

SW 実験 II テーマ 2,3

レポート

C 過程 S-15 組 1TE20137W 2023/01/18

目次

テーマ 2:マルチタスクカーネルの制作	1
テーマ 2 について.....	1
テーマ 2 の実験全体に占める位置づけ	1
プログラムのリスト及びその説明	2
プログラムにおいて、注意したこと	17
プログラムに発生した問題とその原因、解決方法.....	17
テーマ 3:応用	18
テーマ 3 について.....	18
テーマ 3 の実験全体に占める位置づけ	18
プログラムのリスト及びその説明	18
プログラムにおいて、注意したこと	30
プログラムに発生した問題とその原因、解決方法.....	30
個人課題について.....	30
考察	31

テーマ 2:マルチタスクカーネルの制作

テーマ 2 について

本実験のテーマ 2 では、テーマ 1 で移植した C 言語標準入出力ライブラリを用いてマルチタスクカーネル処理、つまり複数の「仕事」を 1 つのシステム下で切り替えながら実行する処理を記述することが目的である。タスクの切り替えは以下の二つがある。

- タスク間で資源を共有する場合の排他制御 (P・V システムコール)
- タイマ割り込みによるタスクの切り替え

これらを実装するために 2 つのキュー — ready キュー、各共有資源のキュー — を用意する。前者は順番が来たらいつでも実行可能なタスクの待ち行列で、後者は各共有資源の (後述する) セマフォが持つキューで他のタスクが資源を占有しているために休眠、つまり実行を待機している、タスクの待ち行列である。

テーマ 2 の実験全体に占める位置づけ

ここでは、マルチタスク処理を実行する上で必要な以下の 4 つの事項に注意する。

1. 特権命令の概念
2. 排他制御 (P・V システムコール) の概念
3. 1 つのシステムで複数タスク切り替えながら実行するために必要な処理
4. タスク切り替えのタイミングによるカーネル再入防止

1 つ目の特権命令は、OS には実行可能で利用者プログラムでは実行不可となるような処理を実装する際に必要なものである。OS と利用者プログラムで実行可能性を区別するために 2 つのモードを用いる。それはすべての命令が実行可能であるスーパーバイザモードと一部の命令は実行不可能となるユーザーモードである。スーパーバイザモードで実行するものは、入出力とタスク切り替えを担うタイマのセット・リセットであり、具体的にはエコーバックプログラムで作成した TRAP 命令を用いる。

2 つ目の排他制御は、複数のタスクが共有資源を同時に操作しないように制御するものである。各共有資源にはセマフォと呼ばれる、カウンタとキューを持つ構造体を持たせる。カウンタは初期化時には 1 にセットしてある。そこであるタスクがその資源を使用する際に、カウンタの値を一つ減らす。これを P 命令とする。そしてタスクが資源利用を終えるとカウンタを一つ増やす。これを V 命令とする。このときカウンタに着目すると、1 のときは資源利用可能、0 のときは他のタスクが資源利用中、0 未満のときは他のタスクが利用中かつさらに他のタスクが資源利用を待っている状態を示している。そして 0,0 未満のときに資源利用を待っているタスクをキューに保存して、順に取り出すことで排他制御を可能にしている。

3 つ目の 1 つのシステムで複数タスク切り替えながら実行するために必要な処理は、タスクの実行が中断され ready キューに計算機内部の状態が正しく保存され、逆にタスクが中断されていたタスクを回復する際に元の計算機内部の状態を復元することである。具体的には、実行中断時の PC（プログラムカウンタ）の値、レジスタの値、スタックの内容を保存・復元が必要である。

4 つ目のタスク切り替えのタイミングによるカーネル再入防止は、本実験においては P・V 命令とタイマ割り込みの整合性をとるために、P・V 命令の発効前に走行レベルを 7 にして割り込みを禁止する。

プログラムのリスト及びその説明

本テーマではグループでの共同作業であったため、担当部分については詳細に、担当外の部分については概要を記す。なお当グループの担当箇所の分担は、マルチタスク制御、タイマ制御、セマフォ制御、タスク切り替え制御部の 5 つで、当人はタイマ制御部（init_timer:, set_timer:, reset_timer:, hard_clock）を担当した。

