Final Exam

601.229 Computer Systems Fundamentals

Spring 2020

Johns Hopkins University

Instructors: Xin Jin and David Hovemeyer

14 May 2020

Complete all questions.

Use additional paper if needed.

Take-home exam

Q1. Caches *20 points*

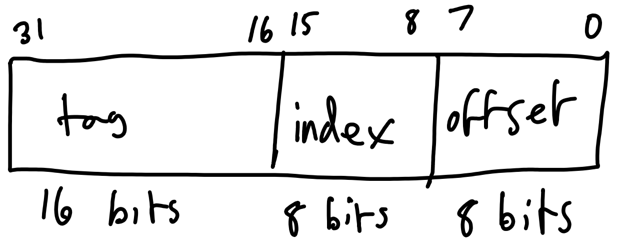
(a) [4 points] An architecture has 32 bit addresses. Assume the following cache parameters:

- 262,144 (2^{18}) total bytes of data
- Each block contains 256 (2^{8}) bytes
- 4-way set associativity

$$2^{18} \text{ bytes} \times \frac{1 \text{ block}}{2^{8} \text{ bytes}} = 2^{10} \text{ blocks}$$

$$2^{10} \text{ blocks} \times \frac{1 \text{ set}}{4 \text{ blocks}} = 2^{8} \text{ sets}$$

Draw a diagram showing the structure of an address, indicating the exact ranges (bit positions) of the offset bits, the index bits, and the tag bits.



(b) [4 points] How many address bits are used to represent the index when a cache is fully associative? Explain briefly.

None. The number of index bits is $\log_2 S$ where S is the number of sets. A fully associative cache has only 1 set, and $\log_2 1 = 0$.

(c) [12 points] Complete the following table. For each address in the *Request* column, indicate the tags of cached blocks after handling the request. Addresses are 8 bits, blocks are 8 bytes, there are 4 sets, and the cache is 2-way set associative. All slots are initially empty. Use FIFO (First-In, First-Out) when replacing blocks.

	Set 0		Set 1		Set 2		Set 3	
Request	Slot 0	Slot 1	Slot 0	Slot 1	Slot 0	Slot 1	Slot 0	Slot 1
	empty	empty	empty	empty	empty	empty	empty	empty
01101111			011					
10010111			↓		100			
10100111	101		↓		↓			
00010111	↓		↓		↓	000		
10101100	↓		↓	101	↓	↓		
11001010	↓		110	↓	↓	↓		
00101100	↓		↓	001	↓	↓		
10100101	↓		↓	↓	↓	↓		
10000101	↓	100	↓	↓	↓	↓		
01011011	↓	↓	↓	↓	↓	↓	010	

— offset

— index

— tag

Q2. Processes and linking *20 points*

(a) [10 points] Consider the following program:

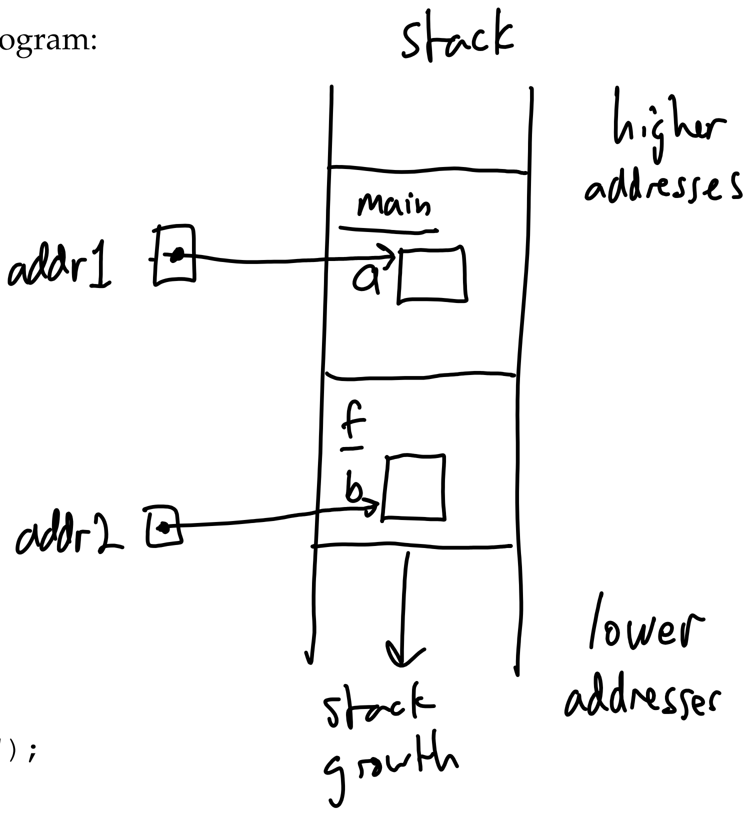
```
#include <stdio.h>
#include <assert.h>

void *addr1, *addr2;

void f(void) {
  int b;
  addr2 = &b;
}

int main(void) {
  int a;
  addr1 = &a;
  f();
  assert(addr1 != addr2);
  if (addr1 < addr2) {
    printf("addr1 < addr2\n");
  } else {
    printf("addr1 > addr2\n");
  }
  return 0;
}
```

note typo



Assuming that this program is compiled for x86-64, what output will the program produce? Explain.

It prints $\boxed{\text{addr1 > addr2}}$.
On x86-64, the stack grows down (towards lower addresses), so f's stack frame occupies a lower range of addresses than main's.

