Please enter your name, Athena login name, and recitation section above. Enter your answers in the spaces provided below. Show your work for potential partial credit. You can use the extra white space and the back of each page for scratch work.
Problem 1. The wonderful and terrible world of Operating Systems (19 points)

(A) (3 points) Let’s start with something simple: From the options below, circle all the ones that correspond to the main responsibilities/functions of operating systems.

1. Abstraction
2. Isolation
3. Branch prediction
4. Hazard resolution
5. Resource management
6. Data encryption
7. Caching

(B) You are managing a RISC-V single CPU system that runs the following application (shown on the next page), which uses two threads. The OS is prioritizing one thread over the other by allocating double the amount of time to it, i.e., thread 1 gets 20ms of time, while thread 2 gets 10ms of time. The OS uses timer interrupts to enforce these CPU time allocations. Thread 1 starts execution first.

Below you see a subset of the instructions each thread executes, as well as how long it takes to execute different regions of the code. You can also see when external events happen, such as network requests and disk I/O requests. Assume “… ” represent independent instructions that do not affect any registers or memory locations you care about. Assume that the lw and sw instructions do not cause exceptions.

You also know that the hardware does not have a hardware implementation for division and instead uses instruction emulation for it.
Before the start of execution: M[0x504] = 0x4

i. (5 points) In the diagram below, indicate the time ranges in which the processor runs in user mode vs supervisor mode by drawing a horizontal line in user or supervisor mode indicating approximate time taken in each mode and diagonal lines indicating transitions from user to supervisor mode and vice versa, as shown in the example below. Approximate timings are ok, but you need to get the number of transitions between modes correct. **Label each transition between user and supervisor mode with its cause as shown in the example below** (you may use NI for network interrupt, and DIO for disk I/O). **Assume that each exception handling takes less than 0.5ms.**

ii. (2 points) The value of x2 at T=30ms is: __________ 4 __________

The value of x2 at T=30ms is: __________ 4 __________

6.191 Fall 2023 - 3 of 21 - Quiz #3 Solutions
(C) (2 points) The programmer does not think it is fair for Thread 1 to get double the time of Thread 2, so they change the CPU time allocations to be 10ms for each thread.

The value of x₂ at T=30ms now is: 8

(D) (1 point) Is the program’s output independent of the CPU time allocations? Circle one.

YES               NO

(E) (2 points) If yes, how should the programmer set the CPU time allocations? If no, how can you resolve this problem without relying on correctly setting the CPU time allocations?

Use semaphores to ensure precedence constraints are satisfied.

(F) (2 points) There seem to be many transitions between user mode and supervisor mode in the execution above. You realize that the network and disk I/O requests do not make a difference to the program’s execution. Given this, are there any optimizations you can perform in terms of when you handle these requests to optimize performance?

Aggregate multiple network and disk requests and handle them together instead of switching back and forth between user and supervisor mode every time.

(G) (2 points) Can you apply the same optimization to exceptions (caused by the program) and to emulated instructions? Circle Yes or No and explain why or why not.

YES               NO

Explanation: Instructions must be executed in order so you cannot aggregate exceptions caused by the program as you can with network and disk I/O exceptions.
Problem 2. Virtual Memory (19 points)

Consider a RISC-V processor that is connected to a memory management unit (MMU) that uses a single level page table to translate 34-bit virtual addresses to 30-bit physical addresses using a page size of $2^{20}$ bytes. Assume all physical pages are currently in use.

The MMU and the page fault handler implement an LRU replacement strategy. **Assume that the dirty bit in the page table only gets updated when a VPN is evicted from the TLB.**

(A) (2 points) Assuming the page table includes the standard dirty and resident bits, specify the width of each page table entry in bits, and the number of entries in the page table.

Size of page table entry in bits: _____12_____

Number of entries in the page table: _____$2^{14}$_____

(B) (2 points) How do these parameters change if the size of physical memory is doubled and the size of each page doubles, but the size of virtual memory remains the same? Use a letter “a” through “e” to indicate how the new value of the parameter compares to the old value of the parameter.