テーマ 2 で実装・変更したプログラムを以下に示す。

mtk_c.h

```
/*
 * mtk_c.h
 *
 * 各種定数を定義する
 */
#ifndef mtk_c_H /* if not defined this */
#define mtk_c_H /* define it */

/* 定数の定義 */
#define NULLTASKID 0 /* キューの終端 */
#define NUMTASK 7 /* 最大タスク数 */
#define STKSIZE 1024 /* スタックサイズ = 1KB */
#define NUMSEMAPHORE 3

/* int の型エイリアス */
typedef int TASK_ID_TYPE;

/* 各構造体の定義 */
typedef struct {
    int count;
    TASK_ID_TYPE task_list;
} SEMAPHORE_TYPE;

typedef struct{
    void (*task_addr)();
    void *stack_ptr;
    int priority;
    int status;
    TASK_ID_TYPE next;
} TCB_TYPE;

typedef struct{
    char ystack[STKSIZE];
    char sstack[STKSIZE];
} STACK_TYPE;

/* 大域変数の宣言 */
TASK_ID_TYPE curr_task;
TASK_ID_TYPE new_task;
TASK_ID_TYPE next_task;

TASK_ID_TYPE ready;
```

```

SEMAPHORE_TYPE semaphore[NUMSEMAPHORE];
TCB_TYPE task_tab[NUMTASK + 1];

STACK_TYPE stacks[NUMTASK];
/* 関数のプロトタイプ宣言 */
void init_kernel(void);
void set_task(void *p);
void* init_stack(int id);
void begin_sch(void);
void sched(void);
TASK_ID_TYPE removeq(TASK_ID_TYPE *q);
void addq(TASK_ID_TYPE *q, TASK_ID_TYPE task_id);
//void addq1(TASK_ID_TYPE *q, TASK_ID_TYPE task_id);
void sleep(int s_id);
void wakeup(int s_id);
void p_body(int s_id);
void v_body(int s_id);

/* 関数の extern 宣言 */
extern void first_task(void);
extern void pv_handler(void);
extern void P(int semaphore_id);
extern void V(int semaphore_id);
extern void swtch(void);
extern void hard_clock(void);
extern void init_timer(void);

#endif /* mtk_c_H */

```

ヘッダファイル” mtk_c.h”では、” mtk_c.c”と”test2.c”の外部で定義される関数の extern 宣言とカーネル関連の大域変数を含んでいる。

大域変数

curr_task	現在実行中のタスク ID
new_task	現在登録中のタスク ID
next_task	次の実行するタスク ID
ready	実行待ちのタスクキュー
semaphore	共有資源のセマフォ
task_tab	タスクコントロールブロックの配列
stacks	タスクスタックの配列

mtk_c.c

```
/*
 *
 * sched() : タスクのスケジュール関数
 * removeq() : キューからタスクを取り出す
 * addq() : キューの最後尾にタスクを登録
 * sleep() : タスクを休眠状態にしてタスクスイッチをする
 * wakeup() : 休眠状態のタスクを実行可能状態にする
 *
 * In the future, these functions will be included into mtk_c.c
 */
#include "mtk_c.h"
#include <stdio.h>

void init_kernel() {
    ready = 0; //ready の初期化
    *(int*)0x084 = (int)pv_handler; //pv_handler を trap#1 の割り込みベクタ
    に登録
    for (int i=0; i<NUMSEMAPHORE; i++) {
        semaphore[i].count = 1;
        semaphore[i].task_list=0;
    }
    for (int i=0; i<NUMTASK; i++) {
        task_tab[i+1].task_addr=0;
        task_tab[i+1].stack_ptr=0;
        task_tab[i+1].priority=0;
        task_tab[i+1].status=0;
        task_tab[i+1].next=0;
    }
}

void set_task(void *p) {
    int i = 1;
    while (1) { //タスク ID の検索
        if (task_tab[i].status == 0) {
            break;
        }
        i = i+1;
    }
    new_task = i;
    task_tab[new_task].task_addr = p; //TCB の更
    新
    task_tab[new_task].status = 1;
}
```

```

    task_tab[new_task].stack_ptr = init_stack( new_task );           //ス
    タックの初期化
    addq(&ready, new_task);                                         //キューへの登録
}

void *init_stack(int id) {
    int *ssp1;                                                       //4 バイト用 int
    unsigned short int *ssp2;                                       //2 バイト用 unsigned short int
    void *back;                                                      //戻り値用
    ssp1 = (int *)&stacks[id-1].sstack[STKSIZE];                 //stacks[id-1]に各値
    を代入
    *--ssp1 = (int)task_tab[id].task_addr;
    ssp2 = (unsigned short int *)ssp1;                             //2 バイトに切り替え
    *--ssp2 = 0x0000;
    ssp1 = (int *)ssp2;
    for (int i=0; i<15; i++) {                                     //15*4 バイト戻す
        *--ssp1;
    }

    *--ssp1 = (int *)&stacks[id-1].ustack[STKSIZE];

    back = (void *)ssp1;
    return back;
}

void begin_sch()
{
    int i;
    i = removeq(&ready);
    curr_task = i;

    init_timer();
    first_task();
    return;
}

TASK_ID_TYPE removeq(TASK_ID_TYPE *q){
    // 先頭タスクの id を保存
    TASK_ID_TYPE head_task_id = *q;
    // 先頭タスクを除去して新たなタスク id を登録
    *q = task_tab[head_task_id].next;

    return head_task_id;
}

