Please write your name and Kerberos on your solutions.

Each subproblem is labelled with (Label: X), (e.g., Label: 1A_1). Please make sure you write down the label of each subproblem together with its solution and circle each answer. So, for example, if your computed value of a0 for problem 1A_1 is 5, you would write:

1A_1: 5

Problem 1. Honor Code Agreement (0 points)

We are using the honor system during this exam, and would like you to accept the following terms of this honor system:

1. You will not download a copy of the exam
2. You will not share the exam with anyone
3. You will not discuss the material on the exam with anyone until after solutions are released
4. You understand that the exam is closed book and that you may only use reference material provided with the exam.

(label: 1A) I will abide by the above terms (circle one): Agree / Do Not Agree
Problem 2. Operating Systems (10 points)

Brian has written a short user program in RISC-V assembly, which computes the very simple mathematical expression \((1 + 1)^2 = 4\). Each instruction is labeled with its (virtual) address:

<table>
<thead>
<tr>
<th>Instruction address</th>
<th>Instruction:</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x1000</td>
<td>addi a0, zero, 1 // set a0 to 1</td>
</tr>
<tr>
<td>0x1004</td>
<td>add a1, a0, a0 // set a1 to a0 + a0</td>
</tr>
<tr>
<td>0x1008</td>
<td>mul a2, a1, a1 // set a2 to a1 * a1</td>
</tr>
<tr>
<td>0x100c</td>
<td>unimp</td>
</tr>
</tbody>
</table>

Note that the function uses the mul instruction from a RISC-V extension. mul rd, rs1, rs2 multiplies the contents of registers rs1 and rs2 together and stores the result in register rd.

Brian starts a user process he calls Process B running the above program on a RISC-V processor, which has an operating system that is also running other user processes and that schedules each user process to run on the CPU for some time.

(A) (4 points) Right after Process B executes the addi instruction but before it executes the add instruction, a timer interrupt occurs and the operating system switches to running a different user process. Answer the following questions about a0 and pc at various times while this occurs; if there is not enough information to determine a particular value, write CAN'T TELL.

(i) What is the value in register a0 when the initial timer interrupt occurs in Brian’s program?

(label: 2A_1) Value of a0, or CAN’T TELL: ___1___

The interrupt doesn’t affect the value of registers.

(ii) The common handler calls an exception handler that tells the common handler to switch to a different user process. What is the value in register a0 immediately after the common handler executes mret to return to a user process other than Process B?

(label: 2A_2) Value of a0, or CAN’T TELL: ___ CAN’T TELL _____

It will be whatever a0 was for the other program when it was interrupted.

(iii) After scheduling other user processes for some time, the operating system schedules Process B again. What is the value of register a0 and of the program counter pc immediately after the common handler executes mret to return to Brian’s process?

(label: 2A_3_a0) Value of a0, or CAN’T TELL: ___1____

The user program’s value of a0 must be preserved.

(label: 2A_3_pc) Value of pc, or CAN’T TELL: ___0x1004____

pc is not affected by the interrupt. The add instruction was never executed.
(B) (6 points) Unfortunately, the RISC-V processor that Brian’s program is running on does not actually implement the mul instruction in hardware. Fortunately, the operating system kernel has code to emulate the mul instruction in software. More precisely:

1. The kernel has a function `asm_mul` written in RISC-V assembly that follows RISC-V calling convention, takes two arguments, multiplies them together, and returns the result. `asm_mul` is implemented using only instructions that are supported by the processor’s hardware.
2. The kernel also has an exception handler that catches illegal mul instructions and emulates them by decoding the instruction, calling `asm_mul` on the values of rs1 and rs2 in the user process, and setting the value of rd in the user process to the result.

As a result, Brian’s process will be able to run successfully. Answer the following questions about a0 and pc at various times while this occurs; if there is not enough information to tell a particular value, write CAN’T TELL.

