Elevator Operating System

You have been promoted to assistant (to the) CEO at GATE Co. The CEO was very happy with your two way set associative cache, and now has a new project for you: overhauling the Elevator’s Operating System.

Currently the elevator microcontrollers only run one simple program so they have no need for an operating system, but the CEO has some big plans for the future of GATE Co’s elevators, so he wants you to make a simple operating system for the Elevator that will be able to support multiprogramming.

1 Legacy Code and Printing

*This part uses the code in the PartI subfolder.*

Each elevator produced by GATE Co. uses a software control algorithm that must be run periodically (about every 100 ms). Currently the control algorithm is written in assembly, and it uses a loop to implement the 100 ms delay. There is a function `print_string` (implemented in c in the file `print.c`) intended to print debugging information from the controller, but the current implementation uses the incomplete function `print_char` (found in `controller.user.S`). As a result, when this controller is run, nothing prints.

**Exercise 1 (5 Points):** Implement `print_char` as a function in assembly in `controller.user.S`. `print_char` should take in one argument (as `a0`) and it should print it as a character using one of the memory mapped IO devices mentioned in the appendix in Section 4.5 of this handout. To compile the code, run `make` within the PartI folder (you will have to run `make gcc` the first time). To run the code, use the `run.py` python script by running `python3 run.py`. This script runs the simulator for 50,000 instructions. If you run the script, it should print “Computing elevator control” 9 times before hitting the instruction limit.

*Hint:* If you need to debug your code, the disassembled dump file can be found in `build/controller.user.vmh`. You can edit the `run.py` script to output whatever processor state you desire for debugging. To see what state you can access from the simulator, you can view the code for the simulator in `Riscv.py`.

*Note:* For safety you should commit your changes and push before continuing.

2 Virtual Memory and System Calls

*This part uses the code in the PartII subfolder.*

*Note: The appendix in Sections 4 and 5 will help you understand the details of this lab’s processor.*

**Named Variables in Assembly** Up to this point, when programming in assembly you have not used any named data. You only dealt with data passed in as part of a function call. In this program you will have to deal with named variables. Please review Section 6.2 to see how this is done in RISC-V assembly.

**Virtual Memory** This processor uses base and bound virtual memory for user processes. The base is stored in the `vmbase` CSR, and the bound is hard-coded to 0x10000 for all user processes. If you need to change the `vmbase` CSR, you can use the `set_vmbase` helper function. The initial virtual memory setup has already been handled for you in `kernel.S`, so you will not have to change `vmbase` until PartIII.
**Handler**  In a typical system, the first thing a trap handler must do is save all the registers somewhere in memory. In recitation, we just dumped the registers to the same stack as the running process because that was the easiest thing to do at the time. Unfortunately when you add virtual memory, dumping registers there is no longer an easy option.

**Discussion Question 1 (5 Points):** Why does virtual memory make it harder to dump registers to the running process’s stack from an exception handler?

In this lab we will dump the registers in a special location per process. The named variable `process1regs` contains the address of the chunk of memory for dumping registers for the controller.

**Exercise 2 (25 Points):** Implement the trap handler in `handler.S`. It should:

- Save all the registers starting at the address stored in the named variable `process1regs`. To find the address for saving register x, read the value of the named variable `process1regs` and add 4*n.
- Set the stack pointer to make a kernel stack grow starting at the address in named variable `kernelSP`.
- Call `ih_dispatcher` with the proper arguments (`ih_dispatcher` is defined in `dispatcher.c`).
- `ih_dispatcher` will return the next address to jump to in the user program. Write that address in `mepc`.
- Restore the registers from the address stored in the named variable `process1regs`.
- Return to user mode where the elevator controller program was.

The handler can be compiled using `make`. The handler can be tested independently using the testing script in `TestHandler.py`. This script uses the simulator to simulate an exception with the processor in a specific state. The test executes through the entire exception handler and makes sure the final state of the processor is correct when the `mret` instruction is reached. To run this test program, run `python3 TestHandler.py`.

**Discussion Question 2 (5 Points):** Why do we need to change the stack pointer in the interrupt handler? (Instead of keeping `sp` unchanged)

**System Calls**  Now that the elevator controller is using virtual memory, the elevator controller can no longer access memory mapped IO. We need our machine mode program to do all of the memory-mapped IO for the user program. In order for the user program to ask for the machine mode program to perform some memory-mapped IO on its behalf, the user program must make a system call using the `ecall` instruction. Please review Section 5.1 for information about system calls and the `ecall` instruction.