```

```

void sched(void) {
    printf("\n\n timer ----- \n\n");
    // ready キューの先頭タスクを取り出して next_task へ
    next_task = removeq(&ready);
    // next_task が NULL の場合はタイマ割り込みが来るまでループ
    while(!next_task);
}

void addq(TASK_ID_TYPE *q, TASK_ID_TYPE task_id){
    // id として先頭 id を保存
    TASK_ID_TYPE id = *q;
    *(char *)0x00d0002f = 'a';
    if (!id) { // 渡された queue の先頭が NULL だった場合
        *q = task_id;
        task_tab[task_id].next = NULLTASKID;
        return;
    }
    *(char *)0x00d0002d = 'a';
    // task_tab の next が NULL になるまでたどる
    while (task_tab[id].next) id = task_tab[id].next;
    *(char *)0x00d00039 = 'a';
    // 最後尾にタスク id を追加
    task_tab[id].next = task_id;
    task_tab[task_id].next = NULLTASKID;
}

void sleep(int s_id){
    TASK_ID_TYPE* head_q = &(semaphore[s_id].task_list);
    // 該当セマフォの待ち行列に現タスクをつなぐ
    addq(head_q, curr_task);
    // 次に実行するタスク id を next_task にセット
    sched();
    // タスクを切り替え
    swtch();
}

void wakeup(int s_id){
    // セマフォの待ち行列の先頭タスクを取得
    TASK_ID_TYPE* head_sem = &(semaphore[s_id].task_list);
    // セマフォの待ち行列の先頭タスクをキューから除去
    TASK_ID_TYPE head_task = removeq(head_sem);
    // ready キューにその先頭タスクを追加
    addq(&ready, head_task);
}

```



```

void p_body(int s_id){

    semaphore[s_id].count--; //セマフォの count 値を 1 減らす

    if (semaphore[s_id].count < 0){
        sleep(s_id);          //セマフォが獲得できない場合は休眠状態に入る
    }

}

void v_body(int s_id){

    semaphore[s_id].count++; //セマフォの count 値を1増やす

    if (semaphore[s_id].count <= 0){
        wakeup(s_id);          //セマフォが空けばそのセマフォを待っているタスクを1
        つ実行可能状態にする
    }

}

```

init_kernel

TCB 配列の初期化、ready キューの初期化、P・V 命令のベクタ登録、セマフォの初期化を担う。

set_task

ユーザータスク関数へのポインタを引数に取り、task_tab の中から空きスロットを見つけ、new_task に代入する。それをもとに TCB の更新、スタックの初期化、ready キューへの登録を行う。これは test ファイルの main の最初に呼び出される。

init_stack

ユーザータスク用のスタック ssp の初期化を担う

begin_sch

ready キューからタスクを一つ取り出し curr_task に代入し、タスク切り替えのタイマの開始、最初のタスクの起動を担う。

addq

引数にキューへのポインタとタスク ID を取り、TCB をキューの最後に登録する。

removeq

引数にキューへのポインタとタスク ID を取り、キューから先頭タスク ID を返す。

p_body

セマフォのカウントを 1 つ減らした後、カウントが 0 未満ならそのタスクをキューに入れ、休眠状態にさせる。

v_body

セマフォのカウントを 1 つ増やした後、カウントが 0 以下ならキューの先頭タスクを復帰させ、ready キューに入れる。

sched

ready キューの先頭タスクを取り出して、next_task に登録する。また next_task が null の場合はタイマ割り込みがあるまで無限ループを行う。

sleep

タスクを休眠状態にする。一セマフォの待ち行列に休眠すべきタスクをつなぎ、next_task のセット、タスク切り替えを行う。

wakeup

休眠状態のタスクを回復させる。一セマフォのキューの休眠状態だった先頭タスクをポップし、ready キューに追加する。

mtk_asm.s

```
.section .bss

.global first_task
.global pv_handler
.global P
.global V
.global swtch
.global hard_clock
.global init_timer

.extern addq
.extern sched
.extern curr_task
.extern next_task
.extern task_tab
.extern ready

.equ    SIZE_OF_TCB, 0x14
.equ    IOBASE,      0x00d00000
.equ    LED7,        IOBASE+0x000002f | ボード搭載の LED 用レジスタ
.equ    LED6,        IOBASE+0x000002d | 使用法については付録 A.4.3.1
.equ    LED5,        IOBASE+0x000002b
.equ    LED4,        IOBASE+0x0000029
.equ    LED3,        IOBASE+0x000003f
.equ    LED2,        IOBASE+0x000003d
.equ    LED1,        IOBASE+0x000003b
.equ    LED0,        IOBASE+0x0000039
.section .text
.even
swtch:
    * %SR, 全レジスタの値をスタックに積む
    move.w %SR, -(%SP)
    movem.l %D0-%D7/%A0-%A6, -(%SP)
    move.l %USP, %a0
    move.l %a0, -(%SP)
    jsr    get_tcb_ssp                | %a0 <- TCB の SSP の位置
    move.l %SP, (%a0)                | SSP を TCB の所定の位置に格納
    move.l next_task, curr_task
    jsr    get_tcb_ssp                | %a0 <- TCB の SSP の位置
    move.l (%a0), %SP                | TCB に記録されている SSP を回復
```

```

bra      end_swch

get_tcb_esp:
    move.l  curr_task, %d0          | %d0 = 現タスクの ID 例
d0 = 2
    move.l  #SIZE_OF_TCB, %d1      | %d1 = TCB のサイズ
void が4バイトで要素が5個だから 4*5
    mulu.w  %d0, %d1               | 該当タスクの TCB にアクセ
スするためのオフセット
    lea.l   task_tab, %a0          | task_tab の先頭アドレス
    adda.l  %d1, %a0               | アドレスとオフセットを加算
= 現
    adda.l  #4, %a0                |
task_tab[ID].stack_ptr の位置を取得
    rts
end_swch:
    move.l  (%SP)+, %A0
    move.l  %A0, %USP
    movem.l (%SP)+, %D0-%D7/%A0-%A6 | 各レジスタを復帰
    rte

pv_handler:

    movem.l %D0-%D7/%A0-%A6, -(%sp) | 使用可能性のあるレジスタの退避
    move.w  %SR, -(%sp)             | SR の退避
    move.w  #0x2700, %SR            | 走行レベルを7へ
    movem.l %D1, -(%sp)            | D1 を引数としてスタックに積む

    cmp.l   #0, %D0                | P からの呼び出しの場合
    beq     p_order
    cmp.l   #1, %D0                | Q からの呼び出しの場合
    beq     v_order

p_order:

    jsr p_body                    | p 命令実行
    bra end_pv_handler

v_order:

    jsr v_body                    | v 命令実行
    bra end_pv_handler

end_pv_handler:
    add.l   #4, %sp                | 引数(d1)分 sp を上げる
    move.w  (%sp)+, %SR            | SR を復帰
    movem.l (%sp)+, %D0-%D7/%A0-%A6 | レジスタの復帰
    rte