(i) What is the value of register a0 when the processor attempts to execute the mul instruction in Process B and fails, causing an illegal instruction exception?

```
(label: 2B_1) Value of a0, or CAN’T TELL: _____1_____  
Still unchanged.
```

(ii) Inside the operating system kernel, what is the value of register a0 when the common handler calls the exception handler for illegal instructions?

```
(label: 2B_2) Value of a0, or CAN’T TELL: _____ CAN’T TELL _____ 
There’s no way to know what the exception handler’s first argument is. It might be a process ID, which we definitely do not know.
```

(iii) Inside the operating system kernel, what is the value of register a0 when the exception handler calls `asm_mul`?

```
(label: 2B_3) Value of a0, or CAN’T TELL: _____2_____ 
asmp_mul is called with the values of rs1 and rs2, which are both a1 in the user program, with value 2.
```

(iv) Inside the operating system kernel, what is the value of register a0 when `asm_mul` returns?

```
(label: 2B_4) Value of a0, or CAN’T TELL: _____4_____  
By calling convention, asm_mul returns the product in a0.
```

(v) After the kernel has emulated the multiplication instruction, what is the value of register a0 and of the program counter pc immediately after the common handler executes `mret` to return to Process B?

```
(label: 2B_5_a0) Value of a0, or CAN’T TELL: _____1_____  
a0 must be restored to its former value in the user program since mul doesn’t affect it.
```

```
(label: 2B_5_pc) Value of pc, or CAN’T TELL: _____0x100C_____  
The mul instruction should be skipped since it was emulated.
```
Problem 3. Virtual Memory (26 points)

For the following questions, assume a processor with 64-bit virtual addresses, 40-bit physical addresses and page size of 4096 \( (2^{12}) \) bytes per page. The Page Table of this processor uses an LRU replacement strategy, and handles missing pages using a page fault handler.

(A) (4 points) What is the size of the page table? Assume that each page table entry includes a dirty bit and a resident bit. Specify the number of page table entries and the width of each entry.

(label: 3A_entries) Number of entries in the page table: \( 2^{52} \)

(label: 3A_width) Width of each page table entry (bits): \( 30 \)

(B) (2 points) What is the maximum fraction of virtual memory that can be resident in physical memory at any given time (assuming the page table is not in physical memory)?

(label: 3B) Max fraction of virtual memory that can be resident in physical memory: \( 1/2^{24} \)

(C) (4 points) If we half the size of virtual memory but keep the same physical address length and page size \( (2^{12} \) bytes per page), what effect will the change have on the size of a page table entry and on the number of entries in the page table? 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

(label: 3C_entries) Number of entries in the page table: \( e \)

(label: 3C_width) Width of each page table entry in bits: \( c \)
(D) (8 points) The following program fragment is executed and a record is made of the inputs and outputs of the Memory Management Unit. The record is shown in the table below.

\[
\begin{align*}
\text{sw} & \ x11, \ 0(x10) \\
\text{lw} & \ x11, \ 4(x13) \\
\text{lui} & \ x12, \ 4
\end{align*}
\]

<table>
<thead>
<tr>
<th>Access type</th>
<th>Virtual address</th>
<th>Physical address</th>
</tr>
</thead>
<tbody>
<tr>
<td>Inst. fetch</td>
<td>0x60FF8</td>
<td>0x10FF8</td>
</tr>
<tr>
<td>Data write</td>
<td>0x04600</td>
<td>0x74600</td>
</tr>
<tr>
<td>Inst. fetch</td>
<td>0x60FFC</td>
<td>0x10FFC</td>
</tr>
<tr>
<td>Data read</td>
<td>0x18410</td>
<td>0x169410</td>
</tr>
<tr>
<td>Inst. fetch</td>
<td>0x61000</td>
<td>0x09000</td>
</tr>
</tbody>
</table>

Using information from the program and the table above, please deduce the contents of as many entries as possible in the page table. Assume the original address sizes of 64-bit virtual addresses, 40-bit physical addresses, and page size of 4096 \(2^{12}\) bytes per page. **Assume that pages holding instructions are read-only.**