(a) doubled (b) increased by 1 (c) stays the same (d) decreased by 1 (e) halved

Size of page table entry in bits: _____e_____

Number of entries in the page table: _____e_____

Consider the original design with $2^{20}$ bytes per page. The contents of the first 8 entries of the page table are shown below, together with the contents of a 4-entry fully associative TLB (translation lookaside buffer) which uses an LRU replacement policy:

### TLB

<table>
<thead>
<tr>
<th>VPN</th>
<th>V</th>
<th>D</th>
<th>PPN</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x1</td>
<td>1</td>
<td>0</td>
<td>0x21</td>
</tr>
<tr>
<td>0x4</td>
<td>1</td>
<td>1</td>
<td>0x58</td>
</tr>
<tr>
<td>LRU→</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0x3</td>
<td>1</td>
<td>0</td>
<td>0x85</td>
</tr>
<tr>
<td>Next</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0x6</td>
<td>1</td>
<td>1</td>
<td>0x34</td>
</tr>
</tbody>
</table>

### PAGE TABLE

<table>
<thead>
<tr>
<th>VPN</th>
<th>R</th>
<th>D</th>
<th>PPN</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x0</td>
<td>1</td>
<td>1</td>
<td>0x63</td>
</tr>
<tr>
<td>0x1</td>
<td>1</td>
<td>0</td>
<td>0x21</td>
</tr>
<tr>
<td>LRU→</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0x2</td>
<td>1</td>
<td>1</td>
<td>0x49</td>
</tr>
<tr>
<td>0x3</td>
<td>1</td>
<td>0</td>
<td>0x85</td>
</tr>
<tr>
<td>0x4</td>
<td>1</td>
<td>0</td>
<td>0x58</td>
</tr>
<tr>
<td>0x5</td>
<td>0</td>
<td>--</td>
<td>--</td>
</tr>
<tr>
<td>0x6</td>
<td>1</td>
<td>0</td>
<td>0x34</td>
</tr>
<tr>
<td>0x7</td>
<td>1</td>
<td>1</td>
<td>0x4</td>
</tr>
</tbody>
</table>
(C) (6 points) Consider the following sequence of virtual address read memory accesses. For each access, compute its corresponding virtual page number (VPN), physical page number (PPN), and physical address and indicate whether the access causes a TLB miss and/or a page fault. Assume that the access to virtual address 0xABCDE begins with the TLB and Page Table state above. Then 0x510404 is accessed with the resulting TLB and page table and so on.

<table>
<thead>
<tr>
<th>Virtual Address</th>
<th>VPN</th>
<th>PPN</th>
<th>Physical Address</th>
<th>TLB Miss? (Yes/No)</th>
<th>Page Fault (Yes/No)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x0</td>
<td>0x0</td>
<td>0x63</td>
<td>0x63ABCDE</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>0x510404</td>
<td>0x5</td>
<td>0x49</td>
<td>0x4910404</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>0x678F90</td>
<td>0x6</td>
<td>0x34</td>
<td>0x3478F90</td>
<td>No</td>
<td>No</td>
</tr>
</tbody>
</table>

(D)(2 points) Which VPN(s), if any, were evicted from main memory, and which PPN(s), if any, were written back to disk during execution of the three memory accesses from part (C). If there are no pages to list, then enter NONE.

Evicted VPN(s) (0x): _______ 2 _________
Written back PPN(s) (0x): _______ 49 _________

(E) (4 points) Fill in any rows of the TLB and first 8 rows of the page table that were updated while executing the three memory accesses of part (C). Leave all unchanged rows blank. You do not need to update which rows are the LRU and Next LRU.

**TLB**

<table>
<thead>
<tr>
<th>VPN</th>
<th>V</th>
<th>D</th>
<th>PPN</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x0</td>
<td>1</td>
<td>1</td>
<td>0x63</td>
</tr>
<tr>
<td>0x5</td>
<td>1</td>
<td>0</td>
<td>0x49</td>
</tr>
</tbody>
</table>

**PAGE TABLE**

<table>
<thead>
<tr>
<th>VPN</th>
<th>R</th>
<th>D</th>
<th>PPN</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x0</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0x1</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0x2</td>
<td>0</td>
<td>--</td>
<td>--</td>
</tr>
<tr>
<td>0x3</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0x4</td>
<td>1</td>
<td>1</td>
<td>0x58</td>
</tr>
<tr>
<td>0x5</td>
<td>1</td>
<td>0</td>
<td>0x49</td>
</tr>
</tbody>
</table>
(F) (3 points) Now consider a new processor with a 5-bit VPN that is translated to a PPN using the two-level hierarchical page table shown below. The top 3 bits of the VPN are used as the first level index, and the bottom 2 bits of the VPN are used as the second level index. The bottom 8 bits of the virtual address are the page offset. The values listed in the level 2 page tables are the PPNs.