```

```

P:
    movem.l %D0-%D1, -(%sp)
    move.l #0, %D0          | P システムコールの ID=0 を D0 レジスタに
    move.l 12(%sp), %D1     | スタックに入っている引数を D1 に (d0-d1+ 戻り番地の
PC= 12)
    trap #1
    movem.l (%sp)+, %D0-%D1
    rts

V:
    movem.l %D0-%D1, -(%sp)
    move.l #1, %D0          | V システムコールの ID=1 を D0 レジスタに
    move.l 12(%sp), %D1     | スタックに入っている引数を D1 に (d0-d1+ 戻り番地の
PC= 12)
    trap #1
    movem.l (%sp)+, %D0-%D1
    rts

*****
** hard_clock
** SW 実験 1 で作成したタイマ用ハードウェア割り込み処理インターフェースから呼び出される
**
** このルーチン内で使用するレジスタをスーパーバイザスタックに積むが、
** タイマ割り込みで実行されるルーチンであるため退避忘れが多いらしい
*****

hard_clock:
    movem.l %D0-%D7/%A0-%A6, -(%sp)          | 使用可能性のあるレジスタ
の退避
    move.w %SR, %D0                          | %SR を %D0 へ
    btst.l #13, %D0                          | SR の 13bit 目を見てスー
パーバイザか否かを確認
    beq     end_hard_clock                   | 13bit 目が 1 でなければ、
スーパーバイザモードでなければ終了
    move.l curr_task, -(%sp)
    move.l #ready, -(%sp)
    move.b #'a', LED1
    jsr     addq                              | current_task を ready
の末尾に追加
    add.l   #8, %sp
    move.b #'a', LED2
    jsr     sched                            | sched を起動し、次に実行
されるタスク ID が next_task にセット
    move.b #'a', LED3

```

```

        jsr      swtch                                | swtch の起動
        move.b   #'a', LED4

end_hard_clock:
        movem.l  (%sp)+, %D0-%D7/%A0-%A6             | レジスタ復帰退避
        rts

init_timer:
        movem.l  %D0-%D2, -(%sp)                     | 使用可能性のあるレジスタの退避
        move.l   #3, %D0                             | システムコール RESET_TIMER の番号
        trap     #0                                   | RESET_TIMER 呼び出し
        move.l   #4, %D0                             | システムコール SET_TIMER の番号
        move.w   #10000, %D1                          | 割り込み発生周期は 1 秒に設定
(p38)
        move.l   #hard_clock, %d2                    | 割り込み時に起動するルーチンの先頭
アドレス
        trap     #0                                   | SET_TIMER 呼び出し
        movem.l  (%sp)+, %D0-%D2                     | SR の退避
        rts

*****
**first_task
*****

first_task:
        move.l   curr_task, %d0                      /* TCB 先頭番地の計算 */
        lea.l    task_tab, %a1                      /* 見つけたアドレスを%a1に */
loop_first_task:
        subq.l   #1, %d0                             /* 配列の何番目なのか計算 */
        add.l    #0x14, %a1                          /* %a1:先頭アドレス */
        cmp      #0, %d0
        bne      loop_first_task
        /*USP,SSP の値の回復*/
        add.l    #4, %a1                             /* a1 に TCB の SSP */
        move.l   (%a1), %ssp

        movem.l  (%sp)+, %a0
        move.l   %a0, %usp
        movem.l  (%sp)+, %D0-%D7/%A0-%A6
        rte                                           /*rte*/

```

first_task

最初のタスクの起動を担う。

`pv_handler`

走行レベルを 7 にして他の割り込み（特に、タイマ割り込み）を禁止して P・V 命令を発行する。

P

P システムコールの入口で、`pv_handler` から呼ばれて TRAP #1 命令を実行する。

V

V システムコールの入口で、`pv_handler` から呼ばれて TRAP #1 命令を実行する。

`swtch`

タスク切り替えを起こす関数で、PC（プログラムカウンタ）の値、レジスタの値、スタックの内容を保存・復元を行う。

`hard_clock`

スーパーバイザモードでタイマ割り込みを行うサブルーチンである。まず、`addq` を呼び出して実行中のタスクを ready キューの末尾に追加する。次に `shed` を呼び出し ready キューの先頭にあるタスクを `next_task` にセットして `swtch` を起動することでタイマ割り込みを可能にしている。