*(label: 3D) Please make an entry in the table below for each page table entry we learn about after the execution of the program fragment. Note that the table below is not an actual page table, it is just a list of entries from the page table that you can infer from this problem. For each entry provide the VPN, the dirty (D) and resident (R) bits, and the PPN if they are known. If you can’t deduce the value of a field, enter a ‘?’ for that field. You may not need to use all the rows of the table below.*

<table>
<thead>
<tr>
<th>VPN</th>
<th>D</th>
<th>R</th>
<th>PPN</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x60</td>
<td>0</td>
<td>1</td>
<td>0x10</td>
</tr>
<tr>
<td>0x04</td>
<td>1</td>
<td>1</td>
<td>0x74</td>
</tr>
<tr>
<td>0x18</td>
<td>?</td>
<td>1</td>
<td>0x169</td>
</tr>
<tr>
<td>0x61</td>
<td>0</td>
<td>1</td>
<td>0x9</td>
</tr>
</tbody>
</table>
(E) (8 points) At some later point in time, suppose that the contents of the page table and its corresponding fully associative TLB are as shown to the right. As previously mentioned, the page table uses an LRU replacement policy, and the LRU page (shown below) will be chosen for replacement if necessary. For each of these four accesses, compute its corresponding physical address and indicate whether the access causes a TLB miss and/or a page fault. **Assume each access below is independent of the others and starts with the TLB and page table state shown to the right.**

---

**TLB**

<table>
<thead>
<tr>
<th>VPN (tag)</th>
<th>V</th>
<th>D</th>
<th>PPN</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x0</td>
<td>1</td>
<td>0</td>
<td>0x1337</td>
</tr>
<tr>
<td>0x1</td>
<td>0</td>
<td>0</td>
<td>0x7</td>
</tr>
<tr>
<td>0x6</td>
<td>1</td>
<td>1</td>
<td>0x33</td>
</tr>
<tr>
<td>0x3</td>
<td>1</td>
<td>0</td>
<td>0x534</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>0</td>
<td>1</td>
<td>0</td>
<td>0x1337</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>---</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td>0</td>
<td>0x450</td>
</tr>
<tr>
<td>3</td>
<td>1</td>
<td>0</td>
<td>0x534</td>
</tr>
<tr>
<td>4</td>
<td>0</td>
<td>0</td>
<td>---</td>
</tr>
<tr>
<td>5</td>
<td>1</td>
<td>1</td>
<td>0xAB5</td>
</tr>
<tr>
<td>6</td>
<td>1</td>
<td>1</td>
<td>0x33</td>
</tr>
</tbody>
</table>

---

**Fill in table below**

<table>
<thead>
<tr>
<th>Virt Addr</th>
<th>PPN (in hex)</th>
<th>Phys Addr (in hex)</th>
<th>TLB Miss?</th>
<th>Page Fault?</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. 0x6004</td>
<td>0x33</td>
<td>0x33004</td>
<td>Y</td>
<td>N</td>
</tr>
<tr>
<td>2. 0x6030</td>
<td>0x33</td>
<td>0x33030</td>
<td>Y</td>
<td>N</td>
</tr>
<tr>
<td>3. 0x1234</td>
<td>0x53</td>
<td>0x534234</td>
<td>Y</td>
<td>N</td>
</tr>
<tr>
<td>4. 0x2008</td>
<td>0x45</td>
<td>0x450008</td>
<td>Y</td>
<td>N</td>
</tr>
</tbody>
</table>
Problem 4. Pipelined Processors (24 points)

Consider the execution of the following code sequence on a 5-stage pipelined RISC-V processor, which is fully bypassed and has branch annulment. Branch decisions are made in the EXE stage and branches are always predicted not taken. The instruction unimp signals the end of the program.