<table>
<thead>
<tr>
<th>Virtual Address</th>
<th>Physical Address</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x545 VPN = 001_01</td>
<td>0x645</td>
</tr>
<tr>
<td>0xFFA VPN = 011_11</td>
<td>0xFDFA</td>
</tr>
<tr>
<td>0x1987 VPN = 110_01</td>
<td>0x587</td>
</tr>
</tbody>
</table>

Translate the following virtual addresses to physical addresses using this two-level hierarchical page table. If a virtual address does not map to a physical address according to the diagram above, then write MISSING.
Problem 3. Parallel Processing (12 points)

Linda Loopy is deciding how to optimize a program that sums up all the elements in a 64x64 element matrix M, which is stored in memory in row-major order. Assume the variable sum is stored somewhere in shared memory. She considers the following two pieces of code:

Program 1

```c
int sum = 0;
for (int j = 0; j < 64; j++)
    for (int i = 0; i < 64; i++)
        sum += M[i][j];
```

Program 2

```c
int sum = 0;
for (int i = 0; i < 64; i++)
    for (int j = 0; j < 64; j++)
        sum += M[i][j];
```

Each program is run on a direct mapped cache with 64 words and a block size of 8 words. The data cache is separate from the instruction cache.

(A)(3 points) To minimize the total number of data cache misses, circle which program Linda should use. Explain why.

**Explanation:**

A cache line holds array elements with the same [i], and consecutive [j]. Since program 2 accesses the elements of a row of M with increasing indices of j this means that for every 8 accesses, you will only get one cache miss when you bring in a new line.

(B)(2 points) During the execution of the code you chose in part (A), how many total data cache misses are there? No need to simplify the expression.

$$64(64)/8 = 512$$

**Number of data cache misses: **$512$
The two programs are repeated above for your convenience.

(C)(4 points) Perform loop unrolling on the inner loop of Program 1 with an unrolling factor of 4 and fill in the new code below.

```c
int sum = 0;
for (int j = 0; j < 64; j++) {
    for (int i = 0; i < 64; i+=4) {
        sum += M[i][j];
        sum += M[i+1][j];
        sum += M[i+2][j];
        sum += M[i+3][j];
    }
}
```

(D)(3 points) Assume we now run Program 2 on a multithreaded processor with 8 threads. What resource(s) do these threads share? Does this cause any problems?

The variable sum is shared by all the threads, so unless we use a semaphore around the updating of sum, we will run into problems updating the value of sum correctly.
Problem 4. Exceptions (16 points)

A RISC-V system with segmentation-based virtual memory is currently running a process A in user space with segment base register = 0x20000 and bound register = 0x20000.

(A) (2 points) Here are 2 virtual addresses. Determine if we will get an out-of-bounds violation or not. If the virtual address is in the bounds, then translate it to a physical address, otherwise write N/A.

i. 0x200
   Out of bounds: YES NO
   Physical Address: __0x20200___

ii. 0x21000
    Out of bounds: YES NO
    Physical Address: _____N/A_____

(B) (4 points) Here is the code for process A and a toy kernel-space handler. Determine which instructions in the process A code trigger an exception. Write a Y in the brackets in the first column for each instruction that causes an exception. Leave all other brackets blank.

```
// process A code
[ ] li a0, 0x40000
[ ] li a1, 0x20
[ ] lw a2, 0x0(a0)
[ ] li a3, 0x200
    loop:
        add a2, a2, a1
        addi a1, a1, -1
        addi a3, a3, 4
        sw a2, 0(a3)
        bnez a1, loop
[ ] addi a3, a3, -0x200
[ ] li a7, 0x13 // syscall number
[ ] mv a0, a3
[ ] ecall
```

```
    // kernel space
    // toy_handler:
    [ ] li a1, 0x20
    [ ] lw a2, 0x0(a0)
    [ ] li a3, 0x200
    loop:
        add a2, a2, a1
        addi a1, a1, -1
        addi a3, a3, 4
        sw a2, 0(a3)
        bnez a1, loop
        addi a4, a4, 1
        csrr a5, mepc
        mret
        addi a5, a5, 4
        csrw mepc, a5
```

...
The program runs on a RISC-V processor with full bypassing and annulment and with exceptions enabled. Assume that exceptions are handled immediately.