**Exercise 3 (10 Points):** Implement the system call for printing a string by filling in the assembly for the `syscall_printstring_asm` function in `syscall.S`. Be careful about the address where the string that you need to print is. You will need to perform the virtual address translation in software to pass a physical address to the `print_string` function. You can use the named variable `process1base` to get the base address for the controller process.

**Exercise 4 (5 Points):** Make use of this system call in the code of `controller.user.S`. Make sure to follow the convention specified in Section 5.1.

*Note: At this point, you can build your code using `make` and run the elevator controller using `python3 run.py`. You should see the message “Computing elevator control” printed 7 times. It is printed fewer times than before because (1) running a system call has a higher overhead than a normal function call, and (2) now that the program is running in user mode, there are now timer interrupts interrupting the program.*

---

1 You can use the mscratch CSR as a scratch register if needed, see the Appendix.
3 Multiprogramming (Optional)

This part uses the code in the PartIII subfolder. This part is now optional. Completing parts 1 and 2 is sufficient to earn full credit. This part can be used to satisfy some of the points of the design project.

The CEO's niece is really into cryptocurrencies, so the CEO has decided that GATE Co. should launch its own cryptocurrency: Elevator Coin (EVC). The CEO has realized that GateCo’s controllers could be used to mine EVC, as they are unused most of the time. For example:

- When nobody is using the elevator.
- When someone is using the elevator, the controller computes what to do in hundreds of microseconds, and doesn’t have to compute a new control output for 100 ms.

The CEO got a junior engineer to develop the cryptocurrency, and another junior engineer to figure out how to mine elevator coins while nobody is using the elevator. It is up to you, the CEO’s most trusted employee, to figure out how to mine elevator coins while the elevator is in use.

Context Switch during Sleep

To implement this functionality, you decide to modify the current sleep function. Instead of busy waiting (executing a loop that does nothing), you decide to use a system call to allow the kernel (the machine mode program) to switch to another process (the Elevator Coin miner). To do this, when the *deep_sleep* function is called in *controller.user.S*, the program should:

- Trigger an exception as part of the system call for sleep.
- Save the state of the elevator controller program and register for how long it is acceptable to sleep.
- Switch to the mining program.
- Every few milliseconds, the processor gets a **timer interrupt**.
  - While the current time is less than the wakeup time, remain in the mining process.
  - Once the current time is higher than the wakeup time, switch back to the elevator controller.

Note that there are three contexts here:

- The elevator controller program (process 1).
- The miner program (process 2).
- The handler for the system call and timer interrupt (kernel).

The kernel handles the scheduling of process 1 and process 2. Every time the kernel deschedules a program, the kernel saves the program’s state. Specifically, the kernel maintains the following state:

- The kernel knows which process was previously running by the named variable *processActivated*.
- The kernel should save the registers at the address stored in named variables *process1regs* or *process2regs* (depending on which process was active).
- The kernel should save the pc where we stopped in named variables *process1pc* or *process2pc*.

Note that when you are writing a handler that is intending to switch processes, you must change the value of the named variable *processActivated* to match the newly-scheduled process.

**Exercise 5 (10 Points):** In *handler.S*, implement this new trap handler that saves the current state of the interrupted program in the appropriate area in memory, and restores the context of the correct process when exiting the handler. This includes writing *mepc* to the correct named variable for the process (*process1pc* or *process2pc*).
Exercise 6 (10 Points): Implement sleep as a system call that switches to the Elevator Coin miner when the elevator controller is sleeping. Part of the system call implementation is already provided in dispatcher.c. It is your responsibility to fill in the missing function syscall_sleep_asm in syscall.S. Relevant details for this function can be found in comments in syscall.S.

Exercise 7 (10 Points): Implement the part of the interrupt handler that is called when the sleep timer is up (function interrupt_timer_asm in syscall.s). This should handle the context switch back to the controller process. Relevant details for this function can be found in comments in syscall.S.

Exercise 8 (5 Points): Copy and adjust your implementation for the system call for printing a string. This is done by filling in the assembly for the syscall_printstring_asm function in syscall.S. Now that there are two user processes with different base addresses, you need to modify your scheme for performing the virtual address translation.

Exercise 9 (5 Points): Implement the missing sleep system call in the deep_sleep function in controller.user.S. You should take the first argument passed to deep_sleep and pass it to the sleep system call. Also reimplement the missing print string system call that you added in PartII.

Note: You can build all your code for this part by running make. You can run a unit test to verify a couple of invariants on your handler by running python3 TestHandler.py. You can run the full elevator controller including switching to the Elevator Coin miner program by running python3 run.py. You should see the message “Computing elevator control” printed 8 times with the symbol $ printed between each message denoting the completion of an iteration of elevator coin mining.