. = 0x100
L1:
    bgtz x10, done  // assume this branch is not taken
    li  x11, 0x4
L2:
    add  x15, x10, x11
    lw  x17, 0x500(x11)
    addi x11, x11, -4
    sub  x18, x18, x17
    bgez x11, L2  // assume this branch is taken
    or  x16, x16, x18
    addi x10, x10, 4
    j  L1
done:
    unimp

(A) (12 points) (label: 4A) Fill in the pipeline diagram for cycles 200-210, assuming that at cycle 200 the instruction at L1 is fetched. Assume that the bgtz branch is not taken and the bgez branch is taken. Also assume the bgtz instruction does not have any data hazards. Draw arrows indicating each use of bypassing. You can ignore any cells shaded in gray.

(B) (2 points) Which opcode(s), if any, are stalled in the execution of part A?

(label: 4B) Opcode(s) that are stalled or NONE: _______sub, bgez_______
(C) (2 points) How many instructions were annulled in part A?

(label: 4C) Number of instructions annulled: ______2_______

(D) (2 points) Consider a modified processor, P2, which has extra hardware to check if a register is greater than or equal to zero in the decode stage, so branch decisions for \texttt{bgez} can be made in the decode stage. If the same code sequence of part A is now executed on P2, how many instructions would be annulled?

(label: 4D) Number of instructions annulled on P2: ______1_______

(E) (6 points) Now consider a third processor, P3, which is the same as processor P1 except that it \textbf{does not have a bypass path from the EXE to the DEC stage}. It does have the other bypass paths. Redo cycles 200-206 of part (A) using processor P3.

(label: 4E) Fill in the pipeline diagram. Draw arrows indicating each use of bypassing. You can ignore any cells shaded in gray.

<table>
<thead>
<tr>
<th>Cycle</th>
<th>200</th>
<th>201</th>
<th>202</th>
<th>203</th>
<th>204</th>
<th>205</th>
<th>206</th>
</tr>
</thead>
<tbody>
<tr>
<td>IF</td>
<td>bgtz</td>
<td>li</td>
<td>add</td>
<td>lw</td>
<td>lw</td>
<td>addi</td>
<td>sub</td>
</tr>
<tr>
<td>DEC</td>
<td>bgtz</td>
<td>li</td>
<td>add</td>
<td>add</td>
<td>lw</td>
<td>addi</td>
<td></td>
</tr>
<tr>
<td>EXE</td>
<td>bgtz</td>
<td>li</td>
<td>NOP</td>
<td>add</td>
<td>lw</td>
<td></td>
<td></td>
</tr>
<tr>
<td>MEM</td>
<td>bgtz</td>
<td>li</td>
<td>NOP</td>
<td>add</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>WB</td>
<td>bgtz</td>
<td>li</td>
<td>NOP</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Problem 5. Processor Pipeline Performance (20 points)

Ben Bitdiddle and Alyssa P. Hacker are building a five stage RISC-V processor to help grade 6.004 exams. Their pipeline has the standard stages and functionality presented in lecture (IF, DEC, EXE, MEM, and WB). You may assume that instruction and data memories are single-cycle memories with clocked reads and writes.