Also assume that the toy_handler always returns to the instruction immediately after the instruction that caused the exception. Don’t worry about the actual code in the toy_handler. The code is repeated below for your convenience.

```
// process A code
li a0, 0x40000
li a1, 0x20
lw a2, 0x0(a0)
li a3, 0x200
loop:
   add a2, a2, a1
   addi a1, a1, -1
   addi a3, a3, 4
   sw a2, 0(a3)
   bnez a1, loop
li a7, 0x13 // syscall number
mv a0, a3
ecall
...

// kernel space

toy_handler:
   addi a4, a4, 1
csrr a5, mepc
mret
   addi a5, a5, 4
csrw mepc, a5
```

(C) (5 points) Fill in the pipeline diagram below beginning with the first li instruction (ignore gray cells). No need to show used bypasses, if any. Assume mret behaves like a branch instruction (i.e., next PC is determined in the EXE stage and branches are predicted not taken).
(D)(5 points) Alyssa decides that she no longer wants to do additions in the loop but multiplications. So, she changes the add instruction at the start of the loop to be mul. The updated code is shown below. However, she realizes that the mul instruction is not part of the RISC-V instruction set, so it will be emulated by the operating system. Assume mret behaves like a branch instruction (i.e., next PC is determined in the EXE stage and branches are predicted not taken.

```
// process A code
li a0, 0x40000
li a1, 0x20
lw a2, 0x0(a0)
li a3, 0x200
loop:
    mul a2, a2, a1
    addi a1, a1, -1
    addi a3, a3, 4
    sw a2, 0(a3)
    bnez a1, loop
addi a3, a3, -0x200
li a7, 0x13 // syscall number
mv a0, a3
ecall
...
```

The program has entered the loop and reached steady state. Assume that if any exceptions were encountered that they were handled and that execution resumed. Fill in the pipeline for the cycles 400-408 assuming that the mul instruction is fetched at cycle 400. Ignore the gray cells.

<table>
<thead>
<tr>
<th></th>
<th>400</th>
<th>401</th>
<th>402</th>
<th>403</th>
<th>404</th>
<th>405</th>
<th>406</th>
<th>407</th>
<th>408</th>
</tr>
</thead>
<tbody>
<tr>
<td>IF</td>
<td>mul</td>
<td>addi</td>
<td>addi</td>
<td>csrr</td>
<td>mret</td>
<td>addi</td>
<td>csrw</td>
<td>addi</td>
<td>addi</td>
</tr>
<tr>
<td>DEC</td>
<td></td>
<td>mul</td>
<td>NOP</td>
<td>addi</td>
<td>csrr</td>
<td>mret</td>
<td>addi</td>
<td>NOP</td>
<td>addi</td>
</tr>
<tr>
<td>EXE</td>
<td></td>
<td>NOP</td>
<td>NOP</td>
<td>addi</td>
<td>csrr</td>
<td>mret</td>
<td>NOP</td>
<td>NOP</td>
<td>NOP</td>
</tr>
<tr>
<td>MEM</td>
<td></td>
<td>NOP</td>
<td>NOP</td>
<td>addi</td>
<td>csrr</td>
<td>mret</td>
<td>NOP</td>
<td>NOP</td>
<td>NOP</td>
</tr>
<tr>
<td>WB</td>
<td></td>
<td>NOP</td>
<td>NOP</td>
<td>addi</td>
<td>csrr</td>
<td>mret</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Problem 5. Synchronization (16 points)

Tim is thinking of adding a new option to his donut shop: randomized surprise boxes! Each box will contain four donuts with random flavored frostings and fillings. Tim proposes the following processes to assemble the surprise boxes.

