SW 実験 II テーマ 1 レポート

C 過程 S-15 組 1TE20137W 2022/12/19

柳 鷹

目次

テーマ 1:C 言語ライブラリ関数の移植

テーマ 1 について

本実験のテーマ 1 では、68000 互換 CPU 搭載のターゲットボード上で C 言語処理系（標準入出力ライブラリ）をクロスコンパイラとしてカスタマイズするために入出力関数が動作することが目的である。具体的には以下のような流れでシリアルポートを経由してディスプレイに文字列が表示させることである。

<table>
<tr><td>出力</td><td>入力</td></tr>
<tr><td colspan="2">main()
test1.c</td></tr>
<tr><td>printf()</td><td>scanf()</td></tr>
<tr><td colspan="2">libc.a</td></tr>
<tr><td>write()</td><td>read()</td></tr>
<tr><td colspan="2">csys68k.c</td></tr>
<tr><td>outbyte:
outchr.s</td><td>inbyte:
inchrw.s</td></tr>
<tr><td colspan="2">trap#0</td></tr>
<tr><td>PUTSTRING</td><td>GETSTRING</td></tr>
<tr><td colspan="2">mon.s</td></tr>
<tr><td colspan="2">シリアルポート
Hardware</td></tr>
</table>

m68k-elf-gcc の libc.a に収められているライブラリ関数 printf(), scanf()を test1.c の main()内で呼び出す。これらの入出力関数は csys68k.c 内の write(), read()関数を用いて実装されている。このとき、シリアルポートに対して、write(), read()関数を使うためにソフトウェア実験 I で作成したエコーバックプログラムののシステムコール GETSTRING, PUTSTRING を呼び出すようにする。

テーマ 1 の実験全体に占める位置づけ

本実験のテーマ 1 では、後のテーマ 2,3 に向けて以下を確認することができる

- プログラムの分割コンパイル・アセンブル
- アセンブリ言語プログラムと C 言語プログラムとのリンク
- C 言語プログラムの起動

プログラムの説明

テーマ 1 では上述のように pritnf(), scanf()を用いるために、まずソフトウェア実験 I で作成したエコーバックプログラムを mon.s としてハードウェアを初期化、システムコールのベクタの登録をする。その後、outchr.s, inchrw.s にアセンブラ関数（サブルーチン）として%D0 レジスタにシステムコール番号 4 を入れ 1 文字表示する outbyte:と 、%D0 レジスタにシステムコール番号 3 を入れ 1 文字入力を受け付ける binbyte:を実装した。

プログラムのリスト

テーマ 1 で実装・変更したプログラムを以下に示す。

mon.s （初期化部分だけ）

```
************************************************************
** 初期化
************************************************************
.section .text
.even
.extern start           | crt0.s 内の start を extern
.global monitor_begin   | 大域変数(関数)の宣言
monitor_begin:
    * スーパーバイザ & 各種設定を行っている最中の割込禁止
    move.w #0x2000,%SR
    lea.l SYS_STK_TOP, %SP | Set SSP
    ****************
    ** 割り込みコントローラの初期化
    ****************
    move.b #0x40, IVR       | ユーザ割り込みベクタ番号を
                            | 0x40+level に設定.
    move.l #0x00ff3ff9,IMR | 全割り込みマスク MUART=>0,MTMR1=>1
    ****************
    ** 送受信 (UART1) 関係の初期化 ( 割り込みレベルは 4 に固定されている )
    ****************
    move.l #UART1_interrupt, 0x110  | 受信割り込みベクタをセット
    move.w #0x0000, USTCNT1 | リセット
    move.w #0xe108, USTCNT1 | 送受信可能 , パリティなし , 1 stop, 8 bit,
          | 受信割り込み許可, 送信割り込み禁止
    move.w #0x0038, UBAUD1  | baud rate = 230400 bps
    ****************
    ** タイマ関係の初期化 ( 割り込みレベルは 6 に固定されている )
    ****************
    move.w #0x0004, TCTL1   | restart, 割り込み不可 ,
          | システムクロックの 1/16 を単位として計時,
```

```
        | タイマ使用停止
    move.l #COMPARE_INTERPUT, 0x118 /* level 6 */
    move.l #SYS_CALL, 0x080 /*SYS_CALL の割り込みベクタ設定*/
    ****************************
    ** キューの初期化
    ****************************
    lea.l  top0, %a2
    lea.l  top1, %a3
    move.l %a2, out0
    move.l %a3, out1
    move.l %a2, in0
    move.l %a3, in1
    move.l #0, s0
    move.l #0, s1

    jmp start
```