The following code segment simulates counting the number of correct answers on a question. It loads a student answer, increments the count if its correct, and then loops back to the next student.

```plaintext
... grade_question:
    lw t0, 0(a2)                // load a student's answer
    bne t0, a1, next_student   // check answer; assume bne is NOT TAKEN
    addi t1, t1, 1             // increment num correct
next_student:                  
    addi a2, a2, 4
    blt a2, a3, grade_question // next student; assume blt is ALWAYS TAKEN
next_question:                 
    xor a4, a0, a4
    slli a4, a4, 2
...```

6.004 students are doing well on the exam and are all getting the questions right, so the bne branch is never taken. Also, the processor is in the middle of grading exams, so the blt branch is always taken.

Ben starts out with a fully functional 5-stage RISC-V processor with full bypassing and annulment hardware. Assume branch decisions are made in the EXE stage. Ben’s processor always speculates that nextPC will be PC + 4.

Alyssa thinks she can achieve better performance with a smarter branch predictor to better handle loops. Her processor includes additional hardware to speculate that the branch is not taken (i.e. nextPC = PC + 4) on all forward branches (i.e. the branch target address is greater than the current PC) and that the branch is taken on all backwards branches. More formally:

\[
\text{nextPC} = \begin{cases} 
PC + \text{immB}, & \text{immB} \leq 0 \\
PC + 4, & \text{immB} > 0 
\end{cases}
\]

When the branch is detected in the DEC stage, the hardware will make its prediction based on immB and impact the instruction fetched in that same cycle. The following schematic shows the position of the prediction logic (PRD). The red bars represent the pipeline registers.
The following pipeline diagrams may be useful. You are not required to fill them out.

**Ben’s Pipeline Diagram**

<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
</tr>
</thead>
<tbody>
<tr>
<td>IF</td>
<td>lw</td>
<td>bne</td>
<td>addi</td>
<td>addi</td>
<td>addi</td>
<td>addi</td>
<td>blt</td>
<td>xor</td>
<td>slli</td>
<td>lw</td>
<td></td>
</tr>
<tr>
<td>DEC</td>
<td>lw</td>
<td>bne</td>
<td>bne</td>
<td>bne</td>
<td>addi</td>
<td>addi</td>
<td>blt</td>
<td>xor</td>
<td>NOP</td>
<td></td>
<td></td>
</tr>
<tr>
<td>EXE</td>
<td>lw</td>
<td>NOP</td>
<td>NOP</td>
<td>bne</td>
<td>addi</td>
<td>addi</td>
<td>blt</td>
<td>NOP</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>MEM</td>
<td>lw</td>
<td>NOP</td>
<td>NOP</td>
<td>bne</td>
<td>addi</td>
<td>addi</td>
<td>blt</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>WB</td>
<td>lw</td>
<td>NOP</td>
<td>NOP</td>
<td>bne</td>
<td>addi</td>
<td>addi</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Alyssa’s Pipeline Diagram**

<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
</tr>
</thead>
<tbody>
<tr>
<td>IF</td>
<td>lw</td>
<td>bne</td>
<td>addi</td>
<td>addi</td>
<td>addi</td>
<td>addi</td>
<td>blt</td>
<td>lw</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>DEC</td>
<td>lw</td>
<td>bne</td>
<td>bne</td>
<td>bne</td>
<td>addi</td>
<td>addi</td>
<td>blt</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>EXE</td>
<td>lw</td>
<td>NOP</td>
<td>NOP</td>
<td>bne</td>
<td>addi</td>
<td>addi</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>MEM</td>
<td>lw</td>
<td>NOP</td>
<td>NOP</td>
<td>bne</td>
<td>addi</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>WB</td>
<td>lw</td>
<td>NOP</td>
<td>NOP</td>
<td>bne</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

(A) (4 points) How many cycles does this loop take to execute in each of the processors?

(label: 5A_Ben) Ben’s processor cycles per iteration: ________9________

(label: 5A_Alyssa) Alyssa’s processor cycles per iteration: ________7________

(B) (4 points) Within one iteration of this loop, how many instructions need to be annulled due to incorrect speculation?

(label: 5B_Ben) Number of annulled instructions in Ben’s proc: ________2________

(label: 5B_Alyssa) Number of annulled instructions in Alyssa’s proc: ________0________

(C) (4 points) For this loop, what is the average cycles per instruction (CPI) for each of the processors?

(label: 5C_Ben) Average CPI for Ben’s processor: ________9/5________

(label: 5C_Alyssa) Average CPI for Alyssa’s processor: ________7/5________