<table>
<thead>
<tr>
<th>ALoop:</th>
<th>BLoop:</th>
<th>CLoop:</th>
<th>DLoop:</th>
</tr>
</thead>
<tbody>
<tr>
<td>Get box</td>
<td>Put dough in frier</td>
<td>Choose random frosting</td>
<td>Choose random filling</td>
</tr>
<tr>
<td>Seal box</td>
<td>Take donut from frier</td>
<td>Take unfrosted donut from box</td>
<td>Take unfilled donut from box</td>
</tr>
<tr>
<td>Goto <strong>ALoop</strong></td>
<td>Place donut in box</td>
<td>Frost donut</td>
<td>Fill donut</td>
</tr>
<tr>
<td></td>
<td>Goto <strong>BLoop</strong></td>
<td>Place donut in box</td>
<td>Place donut in box</td>
</tr>
</tbody>
</table>

(A) (3 points) Tim tries running his current code concurrently, with one thread per Loop, and notices a few strange behaviors. For each of the following outcomes, circle whether it is possible or not:

1. A sealed box contains 0 donuts
   (Possible / Not Possible)

2. A sealed box contains an uncooked donut
   (Possible / Not Possible)

3. A sealed box contains a donut with frosting but no filling
   (Possible / Not Possible)
(B) (5 points) Tim realizes that he also needs to report the total number of calories in each box. So, he declares a variable `total_calories` in shared memory and makes the following modifications:

<table>
<thead>
<tr>
<th>ALoop:</th>
<th>BLoop:</th>
<th>CLoop:</th>
<th>DLoop:</th>
</tr>
</thead>
<tbody>
<tr>
<td>Get box</td>
<td>Put dough in frier</td>
<td>Choose random frosting</td>
<td>Choose random filling</td>
</tr>
<tr>
<td>Set total_calories = 0</td>
<td>Let old_calories = total_calories</td>
<td>C1: Let old_calories = total_calories</td>
<td>D1: Let old_calories = total_calories</td>
</tr>
<tr>
<td></td>
<td>Set total_calories = old_calories + 200</td>
<td>C2: Set total_calories = old_calories + 20</td>
<td>D2: Set total_calories = old_calories + 5</td>
</tr>
<tr>
<td>Print total_calories on box</td>
<td>Take donut from frier</td>
<td>Take unfrosted donut from box</td>
<td>Take unfilled donut from box</td>
</tr>
<tr>
<td></td>
<td>Place donut in box</td>
<td>Frost donut</td>
<td>Fill donut</td>
</tr>
<tr>
<td>Seal box</td>
<td>Goto ALoop</td>
<td>Goto BLoop</td>
<td>Goto CLoop</td>
</tr>
</tbody>
</table>

Once again, Tim notices some strange behavior when he runs these processes without any synchronization. Let’s help him investigate. For each of the following sequences of instructions, report the final value of `total_calories` at the end of the sequence. Assume the value of `total_calories` at the beginning of the sequence is 200 and no other instructions run during these sequences.

- **C1 C2 D1 D2**: `total_calories = 225`
- **C1 D1 C2 D2**: `total_calories = 205`
- **C1 D1 D2 C2**: `total_calories = 220`
- **D1 C1 C2 D2**: `total_calories = 205`
- **D1 C1 D2 C2**: `total_calories = 220`
- **D1 D2 C1 C2**: `total_calories = 225`
(C) (8 points) Tim finally decides to get some help adding synchronization to his processes to fix these bugs. He decides to ask you to help him out.

