CS 4348/5348 Operating Systems -- Project 1

Project 1 requires the background knowledge in CS 3377 (C/C++ Programming in a UNIX Environment) and CS2340 (Computer Architecture). We will review the materials needed in CS2340 in the first lecture unit.

The project focuses on simulating some basic computer components, CPU and memory, for executing user programs. Some operating system functionalities required for user program execution (very simple version) are also to be implemented. At the end of programming the system, you need to also learn how to compile and execute code in Unix using some Unix scripts and features.

1 The Simulated Computer Hardware

You need to create 2 code files to simulate some computer components (hardware). At this point, the simulated computer units include CPU and memory to be implemented by **cpu.c** and **memory.c**.

1.1 Simulated Memory

In memory.c, you can define an array **Mem** of size M (M is the number of memory words and each word is of type integer). M will be read from the **config.sys** file which consists of system configuration parameters (system parameter readings will be centralized in computer.c). The functions in memory.c are specified as follows.

mem_init(M)	Initialize memory, including allocating the simulated memory array Mem of M words
mem_read()	copy Mem[MAR] to MBR
mem_write()	copy MBR to Mem[MAR]

Memory read/write are performed on registers, so there are no parameters for the read/write functions. The registers are part of the CPU but should be visible to some other code files (including memory.c), thus, you can define them in **computer.h**. Or, you can redefine the data structures using **extern** definitions.

1.2 Simulated CPU

CPU perform arithmetic logical actions (by ALU) on registers. The micro-instructions for each instruction represent the data movements and the ALU actions happening on the hardware. In cpu.c, you need to define a set of common registers (best is to use struct for the registers to facilitate context switch later), including **PC, IR0, IR1, AC, MAR, MBR, and Base**).

To make it simple, we assume that each instruction consists of 2 words (in real systems, most of the instructions are one word), where the first word contains the opcode and the second the operand. The operand can be a memory address (m-addr) or a constant or 0 (null). The list of simulated instructions is given in the following table.

OP Code	Operand	System actions
1 (load)	constant	Load the constant to AC
2 (load)	m-addr	load Mem[m-addr] into AC
3 (add)	m-addr	load Mem[m-addr] into MBR, add MBR to AC
4 (mul)	m-addr	Same as above, except that add becomes multiply
5 (store)	m-addr	Store AC to Mem[m-addr]
6 (ifgo)	m-addr	If AC > 0 then go to the address given in Mem[m-addr] Otherwise, continue to the next instruction
7 (print)	Null	Print the value in AC
8 (sleep)	Time	Sleep for the given “time” in microseconds, which is the operand
9 (shell)	Code	Execute the shell command according to code (elaborated later)
0 (exit)	Null	End of the current program, null is 0 and is unused

The functions you need to implement for cpu.c are specified as follows.

cpu_fetech_instruction()	Read the 2 instruction words from memory, the addresses are computed from PC

cpu_execute_instruction()	For each instruction code, perform the simulated hardware operations
int cpu_mem_address (int m-addr)	Compute the memory address to be accessed and put it in MAR. The input is the PC or the operand (m-addr) of an instruction.
cpu_operation()	Loop for executing instructions, starting from 0 till the exit instruction

As discussed in the lecture, in each instruction cycle, CPU fetches instructions and execute them. Then CPU goes to check the interrupt vector, which is omitted in our system. For cpu_fetch_instruction, CPU reads 2 instruction words from memory to the instruction registers, IR0 and IR1, where IR0 is for the opcode and IR1 is for the operand. The address for fetching instruction is computed from the program counter PC. Then CPU executes the instruction by performing the data movements and ALU computation following the micro-instructions. If the instruction involves data (i.e., there is the m-addr parameter), then the CPU reads data from memory to complete the instruction. Do not forget to properly set the program counter PC after executing the instructions ("ifgo" instruction will need special PC update).

Memory may be used by multiple programs. So, we cannot store all the programs from address 0. At the same time, all the addresses (PC or m-addr in the instructions) are relative to the base address of the program. Thus, for fetching instructions and data, the CPU needs to compute the actual memory address in the physical memory. The base register, **Base**, stores the starting address of the program in the memory. Thus, the physical memory address should be **Base** (base register) **+ m-addr** (PC or the operand).

The shell commands (9) will be elaborated later. Function cpu_operation simply calls cpu_fetch_instruction and cpu_execute_instruction in a loop and the loop terminates when the exit instruction is executed. The loop control will be modified later when we introduce CPU scheduling.

2 User Program

Your simulated computer executes the user programs. The input user program file contains $N_{code} + N_{data} + 1$ lines. The first line contains two integers, number of instructions N_{code} and number of data N_{data}. Following the first line are N_{code} lines of instructions, where each line contains two integers, the opcode and the operand. Following the instructions are N_{data} lines of data, and each line of data is a single integer.