It appears that the junior engineer in charge of idle EVC mining is having trouble, and they came to you for help.

The junior engineer’s team was checking if the button was pressed each timer interrupt, but due to their suspect software implementation of button debouncing, their button checking routine took longer than the period between timer interrupts, and they weren’t left with any time for EVC mining.

Their second attempt was to check if the button was pressed every 100th timer interrupt, but customers started complaining about the elevator not responding to button pushes.

Discussion Question 3 (5 Points): Explain to the junior engineer how they can use an external interrupt to get the elevator to respond instantaneously to button presses while still mining EVC almost all of the idle time.
4 Hardware Appendix

4.1 Privilege Mode Support

The RISC-V processor used in this lab has two privilege modes: Machine mode and User mode. In general, Machine mode is the highest privilege mode in RISC-V processors and User mode is the lowest. Machine mode can access and modify all aspects of the machine’s state, but User mode is only allowed access to a subset of the processor’s state - usually the subset owned by the program.

Our processor makes two major simplifications to the standard privilege levels in RISC-V:

1. We only allow exceptions and interrupts from User mode. An exception in machine mode causes the simulator to halt, and an interrupt in machine mode is ignored until the processor switches back to user mode. This simplifies the CSR state required to support traps, and allows us to always return to user mode on mret. Standard RISC-V processors allow exceptions from any mode and allow interrupts whenever they are enabled. There is an mstatus CSR that holds additional information about previous privilege modes, global interrupt enables, and other things to support this.

2. User mode always uses base and bound virtual memory. Standard RISC-V processors have a vmmode field in some CSR that can turn on and off virtual memory for User mode, and base and bound-based virtual memory is not one of the legal settings for that field.

In our processor, Machine mode and User mode have the following characteristics (some of these are simplified for the course and are not RISC-V standards):

### Machine Mode

- All addresses are physical addresses.
- Can read and write all CSRs using csrr (CSR Read), csrw (CSR Write), and csrrw (CSR Read and Write) assembly instructions (csrr and csrw are pseudo instructions).
- Can’t use System Calls (ecall instruction).
- Handles all exceptions and interrupts.
- Interrupts are disabled.

### User Mode

- All addresses are virtual addresses (using Base and Bound).
- Can’t access any CSRs.
- Must use System Calls (ecall instruction) to ask the machine mode to read or write part of the processor it doesn’t have access to.
- All exceptions and interrupts return to User mode (using mret instruction).
- Interrupts are enabled.

4.2 Control and Status Registers

The following CSRs are implemented in this processor. To read or write these registers, use the instructions introduced in Section 6.1.

**mepc - Exception PC**

When an instruction causes an exception, the pc of the faulting instruction is stored in this CSR. When an interrupt occurs, the pc of the next instruction is stored in this CSR. When the mret instruction happens, the value in this CSR is written to the pc.
mcause - Trap Cause

When an exception or interrupt causes a trap to happen, the cause of the trap (either the exception code or the interrupt code) is written to this CSR. The following are the only trap causes you need for this lab:

- 0x8 - ecall instruction from User mode
- 0x80000007 - Timer Interrupt

All other causes are unexpected traps and will cause the simulator to halt.

mtvec - Trap Vector

This CSR holds the address of the trap handler. When an exception or interrupt causes a trap to happen, the pc is set to the value in this CSR.

mscratch - Scratch Register

This is an extra register for Machine mode to use during trap handlers to make saving registers easier.

vmbase - Virtual Memory Base

This is a custom CSR added for this lab to support base and bound virtual memory. This CSR holds the base address used for this virtual memory scheme. The bound is hardwired to 0x1000 in the simulator. The 12-bit index for this CSR is 0x7C0.

4.3 Trap Summary

When an exception or interrupt causes a trap in user mode, the current PC is written to mepc, the code corresponding to the cause of the trap is written to mcause, the PC is set to the value stored in mtvec, and the privilege mode is set to Machine mode.

When the trap handler executes the mret instruction to return from the trap handler, the PC is set to the value in mepc, and the privilege mode is set to User mode.

4.4 Timer Interrupt

The RISC-V simulator has a timer interrupt that is configured to fire every 200 instructions. In our processor, Machine mode ignores interrupts, so if the 200 instruction limit is reached during machine mode, the interrupt will not fire until the processor returns to User mode. When the timer interrupt causes a trap, the mcause CSR is set to 0x80000007.

