Laboratory of Electrical Engineering and Computer Science-II

Software Experiment 2 - theme 1

Name : Sihanern Thitisan
Student ID : 1TE19250T
Department : Electrical Engineering and Computer Science

Submission Date : 2022/12/20

Theme 1: Porting of C language library

1.1 Explanation of theme 1

In this experiment, we are required to port functionalities from standard C library to make them work with the CPU board in Motorola68k assembly language. Specifically, the `printf()` and `scanf()` functions are required to be ported.

1.2 Role of theme 1 in the whole project

In theme 1, we focus on making the `printf()` and `scanf()` functions work properly on the CPU board. The `printf()` and `scanf()` functions are one of the most important functions in simple C program. They are used to print and read data from the user. This will ensure that the C language programs in follow up themes will be able to use these functions correctly.

1.3 Program Explanation

1.3.1 inbyte() function

When `scanf()` function is called in a C language program, it will call `read()` function in another C language program. Then, `read()` function will call `inbyte()` function in the assembly program *inchrw.s*. Inside the `inbyte()` function, `GETSTRING` is called using `TRAP #0` instruction to obtain the character input.

`inbyte()` function is created in *inchrw.s* file. Its explanation is written in comments of Listing 1.

Listing 1: *inchrw.s* file

```
.global inbyte                  | Export inbyte function to other files

.text
.even

inbyte:
    movem.l %d1-%d3, -(%sp)     | Save registers that will be used
inbyte_start:
    move.l  #1, %d0             | System call number for GETSTRING
    move.l  #0, %d1             | Serial port number = 0
    move.l  #BUF_INBYTE, %d2    | Buffer address
    move.l  #1, %d3             | Buffer size = 1 byte
    trap    #0                  | Initiate system call to GETSTRING

    cmpi.b  #1, %d0             | Check if GETSTRING was successful (1 byte read)
    bne     inbyte_start        | If not, try again

    clr.l   %d0
    move.b  BUF_INBYTE, %d0     | Move byte from buffer to %d0

    movem.l (%sp)+, %d1-%d3     | Restore registers
    rts                         | Return from function

.section .bss
BUF_INBYTE: .ds.b 1             | 1 byte buffer for inbyte function
    .even
```

1.3.2 outbyte() function

When `printf()` function is called in a C language program, it will call `write()` function in another C language program. Then, `write()` function will call `outbyte()` function in the assembly program *outchr.s*. Inside the `outbyte()` function, `PUTSTRING` is called using `TRAP #0` instruction to print the character to the terminal.

`outbyte()` function is created in *outchr.s* file. Its explanation is written in comments of Listing 2.

Listing 2: *outchr.s* file

```
.global outbyte                 | Export inbyte function to other files

.text
.even

outbyte:
    ** First argument (long word size) from C program is offset 4 from the top of the stack **
    ** to get the byte size argument, we need to offset 3 more bytes from the beginning of the
         first argument **
    ** so 7 is added to the stack pointer to get the byte sized argument sent from the C
        program **
    move.b  7(%sp), BUF_OUTBYTE | Get outbyte argument from stack
    movem.l %d0-%d3, -(%sp)     | Save registers that will be used
outbyte_start:
    move.l  #2, %d0             | System call number for PUTSTRING
    move.l  #0, %d1             | Serial port number = 0
    move.l  #BUF_OUTBYTE, %d2   | Buffer address
    move.l  #1, %d3             | Buffer size = 1 byte
    trap    #0                  | Initiate system call to PUTSTRING

    cmpi.b  #1, %d0             | Check if PUTSTRING was successful (1 byte read)
    bne     outbyte_start       | If not, try again

    movem.l (%sp)+, %d0-%d3     | Restore registers
    rts                         | Return from function

.section .bss
BUF_OUTBYTE:    .ds.b 1         | 1 byte Buffer for outbyte function
    .even
```

1.3.3 test program

In order to test the `printf()` and `scanf()` functions, a simple test program is created. The program simply echo the input from the user to the terminal. This program will terminate when character `q` is presented anywhere in the input.

The test program is written in *test1.c* file. Its explanation is written in comments of Listing 3.

Listing 3: *test1.c* file

```
#include <stdio.h>

int main() {
    char buf;
    printf("Hello World!\n");   // Check if the program is running
    while(1) {
        scanf("%c", &buf);      // Read a character from the keyboard
```

```
8         if (buf != 'q') {
9             printf("%c", buf);  // Print the character
10        } else {
11            break;              // Exit the loop if the character is 'q'
12        }
13    }
14    return 0;
15 }
```

1.4 Important points in programming

1. When connecting a function call from C language program to assembly program, we have to consider how the argumens are pushed to the stack. Arguments are pushed to the stack from the right to the left. Then value of the program counter is pushed to the stack.
   Therefore, to access the first argument, we have to add 4 to the stack pointer. To access the second argument, we have to add 8 to the stack pointer. And so on.
2. When accessing the arguments of the size other than long word, more offset must be added to the offset mentioned above.
   In case of accessing a word size argument, 2 must be added to the offset. In case of accessing a byte size argument, 3 must be added to the offset.
3. To ensure the completion of the system call, we have to check the value of the `D0` register. And the system call must be retried if the value of the `D0` register is not one.
4. For a function with return value, the return value must be stored in the `D0` register before returning from the function.

