2023/01/18 正午時点ですべて書き終わっていないので未完成です。
後日、完成版を改めて提出する予定です。
申し訳ございません。

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

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。

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、逆にタその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

```c
/*
* 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 ustack[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。

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 の捜索
        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;
}
```

```c
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_swtch
get_tcb_ssp:
                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_swtch:
        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

```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 (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);
  // taask3-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

プログラムに発生した問題とその原因、解決方法

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

```
```

inchrw.c

```
```

outchr.s

```
```

test3.c

```
```

プログラムの説明

プログラムにおいて、注意したこと

プログラムに発生した問題とその原因、解決方法

考察