Tim wants his machine to be able to continuously produce boxes. You can assume the frier is a magic frier that never burns no matter how long donuts are left in it. You must satisfy all of the following constraints:

1. Each box has exactly 4 donuts
2. Each donut has 1 random frosting
3. Each donut has 1 random filling
4. Donut can either be frosted and then filled or filled and then frosted
5. The total_calories printed on the box is correct
6. When BLoop, CLoop, or DLoop attempts to take a donut from the box, there is a valid donut to take

Add semaphores, WAIT, and SIGNAL calls to the code below to satisfy the above constraints, while avoiding deadlocks. You are not allowed to introduce any unnecessary precedence constraints. For full credit, use no more than 5 semaphores. Assume that total_calories lives in shared memory. Another copy is provided at the end of the exam.
We concluded it was not possible to satisfy all of the constraints above using the given code as written without introducing any additional precedence constraints. In particular, it is not possible to satisfy constraint 4 as intended (i.e., donuts can be frosted or filled in either order) while also satisfying constraint 6 as intended. The reason for this is the presence of a race for the donut. Take, for example, the following solution:

Semaphore Initialization:  lock = 1, donut = 0, frosting = 0, filling = 0, done = 0

<table>
<thead>
<tr>
<th>ALoop:</th>
<th>BLoop:</th>
<th>CLoop:</th>
<th>DLoop:</th>
</tr>
</thead>
<tbody>
<tr>
<td>Get box</td>
<td>Put dough in frier</td>
<td>Choose random frosting</td>
<td>Choose random filling</td>
</tr>
<tr>
<td>Set total_calories = 0</td>
<td>wait(donut)</td>
<td>wait(frosting)</td>
<td>wait(lock)</td>
</tr>
<tr>
<td>signal(donut)</td>
<td>wait(lock)</td>
<td>wait(lock)</td>
<td>wait(lock)</td>
</tr>
<tr>
<td>signal(donut)</td>
<td>Let old_calories = total_calories</td>
<td>signal(lock)</td>
<td></td>
</tr>
<tr>
<td>signal(donut)</td>
<td>Set total_calories = old_calories + 200</td>
<td>Take donut from frier</td>
<td></td>
</tr>
<tr>
<td>wait(done)</td>
<td>signal(lock)</td>
<td>Place donut in box</td>
<td></td>
</tr>
<tr>
<td>wait(done)</td>
<td>Frost donut</td>
<td>Place donut in box</td>
<td></td>
</tr>
<tr>
<td>wait(done)</td>
<td>Fill donut</td>
<td>Place donut in box</td>
<td></td>
</tr>
<tr>
<td>wait(done)</td>
<td>Goto ALoop</td>
<td>signal(done)</td>
<td>signal(done)</td>
</tr>
<tr>
<td>wait(done)</td>
<td>Goto BLoop</td>
<td>Goto CLoop</td>
<td>Goto DLoop</td>
</tr>
</tbody>
</table>

This solution is mostly correct, but it admits the following race. Say ALoop signals(donut), and BLoop completes, puts a donut in the box, and calls signal(frosting) and signal(filling). Then, CLoop and DLoop both go through the portion of code protected by lock in either order. At this point, both CLoop and DLoop can take the unfrosted donut from the box. If CLoop takes the donut successfully, leaving no donuts in the box, DLoop can attempt to take from the box when no valid donut exists, violating constraint 6.
There are a number of ways this race could be prevented, each of which introduce additional precedence constraints, change the assumptions of the problem, and/or add an additional semaphore. Some examples are:

- Making the “take-frost-place” and “take-fill-place” sequences atomic, by adding an additional semaphore around it. This means we can only frost/fill one donut at a time but does prevent the race. (Shown below in blue.)

Semaphore Initialization:  

- lock = 1, donut = 0, frosting = 0, filling = 0, done = 0, boxlock = 1

### ALoop:

- Get box
- Set total_calories = 0
- signal(donut)
- signal(donut)
- signal(donut)
- signal(donut)
- wait(donut)
- wait(donut)
- wait(donut)
- wait(donut)
- wait(donut)
- wait(donut)
- Print total_calories on box
- Seal box
- Goto **ALoop**

### BLoop:

- Put dough in frier
- wait(donut)
- wait(lock)
- Let old_calories = total_calories
- Set total_calories = old_calories + 200
- signal(lock)
- Take donut from frier
- Place donut in box
- signal(frosting)
- signal(filling)
- Goto **BLoop**

### CLoop:

- Choose random frosting
- wait(frosting)
- wait(lock)
- Let old_calories = total_calories
- Set total_calories = old_calories + 20
- signal(lock)
- Take unfrosted donut from box
- Frost donut
- Place donut in box
- signal(boxlock)
- signal(done)
- Goto **CLoop**