inchrw.s

```
.global inbyte

.text
.even

inbyte:
    movem.l %D1-%D3, -(%sp)
inbyte_loop:
    move.l #1, %D0          |GETSTRING を呼び出す
    move.l #0,   %D1        | ch   = 0
    move.l #In_BUF, %D2     | p    = #BUF
    move.l #1, %D3          | size = 1
    trap   #0
    cmpi   #1, %d0          | GETSTRING の返り値が 1 と一致するか確認
    bne    inbyte_loop      | 一致しなければ再受信
    move.b In_BUF, %d0      |戻り値を d0 に追加
    movem.l (%sp)+, %D1-%D3
    rts

.section .bss
In_BUF:
    .ds.b 1
```

outchr.s

```
.global outbyte

.section .text
.even
*******************************
** outbyte
** 引数:
**       char 型のデータ(スタックに格納済)
** 出力:
**       シリアルポート 0 に出力
*******************************
outbyte:
    movem.l %d0-%d3, -(%sp)
out_byte_loop:
    move.l #SYSCALL_NUM_PUTSTRING, %d0 | PUTSTRING を指定
    move.l #0, %d1        | ch   = 0
    move.l %sp, %d2       | data = char[1]
    add.l  #23, %d2
    move.l #1, %d3        | size = 1
    trap   #0
    cmpi.l #1, %d0        | PUTSTRING の返り値が送信数と一致するか確認
    bne    out_byte_loop | 一致しなければ再送信
end_outbyte:
    movem.l (%sp)+, %d0-%d3
    rts

.equ SYSCALL_NUM_PUTSTRING,    2
```

test1.c

```
#include <stdio.h>

int main(void){
    while(1){
        char c[200];
        printf("input: ");
        scanf("%s", c);
        printf("output: %s\n", c);
    }

    return 0;
}
```

プログラムにおいて、注意したこと

mon.s の初期化の実装、GETSTRING で 1 文字を呼び出す inbyte:の実装の理解に問題はなかった。

本テーマのハイライトは outbyte: で PUTSTRING を呼び出す際の引数の取り方であったと考えている。PUTSTRING を trap 命令で呼び出すときの引数(括弧内は固有値)は 4 つで、システムコール番号(2)、チャンネル番号(0)、データの読み込み先の先頭アドレス(変数)、送信するデータ数(1)をそれぞれ%D0~%D3 レジスタに入れる。ここで気を付けるのは%D2 レジスタに入れるデータの読み込み先の先頭アドレスである。この outbyte:を呼び出すとき、スタックポインタには戻り先の PC（プログラムカウンタ）と%D0~%D3 レジスタの値がスタックポインタに積まれているので、データの読み込み先の先頭アドレスを取り出すには下に積んであるロングサイズのレジスタ 4 個の PC 分を考慮して、(スタックポインタの示す値+23)のアドレスを参照した。

プログラムに発生した問題とその原因・解決

本プログラムを作成中、outbyte のスタックポインタに積まれた退避したレジスタの値と PC に気を付けていたが、どうしてもターミナル上で表示できない問題が発生した。そこで mon.s（ソフトウェア実験 I で作成したエコーバックプログラム）の PUTSTRING のレジスタ退避部に問題があることが分かり、修正すると無事に動いた。（確認はしていないが、mon.s の差し替えでも解決したかもしれない。）

考察

本テーマでは、今までアセンブリ言語で開発していた 68000 互換 CPU 搭載のターゲットボード上で C 言語処理系も動作確認することができた。このアセンブリ言語プログラムと C 言語プログラムとのリンクができたことで、レジスタやメモリを意識したプログラムが書きやすいアセンブリ言語と、for や while などの条件分岐・繰り返しや入出力関数を書きやすい C 言語それぞれのメリットを生かした分割コンパイル・アセンブルの開発ができるようになったと感じている。しかしながら、outbyte:での引数の渡し方のように別々のプログラミング言語で開発を並行して進めるうえでのデメリットも発生するため、注意する必要があると感じた。

【準備】テーマ 2: マルチタスクカーネルの制作

ready キューを有する時分割マルチタスク環境下の思考実験

プログラムの説明

本項ではテーマ 2 で作成するマルチタスクカーネルの動作の動作確認をすることを前提に、ready キューを有する時分割マルチタスク環境において以下の 2 条件を満たすユーザタスク群を作成し、思考実験を行う。