`init_timer`

上述の `hard_clock` をベクタテーブルに登録するサブルーチンである。`mon.s`（エコーバックプログラム）の Trap #0 命令でシステムコール 3,4 番の `RESET_TIMER` と `SET_TIMER` を呼び出している。なお、割り込み周期は目視しやすいように 1 秒（=10000[0.1ms]）としている。

test2.c

```
#include <stdio.h>
#include "mtk_c.h"

extern void init_kernel();
extern void set_task(void *a);
extern void begin_sch();
#define L 500
#define M 1000
#define N 2000
void task1(){
while(1){
    // task1-a
    for (int j = 0;j<L;j++){
        printf("task1-a %d \n", 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 (int j = 0;j<L;j++){
        printf("task1-b %d \n", 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);
    // task2-c
    for (int i = 0;i<M;i++){
        printf("task1-c %d \n", 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 (int i = 0;i<M;i++){
        printf("task2-a %d \n", 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 (int i = 0;i<N;i++){
```



```

    printf("task2-b %d \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);
// task2-c
for (int i = 0; i < M; i++){
    printf("task2-c %d \n", 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 (int i = 0; i < M; i++){
        printf("task3-a %d \n", i);
        /* M: タスクリープ 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 (int j = 0; j < L; j++){
        printf("task3-b %d \n", 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);
    // task3-c
    for (int j = 0; j < L; j++){
        printf("task3-c %d \n", 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;
}

```

test2.c ではテーマ 1 のレポートで作成した思考実験を改良してマルチタスク環境下でセマフォによる動作確認をしている。

プログラムにおいて、注意したこと

本テーマで注意したことは、コメントアウトを残すことである。共同開発で進めていくため、単体テスト（主に文法的なエラーをなくす程度）では通っても結合テストを行う場合、他の作業の責任者にプログラムの概略を記す必要がある。もちろん、各担当箇所以外の作業内容を理解しようと実験テキストに目を通すが、やはり実際にソースコードとして実装すると全体の流れは分かっているにもかかわらず一行一行で何をしているかについては責任者以外の理解は曖昧である。したがって、担当箇所についてはほとんど行にコメントアウトを残した。

プログラムに発生した問題とその原因、解決方法

本テーマの D1~D5 の 5 つのデバッグ作業で何度も問題に直面した。ただ、その直面の仕方は主に 2 パターンであった。それはコンパイル前の文法的なエラー（及び警告）とコンパイル後の例外エラーであった。特にバスエラーの等の例外エラーでは、printf デバッグを用いることが難しかった。そのため問題発生後、原因追求と解決のためには

- 文法的なエラーを減らしコンパイルを通す
- LED デバッグで問題発生箇所を発見する
- printf デバッグを行う

という流れで行った。

例えば、デバッグ D1 の作業では何度もバスエラーが起きて各担当者に思い当たる問題がなく、printf デバッグも動作しない状態で四苦八苦していた。そこで C 言語ファイル・アセンブラファイルに関わらず、まず文法的なエラーを担当者がなくした後、分担してソースコード一行一行に LED デバッグを挟み、原因追及を行った。結局、D1 では begin_sch()で呼び出した init_timer()から返ってこず、init_timer で”rts”ではなく”rte”になっていることが分かった。それは init_timer の最終行での LED デバッグには成功し、begin_sch()の init_timer の後、first_task()の前の LED デバッグができなかったからであった。

テーマ 3:応用

テーマ 3 について

テーマ 3 では、テーマ 2 までに作成した C 言語標準入出力ライブラリを用いたマルチタスク処理環境において、以下のようなプログラムを実行することが目的である。

- 3 つ以上のタスクを順に実行する
- マルチタスクとして 2 つのシリアルポートを用いて 2 つのディスプレイに出力するタスクを実行する
- 一方のポートの入力をもう一方のポートで出力するようなタスクを作成し、排他制御を行う

