fwrite(const void *ptr, size_t size, size_t nmemb, FILE *stream);
int fclose(FILE *stream);

/******************************* Low-Level I/O ********************************/
int open(const char *pathname, int flags);
ssize_t read(int fd, void *buf, size_t count);
ssize_t write(int fd, const void *buf, size_t count);
off_t lseek(int fd, off_t offset, int whence);
int dup(int oldfd);
int dup2(int oldfd, int newfd);
int pipe(int pipefd[2]);
int close(int fd);
```

No more questions.