### DLoop:

- Choose random filling
- wait(filling)
- wait(lock)
- Let old_calories = total_calories
- Set total_calories = old_calories + 5
- signal(lock)
- Take unfilled donut from box
- Fill donut
- Place donut in box
- signal(boxlock)
- signal(done)
- Goto **DLoop**

- Serializing – i.e., choosing an order in which to do CLoop and DLoop.
- Remove constraint 6 and assume that “take unfrosted donut” and “take unfilled donut” block until such a donut is available.
Problem 6. Cache Coherence (18 points)

Queen Frog’s domain has split into four parts: Animal Arcadia, Boston Beehive, Cricket Cove, and Dreamy Dog Drive. In order to count their citizens efficiently, each domain operates one core of a four-core processor system, each with its own cache, which are kept coherent using a snoopy-based, write-invalidate MSI protocol, as shown below. The Head of the Census, Matthew, wants to analyze the behavior of this current system, to see if it is possible to do better.

(A) Each core begins by loading and updating its local copy of its population data, which is stored at an aptly-named label (e.g., pop_A for the population of Animal Arcadia) as in the following pseudocode:

```
call pop_count    // counts the additional population of the area
lw a1, <pop_label> // loads the previous population
add a1, a1, a0    // calculates a sum
sw a1, <pop_label> // stores the updated population
```

The memory accesses interleave as follows:

```
(1) lw a1, pop_A  (2) lw a1, pop_B  (3) lw a1, pop_C  (4) lw a1, pop_D
(5) sw a1, pop_A  (6) sw a1, pop_B  (7) sw a1, pop_C  (8) sw a1, pop_D
```
i. (2 points) Matthew wants to take a closer look at what happens on Core A, in particular. Matthew realizes that, since these memory accesses are independent (i.e., pop_A, pop_B, pop_C, and pop_D are different locations in memory), he only needs to look at the memory access on Core A to understand its behavior. Fill in the following table showing the bus transaction(s) that result from each access, and the state of pop_A’s cache line after each access. Write N/A if there is no bus action. If there are multiple bus transactions, then list them in order.

<table>
<thead>
<tr>
<th>Access</th>
<th>Shared bus transactions</th>
<th>Cache A</th>
</tr>
</thead>
<tbody>
<tr>
<td>Initial state</td>
<td></td>
<td>pop_A: I</td>
</tr>
<tr>
<td>lw a1, pop_A</td>
<td>BusRd</td>
<td>pop_A: S</td>
</tr>
<tr>
<td>sw a1, pop_A</td>
<td>BusRdX</td>
<td>pop_A: M</td>
</tr>
</tbody>
</table>

ii. (2 points) Matthew wonders if switching to an MESI protocol, as shown below, would benefit his processor. If, instead, these four cores had a snoopy-based, write-invalidate MESI protocol, what would be the series of bus transactions and cache line states for pop_A from Core A? Write N/A if there is no bus action. If there are multiple bus transactions, then list them in order.

<table>
<thead>
<tr>
<th>Access</th>
<th>Shared bus transactions</th>
<th>Cache A</th>
</tr>
</thead>
<tbody>
<tr>
<td>Initial state</td>
<td></td>
<td>pop_A: I</td>
</tr>
<tr>
<td>lw a1, pop_A</td>
<td>BusRd</td>
<td>pop_A: E</td>
</tr>
<tr>
<td>sw a1, pop_A</td>
<td>-----</td>
<td>pop_A: M</td>
</tr>
</tbody>
</table>

iii. (3 points) For each protocol, what is the number of each type of bus request across all cores?

<table>
<thead>
<tr>
<th>Protocol</th>
<th># of BusRd</th>
<th># of BusRdX</th>
<th># of BusWB</th>
</tr>
</thead>
<tbody>
<tr>
<td>MSI</td>
<td>4</td>
<td>4</td>
<td>0</td>
</tr>
<tr>
<td>MESI</td>
<td>4</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>
(B) Each core then updates the global census count, located at shared label sum, as in the following pseudocode:

```
lw a2, sum       // gets the current global count
add a2, a2, a0   // updates the population
sw a2, sum       // stores the updated population
```

The memory accesses interleave as follows:

<table>
<thead>
<tr>
<th>Access</th>
<th>Shared bus transactions</th>
<th>Cache A</th>
<th>Cache B</th>
<th>Cache C</th>
<th>Cache D</th>
</tr>
</thead>
<tbody>
<tr>
<td>Initial state</td>
<td></td>
<td>sum: I</td>
<td>sum: I</td>
<td>sum: I</td>
<td>sum: I</td>
</tr>
<tr>
<td>Core A: lw a2, sum</td>
<td>BusRd</td>
<td>sum: S</td>
<td>sum: I</td>
<td>sum: I</td>
<td>sum: I</td>
</tr>
<tr>
<td>Core A: sw a2, sum</td>
<td>BusRdX</td>
<td>sum: M</td>
<td>sum: I</td>
<td>sum: I</td>
<td>sum: I</td>
</tr>
<tr>
<td>Core B: lw a2, sum</td>
<td>BusRd, BusWB</td>
<td>sum: S</td>
<td>sum: S</td>
<td>sum: I</td>
<td>sum: I</td>
</tr>
<tr>
<td>Core C: lw a2, sum</td>
<td>BusRd</td>
<td>sum: S</td>
<td>sum: S</td>
<td>sum: S</td>
<td>sum: I</td>
</tr>
<tr>
<td>Core C: sw a2, sum</td>
<td>BusRdX</td>
<td>sum: I</td>
<td>sum: I</td>
<td>sum: M</td>
<td>sum: I</td>
</tr>
<tr>
<td>Core D: lw a2, sum</td>
<td>BusRd, BusWB</td>
<td>sum: I</td>
<td>sum: I</td>
<td>sum: S</td>
<td>sum: S</td>
</tr>
<tr>
<td>Core B: sw a2, sum</td>
<td>BusRdX</td>
<td>sum: I</td>
<td>sum: M</td>
<td>sum: I</td>
<td>sum: I</td>
</tr>
<tr>
<td>Core D: sw a2, sum</td>
<td>BusRdX, BusWB</td>
<td>sum: I</td>
<td>sum: I</td>
<td>sum: I</td>
<td>sum: M</td>
</tr>
</tbody>
</table>

i. (5 points) Fill in the following table showing the bus transactions that result from each access, and the states for sum’s cache line after each access. Extra copy of table at end of exam.

(Recall that the processor uses an **MSI protocol** for cache coherence.)

<table>
<thead>
<tr>
<th>Protocol</th>
<th># of BusRd</th>
<th># of BusRdX</th>
<th># of BusWB</th>
</tr>
</thead>
<tbody>
<tr>
<td>MSI</td>
<td>4</td>
<td>4</td>
<td>3</td>
</tr>
<tr>
<td>MESI</td>
<td>4</td>
<td>3</td>
<td>3</td>
</tr>
</tbody>
</table>

ii. (3 points) Matthew again wonders if switching to an **MESI protocol** would benefit his processor. If, instead, these four cores had a snoopy-based, write-invalidate MESI protocol, what would be the new number of bus transactions of each type? (A row for the MSI protocol is provided for comparison.) **Only include bus transactions for updating the global sum (i.e., do not include the bus transactions from part (A) of the problem).**
(C) (3 points) Matthew realizes that there’s a problem with this implementation of census counting—two processors could read sum at the same time, and then add just their population count without accounting for the other, before writing it back to sum!

Where can Matthew add semaphores to ensure the final value of sum is calculated correctly—i.e., reflects the sum of the values of $a\theta$ for each processor—while also making sure to block the fewest number of instructions? Define and initialize any semaphores you need and add WAIT and SIGNAL calls below to ensure that sum is calculated correctly. You do not need to worry about adding ecall instructions to the code below.

**Semaphore Initialization:**

```
sum_sem = 1
```

**Pseudocode:**

```
call pop_count // counts the new population of the area
lw a1, <pop_label>  // loads the previous population
add a1, a1, a0     // calculates a sum
sw a1, <pop_label> // stores the updated population
wait(sum_sem)

lw a2, sum         // gets the current global count
add a2, a2, a0     // updates the population
sw a2, sum         // stores the updated population
signal(sum_sem)
```

**END OF QUIZ 3!**