4.5 Memory Mapped Input/Output

This processor uses 4 memory mapped I/O locations.

0x40000000 - putchar

Writing a 32-bit value to this address causes the ASCII character corresponding to the value to be printed to the screen. For example, writing 65 to address 0x40000000 prints A on the screen.

0x40000004 - putint

Writing a 32-bit value to this address causes the value to be printed to the screen in decimal. For example, writing 65 to address 0x40000004 prints 65 on the screen.
Writing a 32-bit value to this address causes the value to be printed to the screen in hexadecimal. For example, writing 65 to address 0x40000004 prints 41 on the screen.

Writing a 32-bit value to this address causes the simulator to stop. If the value is 0, PASSED is printed to the screen. Otherwise FAILED is printed to the screen, followed by the value written to the address in decimal. For example, writing 65 to address 0x40000004 prints FAILED 65 on the screen and causes the simulator to halt.

5 Software Appendix

5.1 System Calls

System calls are similar to function calls, but they are call initiated by the ecall instruction, handled by a higher-privileged operating system, and have a different calling convention.

When user mode reaches an ecall instruction, an exception is triggered with mcause = 0x8. The software running in Machine mode should look at the values previously stored in argument registers a0 to a7 to determine what the system call should do. By convention, a7 determines the type of system call like 0x10 for print_char or 0x20 for sleep. Registers a0 through a6 hold the arguments for the specific system call. When the system call returns, the return value (if any) will be stored in a0.

Unlike normal function calls, system calls ensure that all registers are callee-saved except for a0.

Below are all the system call types used in this lab:

0x10 - print_char
Prints the value stored in a0 as an ASCII character.

0x11 - print_int
Prints the value stored in a0 as a decimal value.

0x12 - print_hex
Prints the value stored in a0 as a hexadecimal value.

0x13 - print_string
Prints the string at the address specified by a0.

0x20 - sleep
Causes the process to sleep for a0 timer interrupts.

0x30 - exit
Causes the process to exit with the specified exit code in a0.
6 Assembly Appendix

6.1 Reading and Writing CSRs in Assembly

```assembly
csr rd, <csr_name>
```
Copies the value of the specified CSR into rd.

```assembly
csrw <csr_name>, rs1
```
Writes the value from rs1 into the specified CSR.

```assembly
csrrw rd, <csr_name>, rs1
```
Copies the value of the specified CSR into rd and writes the value from rs1 into the specified CSR.

6.2 Using Named Variables in Assembly

So far you have used labels for targets of branches and jumps. Labels are also used for marking locations in memory where variables are stored. In the example below, the program has two named variables: `myVar1` and `myVar2`. When the program is first loaded into memory, the variable `myVar1` has the value 17, and the variable `myVar2` has the value 0.

**Pseudoinstructions:**

In assembly, labels matching the name of the variable are used to mark the position in memory were the variable is stored. When using labels to mark named variables in memory, there are three new pseudo instructions you can use:

```assembly
la rd, <label>
```
Load the address of the specified label into rd. If the label is the name of a variable, it gets the address of the variable, not the value of it. This is a pseudoinstruction that expands into a sequence of instructions similar to the li pseudo instruction.

```assembly
lw rd, <label>
```
Loads the word stored at the address of the specified label into rd. If the label is the name of a variable, it reads the value of the variable from memory. This is a pseudoinstruction that expands into something similar to `la rd, <label> ; lw rd, 0(rd)`.

```assembly
sw rs1, <label>, rs2 (not recommended)
```
*This pseudoinstruction is not recommended since it writes to rs2 too.* Takes the word stored in rs1 and stores it in memory at the address of the specified label, using rs2 as a temporary register. If the label is the name of a variable, it writes the value in rs1 to that variable. Notice that storing a word into a specified label can’t be done without using an extra temporary register, so this pseudo instruction requires an extra temporary register as a result. Also, the temporary register used by this instruction must be different than the register containing the data to write to memory. This expands into something similar to `la rs2, <label> ; sw rs1, 0(rs2)`.

**Example:**

```plaintext
. = 0
lw t1, myVar1
# loads the value of myVar1 into t1
# t1 now contains 17
```
la t2, myVar2
# loads the address of myVar2 into t2
# t2 now contains 0x1800

sw t1, 0(t2)
# writes t1 into myVar2
# myVar2 now contains 17

# The above two instructions could be replaced with
# the following pseudo instruction, but it is not
# recommended:
# sw t1, myVar2, t2

.my = 0x1000
myVar1:
    .word 17

.my = 0x1800
myVar2:
    .word 0