We will provide a few user programs to test your code. However, when TA tests your system, a different set of user input programs may be used. Here is an example input user program (the comments are just for elaboration, and will not appear in the program file).

```
9 3
2 18   // load a
3 19   // add b
5 18   // store a
9 2   // dump registers
9 3   // dump memory
2 20   // load c
3 18   // add a
7 0   // print
0 0   // exit
10
5
20
```

In this input program, there are 9 instructions and 3 data, a total of 21 integers (N_{code} and N_{data} are not part of the program, they are just for convenience), which are to be loaded into memory. The three data values are 10, 5, and 20 (a, b, and c in the comments).

3 Operating System

For a computer to function, we need to have some operating system functionalities on top of the hardware. Here we consider a very simple OS, including **shell.c** for command processing, **load.c** for loading a program into memory, and **computer.c** for system initialization and activation.

3.1 Loader

You need to write a loader program **load.c** to load the user program to your simulated memory. The loader copies the user program into memory, word by word, starting from the memory address given by the base register **Base**. The user program file name and Base will be passed as parameters to the loader. The functions for load.c are specified as follows.

load_prog(char *fname, int base)	Open the user program file with the file name "fname". Copy the program to memory starting from memory address "base".
load_finish(File *f)	Clean up and close the program file

Consider the example user program given earlier. Also, assume that Base = 8. After loading the program, the memory contents will be as follows. The leftmost column gives the memory address of the first memory word in the row. The light orange words are the three data in the program. The light brown words are the data addresses, m-addr, for the instructions, which are relative to the Base of the program

0	-	-	-	-	-	-	-	-
8	2	18	3	19	5	18	9	2
16	9	3	2	20	3	18	7	0
24	0	0	10	5	20			

3.2 Shell

The computer generally interfaces with the users via command line inputs. In Unix, this is realized by a shell process. In Project 1, we only want to implement shell commands that allow us to test your code. Also, in Project 1, we embed the shell commands in instructions. Later we will separate shell commands from instructions. The specific shell commands are specified in the following.

Action code	System actions
2 (register)	Dump the values of all registers
3 (memory)	Dump the content of the entire memory

You need to implement the shell command processing in **shell.c**. The functions include:

shell_init()	Initialize the shell module
shell_print_registers()	Print out all the registers on the screen
shell_print_memory()	Print out all the words in memory in integer form on the screen
shell_command (int cmd)	For each shell command code, call the corresponding functions. Input cmd is the command code.

When executing instructions in cpu.c, if the instruction is shell (9), then CPU calls shell_command and pass the command code for shell processing. Function shell_init may be empty for Project 1.

3.3 Computer

The program **computer.c** initializes and activates the overall system. Some of the management tasks will be performed in computer.c for now, but may be moved to other modules later. The major functions you need to implement for computer.c are listed in the following table.

process_init_PCB()	Create a PCB for the user program and initialize it.
process_set_registers()	Copy the PCB context data to the registers

The functions defined above are extremely simple for now. They are for preparing for the future projects. The detailed steps in the main function are specified as follow:

- Reads the memory size M from configu.sys and calls memory for initialization.
  File config.sys is a text file containing a single integer M (e.g., 128).
- Invokes the initialization functions in other modules.

- Prompts on the screen to get an input file name and memory Base (of the user program). The following is a sample prompt:
  Input Program File and Base> prog1 8
- Calls loader to load the program into the proper memory locations.
- Calls process_init_PCB to create a PCB for the program and initialize some fields in PCB.
- Calls process_set_registers to load the registers from PCB
- Calls CPU to execute the instructions in the user program. You only need to consider a single user program (no need to loop).
- Calls termination functions to clean up the system.

4 Compiling and Executing Programs in Unix

You need to prepare a **makefile** for compiling the program comprised of "cpu.c", "memory.c", "load.c", "shell.c" and "computer.c" (or .cpp). The generated executable should be named as "**computer.exe**".

Then, write a shell script "**execute**" to compile the program using the "**make**" command and execute computer.exe with an input file "**comp.in**" (redirect comp.in to the executable). In your script, the execution of computer.exe should be done twice. In the first invocation of computer.exe, let the output be redirected to "**comp.out**". In the second invocation of computer.exe, **pipe the output to the "wc" command**.

Note that you should not use #include to include any C code files. You should compile each C code file into an object code file (".o") separately and then link the ".o" files together into the executable.

5 Project Submission

You need to submit your program **before** midnight (11:59pm) of the due date (check the web page for the due date). Use UTD elearning system for submission: "elearning.utdallas.edu".

Your submission should include the following:

- All source code files making up your solutions to this assignment.
  - "cpu.c", "memory.c" (or .cpp)
  - "load.c", "shell.c" and "computer.c" (or .cpp)
  - "computer.h" if there is one
- "makefile" and "execute" script
- A "readme" file
  - Indicate how to compile and run your programs in case it deviates from the specification
  - If you did not finish the project, you need to specify which part(s) you have completed and which part(s) are missing
- You can zip or tar all your submission files together and then submit it on elearning