1.5 Encountered problems and solutions

1. The test of the theme 1 resulted in random characters being printed to the terminal at first. With the help from the given resources, I found out that the problem was caused by the wrong offset when accessing the arguments. This problem was fixed by adding the correct offset (7) to the stack pointer. The reasoning behind this offset is explained in section 1.4.
2. At first, the value when processing `inbyte()` function was not stored in the `D0` register before returning. This caused the value to be lost when the function returned. This problem was fixed by storing the value in the `D0` register before returning from the function.

1.6 Discussion

From the test program, we can see that the `printf()` and `scanf()` functions are working properly, similar to the standard C library functions. This means that the functionalities of the `printf()` and `scanf()` functions are successfully ported to the CPU board.

We have learned how to connect a function call from C language program to assembly program, and how to return the value from the assembly program to the C language program. This knowledge will be useful later, because we will have to utilize the understanding of the connection between C and M68kassembly language in order to add more functionalities to our kernel in next experiment themes.

Theme 2 preparation: Thought experiment

In this thought experiment, I have created 3 tasks that utilize 2 semaphores. The tasks can be written in pseudocode of C language as follows:

Listing 4: pseudocode of 3 user tasks

```
void task1(void){
    P(0);
    // perform some long task
}

void task2(void){
    P(1);
    P(0);
    // perform some task
}

void task3(void){
    V(0);
    // perform some task
}
```

The tasks listed in listing 4 are only created for demonstration purpose. Noticing that the number of P and V instruction are not equal, this will cause a problem when the tasks are executed and a semaphore is in a forever locked state. Also, using another task to release a semaphore (task 3) is not a good practice. This is because the resources that are being protected by the semaphore should be released by the task that is using it.

With the above in mind, I have created Table 1 below to show the state of the system when the tasks are executed.

Time	current task	ready	semaphore[0]		semaphore[1]	
			queue	value	queue	value
a. Initial setting	-	1,2,3,0	0	1	0	1
b. task 1 issue a P instruction on semaphore[0]	***1***	***2,3,0***	***1***	***0***	0	1
c. Task switching to task 2	***2***	***3,0***	1	0	0	1
d. Task 2 issue a P instruction on semaphore[1]	2	3,0	1	0	***2***	***0***
e. Task 2 issue a P instruction on semaphore[0]	2	3,0	***1,2***	***-1***	2	0
f. After the P instruction in task 2	***3***	***0***	1,2	-1	2	0
g. Task 3 issue a V instruction on semaphore[0]	3	***1,0***	***2***	***0***	2	0
h. Continue executing task 1	***1***	***0***	2	0	2	0

Table 1: The state of current task, ready queue, and semaphores after each step in time

2.1 Explanation of the table

- The initial setting of the system is shown in row a. where task 1, 2, and 3 are in the ready queue and both semaphores are in the unlocked state.
- At time b., task 1 is moved from ready queue to current task and it issues a P instruction on semaphore[0]. This causes the semaphore[0] to be in the locked state.
- At time c., task 1 is moved from current task to ready queue (due to clock interrupt putting unfinished task to the back of ready queue) and task 2 is moved from ready queue to current task.
- At time d., task 2 issues a P instruction on semaphore[1]. This causes the semaphore[1] to be in the locked state. Similar to task 1 locking semaphore[0] earlier at time b.
- At time e., task 2 issues another P instruction on semaphore[0]. However, task 2 is not able to lock semaphore[0] because it is already locked by task 1. This causes task 2 to be put into the waiting queue of semaphore[0].
- At time f., task 2 is put into a sleep state after unable to lock semaphore[0]. Task 3 is moved from ready queue to current task.
- At time g., task 3 issues a V instruction on semaphore[0]. This causes the top task in the waiting queue of semaphore[0] to be moved from the waiting queue to the ready queue. In this case, task 1 is moved from the waiting queue to the ready queue.

2.2 Discussion

Althought I have not made a group of user tasks that can be used in the real program in this thought experiment, I think this shows that the semaphore can be used to protect the resources that are being used by the user tasks. This is because the semaphore[0] is used to protect the resource that is being used by both task 1 and task 2. Therefore, I can see why the semaphore is required in the real program. Especially in the multitaslking environment.

To make a concept of semaphore clearer, imagine a playground with a slide, a swing, and a seesaw. There are several kids (tasks) at the playground who want to play on these different equipment, but they need to take turns and can only use one piece of equipment at a time. To help the kids take turns and avoid conflicts, the playground supervisor assigns each piece of equipment a "semaphore" (a flag or sign). When a kid wants to use the slide (similar to P instruction), for example, they have to make sure that the slide is empty (semaphore value is 1) before they can use it. If the slide is occupied (semaphore value is 0 or less), the kid has to wait until the other kid has given up the slide (similar to V instruction). In this way, the semaphores help the kids coordinate their access to the shared resources (the slide, swing, and seesaw) and make sure that everyone gets a turn to play.