- セマフォを 2 つ以上使う
- セマフォ、ready キューそれぞれにおいて、2 つ以上のタスクが入るタイミングが初期状態以外に存在する（セマフォ、ready キューで同時にこの条件を満たす必要はない）

プログラムのリスト

思考実験用の C 言語プログラムを以下に示す。

thought_test2.c

```c
#include <stdio.h>
#define L = 10000
#define M = 20000
#define N = 40000

void task1(){
    while(1){
        // task1-a
        for (j=0;j<L;j++){
            /* L: 1 秒間ループ、この待機ループの間にタイマ割込みは生じない */
        }
        P(0);
        // task1-b
        for (j=0;j<L;j++){
            /* L: 1 秒間ループ、この待機ループの間にタイマ割込みは生じない */
        }
        V(0);
        P(1):
        for (i=0;i<M;i++){
            /* M: 2 秒間ループ、この待機ループの間にタイマ割込みが必ず生じる */
        }
        V(1);
    }
}
```

```
void task2(){
    while(1){
        P(0);
        // task2-a
        for (i=0;i<M;i++){
            /* M: 2 秒間ループ、この待機ループの間にタイマ割込みが必ず生じる */
        }
        V(0);
        P(1);
        // task2-b
        for (i=0;i<N;i++){
            /* N: 3 秒間ループ、この待機ループの間にタイマ割込みが必ず生じる */
        }
        V(1);
        for (i=0;i<M;i++){
            /* M: 2 秒間ループ、この待機ループの間にタイマ割込みが必ず生じる */
        }
    }
}

void task3(){
    while(1){
        // task3-a
        P(0);
        for (i=0;i<M;i++){
            /* M: タスクループ1回につき、この待機ループで 1 回タイマ割込みが必ず生
じる */
        }
        V(0);
        // task3-b
        for (j=0;j<L;j++){
            /* L: 1 秒間ループ、この待機ループの間にタイマ割込みは生じない */
        }
        P(1);
        // taask3-c
        for (j=0;j<L;j++){
            /* L: 1 秒間ループ、この待機ループの間にタイマ割込みは生じない */
        }
        V(1);
    }
}
```

```
int main(void){
    init_kernel();
    set_task(task1);
    set_task(task2);
    set_task(task3);
    begin_sch();
    return 0;
}
```

状態変化の一覧

上記の思考実験プログラムの状態変化・及び動作説明を以下の一覧にまとめた。

タイマ単位時間	実行タスク	ready	semaphore0		semaphore1		状態
			coun	queue	coun	queue	
-		1,2,3	1	-	1	-	set_task
0	1(task1-a)	2,3	1	-	1	-	task1(資源利用無し)
1	2(task2-a)	3,1	0	-	1	-	task2が資源0確保
2	3(task3-a)	1,2	-1	3(task3-a)	1	-	task3が資源0のキューへ
3	1(task1- b)	2	-2	3(task3-a), 1(task1-b)	1	-	task1が資源0のキューへ
4	2(task2-a)	3	-1	1(task1-b)	1	-	task2が資源0解放
5	3(task3-a)	2,1	-1	1(task1-b)	1	-	task3(資源0利用)
6	2(task2-b)	3	-1	1(task1-b)	0	-	task2が資源1確保
7	3(task3-a)	2,1	0	-	0	-	task3が資源0解放
8	2(task2-b)	1,3	0	-	0	-	task2(資源1利用)
9	1(task1-b)	3,2	1	-	0	-	task1が資源0解放
10	3(task3-b)	2,1	1	-	0	-	task3(資源利用無し)
11	2(task2-b)	1,3	1	-	0	-	task2(資源1利用)
12	1(task1-b)	3,2	1	-	-1	1(task1-b)	task1が資源1のキューへ
13	3(task3-c)	2	1	-	-2	1(task1-b),3(task3-c)	task3が資源1のキューへ
14	2(task2-b)	1	1	-	-1	3(task3-c)	task2が資源1解放
15	1(task1-b)	2	1	-	-1	3(task3-c)	task1(資源1利用)
16	2(task2-c)	1	1	-	-1	3(task3-c)	task2(資源利用無し)
17	1(task1-b)	2,3	1	-	0	-	task1が資源1解放
18	2(task2-c)	3,1	1	-	0	-	task2(資源利用無し)
19	3(task3-c)	1,2	1	-	1	-	task3が資源1解放