The logic for Ben’s processor is shown below with the red bars representing pipeline registers.

![Pipeline Diagram for Ben’s Processor]

The logic for the additional hardware that Alyssa added can be viewed as taking values from the DEC stage and adding additional prediction logic (PRD) as shown below (diagram copied from above).

![Pipeline Diagram for Alyssa’s Processor]

Assume that all registers are ideal (\(t_{\text{SETUP}} = t_{\text{HOLD}} = t_{\text{PD}} = t_{\text{CD}} = 0\) ns) and each pipeline stage/piece of combinational logic has the following propagation delays.

<table>
<thead>
<tr>
<th>Stage</th>
<th>Delay (ns)</th>
</tr>
</thead>
<tbody>
<tr>
<td>PRD</td>
<td>1</td>
</tr>
<tr>
<td>IF</td>
<td>3</td>
</tr>
<tr>
<td>DEC</td>
<td>4</td>
</tr>
<tr>
<td>EXE</td>
<td>7</td>
</tr>
<tr>
<td>MEM</td>
<td>5</td>
</tr>
<tr>
<td>WB</td>
<td>2</td>
</tr>
</tbody>
</table>

(D) (4 points) What is the minimum clock period for each processor?

- (label: 5D_Ben) Minimum clock period for Ben’s processor (ns): __7________
- (label: 5D_Alyssa) Minimum clock period for Alyssa’s processor (ns): __8________

(E) (4 points) Which processor takes less time to execute one iteration of this loop, and how much faster is it?

- (label: 5E_1) Processor that executes loop in less time (Ben’s or Alyssa’s):
  __________Alyssa’s__________

- (label: 5E_2) Number of nanoseconds difference (ns): __7_________
Problem 6. Synchronization (20 points)

MIT has been developing its own online grading system, LearnScope, which it hopes will replace how people grade homework and exams both in MIT and beyond.

LearnScope prides itself on parallelized grading by starting multiple threads running the gradeExams function described below in pseudocode.

Shared Memory:

// exams is an array containing the exams
exams = [exam0, exam1, exam2, ..., exam99];
next_exam = 0;

gradeExams:

    // Get ungraded exam
    exam = exams[next_exam]
    next_exam = next_exam + 1

    // Grade exam
    grade(exam)

    goto gradeExams

(A) (6 points) Suppose two threads, A and B, are running the gradeExams code above without any synchronization. For each of the following failure scenarios, circle whether it is possible to happen or not:

1. A and B start grading the same exam:  
   (label: 6A_1) Possible / Not Possible

2. A gets exam k+1 before B gets exam k  
   (label: 6A_2) Possible / Not Possible

3. An exam is skipped and is never graded by either A or B:  
   (label: 6A_3) Possible / Not Possible

Despite LearnScope’s advanced parallel grading technology, it cannot interface with 6.004’s gradebook module. Instead, Silvina manually enters the grades into the gradebook as LearnScope completes its grading. When the gradeExams function finishes grading an exam, it emails a score report for the student to Silvina.

She will read a score report email, enter the grade into the gradebook, and repeat, as represented by the enterGrades function below. Silvina doesn’t like having more than 10 unread score report emails at a time and also does not like checking her email when she doesn’t have any new messages. Assume that there is more than 1 and fewer than 10 threads running the gradeExams function.
(B) (14 points) **label: 6B** Add semaphores below to enforce these constraints:
1. No two LearnScope threads should ever grade the same exam
2. Silvina should never have more than 10 unread LearnScope score report emails
3. readScoreReportEmail() should not be called until there is an unread score report email from LearnScope
4. After all 100 exams are claimed by LearnScope threads, LearnScope should stop attempting to grade exams
5. As long as there are still unclaimed exams, avoid deadlock
6. Use as few semaphores as possible, and do not add any additional precedence constraints

**Shared Memory:**

```c
exams = [exam0, exam1, exam2, ... , exam99];
next_exam = 0;

// Specify your semaphores and initial values here
semaphore remainingExams = 100;
semaphore nextExamLock = 1;
semaphore emailSpaces = 10;
semaphore unreadEmails = 0;
```

**gradeExams:**

```c
wait(remainingExams)
wait(nextExamLock)
// Get ungraded exam
exam = exams[next_exam]
next_exam = next_exam + 1
signal(nextExamLock)

// Grade exam
grade(exam)
wait(emailSpaces)
emailScoreReport(exam)
signal(unreadEmails)

goto gradeExams
```

**enterGrades:**

```c
wait(unreadEmails)
readScoreReportEmail()
signal(emailSpaces)

enterGrade()

goto enterGrades
```

END OF QUIZ 3!