(b) [10 points] When generating a non-position-independent executable, the linker assigns an absolute address to each definition (function or global variable) in the executable. In general, many processes can be executed using the same executable. For example, let's say that

- `/usr/bin/vi` is the executable for the vi editor
- The executable is not position-independent
- Many users are running the executable at the same time

Explain how is it possible that many instances of the executable can be running simultaneously, even though they are all using the same addresses.

Each running instance of /usr/bin/vi is in a separate process, each in a separate virtual address space. The sharable parts of the executable (e.g. .text, .rodata) are loaded into shared memory regions which can be mapped into each process. The non-shared parts (e.g. .data, .bss) are allocated using separate pages of physical memory for each process, but those regions can still be mapped into the same range of addresses in each process because they each have a separate virtual address space.

Q3. Processes and virtual memory *20 points*

(a) [5 points] Name two sections of a Linux executable such that it would be safe for the operating system kernel to map the memory pages containing data loaded from those sections into the address spaces of multiple processes. Explain why it is safe to share the memory for those sections. If any hardware mechanisms are necessary to ensure that the sharing can be done safely, indicate what they are.

.rodata & .text — neither of these needs to be (or should be) writeable. The virtual page mappings in the address space regions for these sections should not be marked as writeable, thus making them read-only. Any attempt to write to a read-only page is trapped by the MMU (page fault), typically resulting in a seg fault signal being delivered to the process.

(b) [5 points] Name a section of a Linux executable where it would *not* be safe for the operating system kernel to map the memory pages containing data loaded from that section into the address spaces of multiple processes. Explain why sharing pages for this section would not be safe.

.data — contains modifiable global variables. We don't want one process's writes to global variables to change the values of global variables in any other process.

(c) [5 points] The x86-64 architecture has (effectively) 48 bit virtual addresses. How much memory would be required to represent the mappings of virtual to physical pages for the entire 48-bit virtual address space if the page tables were a single "flat" array? Assume that each page table entry requires 8 bytes.

Assuming 4K pages (2^{12} bytes)

# pages in 2^{48} byte address space is $2^{48}/2^{12} = 2^{36}$

$$\underbrace{2^{36} \text{ pages} \times 2^3 \frac{\text{bytes}}{\text{page}}}_{\text{size of one page table entry}} = \underbrace{\boxed{2^{39} \text{ bytes}}}_{\text{total size of a "flat" (non-hierarchical) page table array}}$$

REALLY HUGE!

(d) [5 points] On systems with virtual memory, it is possible for the contents of a data cache to be indexed by either virtual or physical addresses. State one advantage and one disadvantage for both indexing a cache by virtual address and indexing a cache by physical address.

index by virtual addresses

pro: cache doesn't need to translate virtual addresses to physical → faster

con: switching address spaces (e.g. because of a process context switch) means the cached data is no longer valid

index by physical addresses

pro: cache contents are still valid after switching to a different virtual address space

con: slower, more complicated

Q4. Threads *20 points*

(a) [5 points] Briefly explain why each thread in a multithreaded program needs its own call stack.

each thread can execute a different computation, with a different sequence of function calls, different local variable values, etc.

(b) [5 points] Is it possible that it would be useful for a program to create multiple threads even if it is running on a single-core CPU? Explain briefly.

Definitely possible! The OS kernel or thread scheduler can (when necessary) suspend the currently-executing thread and resume the execution of a different (currently suspended) thread.

Also, definitely useful, because in many applications, threads are used for <u>concurrency</u>, e.g. a network server handling multiple clients simultaneously.

(c) [10 points] Consider the following `Queue` data type and `queue_try_enqueue` function:

```
struct Queue {
  void *items[MAX];
  int head, tail, count;
  pthread_mutex_t lock;
};

// Try to add item to queue, returns true if successful
// and false if the queue is currently full.
bool queue_try_enqueue(struct Queue *q, void *item) {
  pthread_mutex_lock(&q->lock);

  if (q->count == MAX) { return false; }
  q->items[q->tail] = item;
  q->tail = (q->tail + 1) % MAX;

  pthread_mutex_unlock(&q->lock);

  return true;
}
```

returns with mutex held

Briefly explain the problem with the `queue_try_enqueue` function, and how to fix it.

- problem: early return path doesn't release the mutex, resulting in a deadlock for the next thread attempting to acquire the mutex

- possible solution

```
if (q->count == MAX) {
      pthread_mutex_unlock(&q->lock);
      return false;
}
```

replace

Q5. Networks *20 points*

(a) [5 points] Briefly explain why server programs generally need to use some form of concurrency, such as processes or threads.

server programs typically allow many clients to connect simultaneously. Some form of concurrency is needed to ensure that all clients can make progress independently, i.e., a "slow" client doesn't prevent other clients from making progress

[continued on next page]

(b) [15 points] Complete the following function. It should read one line of text from the file descriptor given as the parameter, and then send back (using the same file descriptor) a line of text of the form

```
Hello, text
```

where *text* is the contents of the line of text read from the file descriptor. Make sure that the function will work correctly if the file descriptor refers to a network socket.

The function should return 1 if successful, and 0 if an error occurs.

main issue is that socket reads & writes may not transfer full amount of data requested

```
int hello_transaction(int fd) {
    char buf[1], message[1024];
    int done_read = 0, count = 0;
    while (!done_read) {
            if (read(fd, buf, 1) < 1) { return 0; }
            if (buf[0] == '\n') { done = 1; }
            else if (count < 1024) {
                    message[count] = buf[0];
                    Count ++;
            }
    }
    char * p = message;
    int remaining = count;
    while (remaining > 0) {
        int sent = write(fd, p, remaining);
        if (sent < 1) { return 0; }
        remaining -= sent;
        p += sent;
    }
    return 1;   // success!
}
```