テーマ 3 の実験全体に占める位置づけ

プログラムのリスト及びその説明

テーマ 3 で実装・変更したプログラムを以下に示す。

csys68k.h

```
#include <stdarg.h>
#include <fcntl.h>
#include <errno.h>

extern void outbyte(int a,unsigned char c);
extern char inbyte(int b);

int read(int fd, char *buf, int nbytes){
    int ch;

    switch(fd){
    case 0:
    case 3:
        ch = 0;
        break;

    case 4:
        ch = 1;
        break;

    default:
        return EBADF;
        break;
    }
    char c;
    int i;

    for (i = 0; i < nbytes; i++) {
        c = inbyte(ch);

        if (c == '\r' || c == '\n'){ /* CR -> CRLF */
            outbyte(ch, '\r');
            outbyte(ch, '\n');
            *(buf + i) = '\n';

            /* } else if (c == '\x8'){ */ /* backspace \x8 */
            /* } else if (c == '\x7f'){ */ /* backspace \x8 -> \x7f (by terminal
config.) */
            if (i > 0){
                outbyte(ch, '\x8'); /* bs */
                outbyte(ch, ' '); /* spc */
                outbyte(ch, '\x8'); /* bs */
                i--;
            }
        }
    }
}
```

```

    }
    i--;
    continue;

} else {
    outbyte(ch, c);
    *(buf + i) = c;
}

if (*(buf + i) == '\n'){
    return (i + 1);
}
}
return (i);
}

int write (int fd, char *buf, int nbytes){
    int i, j, ch;
    switch(fd){
        case 1:
        case 2:
        case 3:
            ch = 0;
            break;

        case 4:
            ch = 1;
            break;

        default:
            return EBADF;
    }
    for (i = 0; i < nbytes; i++) {
        if (*(buf + i) == '\n') {
            outbyte (ch, '\r');          /* LF -> CRLF */
        }
        outbyte (ch, *(buf + i));
        for (j = 0; j < 300; j++);
    }
    return (nbytes);
}

int fcntl(int fd, int cmd, ...) {
    if (cmd == F_GETFL) {
        return O_RDWR;
    }

```

```
} else {  
    return 0;  
}  
  
}
```

read, write

2 ポート入出力を可能にするため、ストリーム入出力関数の `scanf()`, `printf()` を構成する低レベル関数 `read()`, `write()` に引数としてファイルディスクリプタを渡している。テーマ 2 までに実装した `scanf()`, `printf()` も引き続き利用できるように

- ファイルディスクリプタ 0=標準入力に対しては `scanf()`
- ファイルディスクリプタ 1,2=標準出力・標準エラー出力に対しては `printf()`

となるように変更した。また、

- UART1 (つまりポート 0) にはファイルディスクリプタ 3 を割り当て出力
- UART2 (つまりポート 1) にはファイルディスクリプタ 4 を割り当て出力

するように実装することで 2 ポートでの入出力を可能にした。

fcntl

前項の通り、ストリーム関数の入力元・出力先をファイルディスクリプタで判別させるには C ライブラリ関数 `fdopen(int fd, const char *mode)` で入出力ストリームをファイルディスクリプタへの割り当てが必要となる。例えば本実験の場合、ファイルディスクリプタ 3 に、読み込みを示す "r" と指定した場合、UART1 からの読み込みとなる。この `fdopen` が呼び出す低レベル関数が `fcntl` である。これはこのファイルディスクリプタと指定する入出力モードが二つのみを引数にとり、前者の入出力属性と後者が一致するかを確認する。一致した場合「入出力可能」を返し、それ以外は単に 0 を返すように実装した。

inchrw.c

```
.global inbyte

.text
.even

inbyte:
    movem.l %D1-%D3, -(%sp)
inbyte_loop:
    move.l #1, %D0          | GETSTRING を呼び出す
    move.l 16(%sp), %D1      | ch  = 0 or 1
    move.l #In_BUF, %D2     | p   = #BUF
    move.l #1, %D3          | size = 1
    trap #0
    cmpi   #1, %D0          | GETSTRING の戻り値が 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
```

inbyte

テーマ 2 までとの変更点は、ポート番号を引数に取ったことである。引数であるポート番号はスタックポインタから戻り先の PC（プログラムカウンタ）と%D1~%D3 レジスタの値を考慮して参照している。

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 20(%sp), %d1 | ch = 0 or 1
    move.l %sp, %d2 | data = char[1]
    add.l #27, %d2
    move.l #1, %d3 | size = 1
    trap #0
    cmpi.l #1, %d0 | PUTSTRING の戻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
    movem.l (%sp)+, %d0-%d3
    rts

.equ SYSCALL_NUM_PUTSTRING, 2
```

outbyte

テーマ 2 までとの変更点は、引数に char 型のデータに加えポート番号取ったことである。データ読み込み先の先頭アドレスとポート番号はスタックポインタが下から積まれるので、ポート番号、データの読み込み先の順番で、戻り先の PC（プログラムカウンタ）と %D0~%D3 レジスタの値を考慮してデータの読み込み先として参照している。

test3.c

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include "mtk_c.h"
FILE* com0in;
FILE* com0out;
FILE* com1in;
FILE* com1out;

void init_file(){
    com0in = fdopen(3, "r");
    com0out = fdopen(3, "w");
    com1in = fdopen(4, "r");
    com1out = fdopen(4, "w");
}

void show(int width, int height, char question[height][width], char
column[width], int w_answer, int h_answer, int a1){
    int w,h;
    // 描画部分
    // 列名および空白描画
    fprintf(com0out, "      |");
    fprintf(com1out, "      |");
    for (w = 1; w < width-1; w++){
        fprintf(com0out, "%c", column[2*w]);
        fprintf(com1out, "%c", column[2*w]);
        fprintf(com0out, "%c|", column[2*w+1]);
        fprintf(com1out, "%c|", column[2*w+1]);
    }
    fprintf(com0out, "\n      ");
    fprintf(com1out, "\n      ");
    for (w = 0; w < width-1; w++){
        fprintf(com0out, "  |");
        fprintf(com1out, "  |");
    }
    fprintf(com0out, "\n");
    fprintf(com1out, "\n");
    // 間違い本体
    for(h=0; h<height; h++){
        // 列名を描画するかどうか
        for (w = 0; w < width; w++){
            if(h == h_answer && w == w_answer && a1 == 0 ){
                fprintf(com0out, "\x1b[101m%2c\x1b[0m|", question[h][w]);
                fprintf(com1out, "\x1b[101m%2c\x1b[0m|", question[h][w]);
            }
        }
    }
}
```

```

        continue;
    }
    if(h == h_answer && w == w_answer && a1 == 1 ){
        fprintf(com0out, "\x1b[102m%2c\x1b[0m|", question[h][w]);
        fprintf(com1out, "\x1b[102m%2c\x1b[0m|", question[h][w]);
        continue;
    }
    else{
        fprintf(com0out, "%2c|", question[h][w]);
        fprintf(com1out, "%2c|", question[h][w]);
    }
}
fprintf(com0out, "\n");
fprintf(com1out, "\n");
}
}

char *check_answer;
int answerer = 2;

void task0(){
while(1){
    char s0;
    // 入力受付
    fscanf(com0in, "%s", &s0);

    // トーク画面制限
    P(0);
    // 自分の入力値削除
    fprintf(com0out, "\e[A\e[K");
    // 自分をフォントを赤色にして自分の右側に表示
    fprintf(com0out, "\x1b[31m\n%80s\n\x1b[0m", &s0);
    // 相手の入力値削除
    fprintf(com1out, "\e[A\e[K");
    // 自分をフォントを赤色にして相手の左側に表示
    fprintf(com1out, "\x1b[31m\n%s\n\x1b[0m\n", &s0);

    // ゲームスタートコマンドかどうか
    if(strncmp(&s0, "start",5) == 0){
        int width, height;
        int width_input, height_input;
        fprintf(com1out, "Player0 is now setting rule ...");
        fprintf(com0out, "\nThe size of widths(letters) more than 2,
less than 10 ");
        scanf("%d", &width_input);
        width = width_input;
    }
}
}

```

```

fprintf(com1out, "\nThe size of width = %d\n", width);
width += 2;

fprintf(com0out, "The size of heights(letters) less than 26 ");
scanf("%d", &height_input);
fprintf(com0out, "\n");
height = height_input;

fprintf(com1out, "\nThe height of width = %d\n", width);

int h,w;
char column[2*width-1];
char question[height][width];
char q = 'a';
char a = 'e';
int a1 = 2;

int w_answer, h_answer;
w_answer = rand() % (width - 2) + 2;
h_answer = rand() % height;

// 列(int)を作成
for(w=0; w < width-1; w++){
    if(w<10){
        column[2*w] = '0';
        column[2*w+1] = '0' + w;
    }
    else{
        column[2*w] = '0' + (w/10 % 10);
        column[2*w+1] = '0' + (w % 10);
    }
}

// 行(char)を作成(各列の先頭要素を操作)
for(h=0; h < height; h++){
    question[h][0] = 65 + h;
    // 各列の2列目は空白
    question[h][1] = ' ';
}

// 間違い本体部分
for(h=0; h<height; h++){
    for (w = 2; w < width; w++)
    {
        if(h == h_answer && w == w_answer){

```

```

        question[h][w] = a;
    }
    else{
        question[h][w] = q;
    }
}
}

show(width, height, question, column, w_answer, h_answer, a1);

char *check0 = &question[h_answer][0];
char *check1 = &column[2*(w_answer-1)];
char *check2 = &column[2*(w_answer-1)+1];
char *check;
sprintf(check, "%c%c%c\n", check0[0], check1[0], check2[0]);
check_answer = check;

fprintf(com0out, "\nGive cell name faster than your opponent!
ex)A01\n");
fprintf(com1out, "\nGive cell name faster than your opponent!
ex)A01\n");

// トーク画面解放
V(0);

while(1){
    // 入力受付
    fscanf(com0in, "%s", &s0);

    // トーク画面制限
    P(0);

    // 自分の入力値削除
    fprintf(com0out, "\e[A\e[K");
    // 自分をフォントを赤色にして自分の右側に表示
    fprintf(com0out, "\x1b[31m\n%80s\n\x1b[0m", &s0);
    // 相手の入力値削除
    fprintf(com1out, "\e[A\e[K");
    // 自分をフォントを赤色にして相手の左側に表示
    fprintf(com1out, "\x1b[31m\n%s\n\x1b[0m\n", &s0);
    if(answerer == 1){
        fprintf(com0out, "\n\n \x1b[32mPlayer1\x1b[0m solved
faster!\n\n");
        fprintf(com1out, "\n\n \x1b[32mPlayer1\x1b[0m solved
faster!\n\n");
    }
}

```

```

        show(width, height, question, column,
w_answer,h_answer,1);
        break;
    }
    if(strncmp(&s0, check_answer, 3) == 0){
        answerer = 0;
        fprintf(com0out, "\n\n \x1b[31mPlayer0\x1b[0m solved
faster!\n\n");
        fprintf(com1out, "\n\n \x1b[31mPlayer0\x1b[0m solved
faster!\n\n");
        show(width, height, question, column,
w_answer,h_answer,0);
        break;
    }

    if(strncmp(&s0, "restart",7)==0){
        break;
    }
}
// トーク画面解放
V(0);
}
else{
    // トーク画面解放
    V(0);
}
}
}

void task1(){
while(1){
    char s1;
    // 入力受付
    fscanf(com1in, "%s", &s1);

    // トーク画面制限
    P(0);
    // 自分の入力値削除
    fprintf(com1out, "\e[A\e[K");
    // 自分をフォントを緑色にして自分の右側に表示
    fprintf(com1out, "\x1b[32m\n%80s\n\x1b[0m", &s1);
    // 相手の入力値削除
    fprintf(com0out, "\e[A\e[K");
    // 自分をフォントを緑色にして相手の左側に表示
    fprintf(com0out, "\x1b[32m\n%s\n\x1b[0m\n", &s1);
}
}

```

```

        if(!check_answer){
            strncmp(&s1, check_answer, 3) == 0;
            answerer = 1;
        }

        // トーク画面解放
        V(0);
    }
}

int main(void){
    init_kernel();
    init_file();

    set_task(task0);
    set_task(task1);
    fprintf(com0out, "\nFind 'e'\n");
    fprintf(com1out, "\nFind 'e'\n");
    fprintf(com0out, "\nYou're Player0 You can set rule\n");
    fprintf(com0out, "Give 'start' to play!\n");
    fprintf(com1out, "\nYou're Player1 You can't set rule\n");
    fprintf(com0out, "\nYou can also enjoy chatting\n\n");
    fprintf(com1out, "\nYou can also enjoy chatting\n\n");

    begin_sch();
    return 0;
}

```

com0in, com0out, com1in, com1out

RS232 ポートに対するファイルハンドルを保持する FILE 構造体へのポインタの大域変数

個人課題

後述する

プログラムにおいて、注意したこと

本テーマでは 2 ポートの入出力を実装するために、ファイルディスクリプタの割り当て、inbyte, outbyte でスタックからの引数を取り方に気を付けてプログラムを作成した。また test ファイルでは 2 ポートからの入出力、セマフォの利用、タイマ割り込みの 3 つをすべて利用するようなプログラムの作成を行ったが、エスケープシーケンスを用いることで個人課題部分を工夫するだけでなく、デバッグ作業を効率的に行うことができた。

プログラムに発生した問題とその原因、解決方法

本テーマでは、ファイルディスクリプタを用いた 2 ポートへの割り当てが set_task を用いて登録したタスクに関しては行われない問題が発生した。csys68k.h 等の関連するソースコードを何度も見直したが、原因は分からなかった。そこまでは mon.o を SW 実験 1 で作成したエコーバックプログラムを改良した mon.s から作成していたものから、推奨されていた配布された mon.0 との差し替えをすると、解決した。原因が mon.s にあることは分からなかったが根本的な原因追求はできなかった。

個人課題について

test3.c ではマルチタスク環境を用いて、「仲間外れ探し（"a"という文字の中から文字"e"を探す）+チャット」を作成した。ここで **mtk_asm.s** で**タイマ割り込み周期を 1[0.1ms]**に設定しておく。このとき、登録した二つのタスクではそれぞれポート 0,1 から入力を受け付け、入力がなければ一方のタスクに遷移し、入力があれば資源（0）を占有し、入力値の削除・出力を行った後で資源（0）の解放を行っている。そうすることで疑似的に両ポートから同時入力を可能にしている。このとき、自分側の入力は右側に相手の入力は左側になるように出力する、LINE ライクな仕様になっている。

また、ポート 0 側で start と入力すると、力があれば資源（0）を占有し、入力値の削除・出力を行った後で資源の解放を行うのではなく、仲間外れ探しを開始する。ポート 0 側では仲間外れ探しをするときのマップの設定を横・縦の文字数で指定する。なお、このときにポート 1 からの入力値は資源がポート 0（のタスク）に占有されているため起こらない。ポート 0 の設定が終わると資源（0）を開放してゲームが始まる。このとき、再度両ポートからの入力をゲーム開始前と同様に受け付けるが、今度は毎度出力の旅に間違い"e"のある座標かどうか判定し、正しい座標が入ると答えが表示され、勝者が表示される仕様になっている。（厳密にいうと、どちらかが正しい入力を行い、ポートの入力を受け付けているタスク 1 が出力を行う際に答えが表示される）そして、またチャット機能へと戻り、何度でも仲間外れ探しをすることができる。

考察

本レポートで触れたテーマ 2,3 ではマルチタスク環境の作成とその応用を行った。入出力ストリーム関数のように便利な関数のある C 言語と、スタックやキューをレジスタ・メモリ単位で意識したプログラム作成が可能なアセンブリ言語それぞれの特性を生かして開発をすることができたと考えている。ただ、二つの言語を横断して、時に共同開発として分担して作業することで違う言語の関数を参照するときに問題が起こりやすいというのも事実であった。しかしテーマ 3 のファイルディスクリプタを用いて 2 ポート入出力のように流れを理解していないと実装できないものがあったからこそ理解が曖昧であったところが明確になり、学びにつながったと考えている。