ソフトウェア実験II・III

テーマ2,３　レポート

学籍番号：1TE22028G

氏名：一瀬遥希

提出日： 2025年1月23日

**テーマ2：マルチタスクカーネルの製作**

**1. テーマ2についての説明**

テーマ2では, C言語で記述された複数のタスクを適切なタイミングで切り替えながら実行する「マルチタスクカーネル」を作成する.

タスクの切り替えは以下の2つの状況で発生する.

1. 共有資源を利用する際の排他制御

同じメモリ領域やI/Oデバイスを複数のタスクが利用する場合, データ競合を防ぐためにセマフォを使用する. この際, P・Vシステムコールを利用し, リソースのロック（P命令）と解放（V命令）を管理することで, 安全なマルチタスク環境を構築する. また, Readyキュー（実行待ちタスクの管理）やセマフォキュー（特定の共有資源を待機しているタスクの管理）を活用し, タスクの状態を制御する.

1. CPUの独占防止のためのタイマ制御

一定時間経過後にタイマ割り込みを発生させ, 強制的にタスクを切り替える. これにより, 複数のタスクが公平にCPUを利用できるようになる.

さらに, 各タスクの状態を管理するために「タスクコントロールブロック（TCB）」を用いる. これにはプログラムカウンタ, レジスタ, スタックの内容などを保存・復元する機能が含まれる. また, カーネルがタスクの切り替え中に割り込みを受けないよう, 割り込みマスクを用いたカーネル再入防止の仕組みも含める.

これらを実現するうえで, 以下の基本事項への理解が重要である.

1. 特権命令の概念

OSにおいて, すべての命令が実行できるスーパバイザモードと一部の命令が実行できないユーザモードを理解し, スーパバイザモードでのみ実行できる特権命令を活用する. 特に, 入出力やタイマの設定などはこの特権命令として, 必要なときのみスーパバイザモードに切り替えて処理を行うことで, システムに重大な障害を与えるのを防ぐ事ができる.

1. P・Vシステムコールの理解

P・V 命令は, 複数のタスクが共有資源を同時に操作することでデータの整合性が崩れるのを防ぐための**排他制御**と**タスク同期**の手法である. セマフォはカウンタと待ち行列で構成され, P 命令は資源の取得（カウンタ減少）, V 命令は資源の解放（カウンタ増加）を行う. カウンタが負ならタスクを待機, 0なら継続, 正なら何もしない. P・V 命令により, 資源のロック管理やタスク間の処理順制御が可能となる. これらはOSの特権命令としてシステムコールで実装される.

1. タスク切り替えの仕組み

マルチタスク処理では, タスクの途中で実行が中断されても, 再開後の結果が切り替えなしの場合と同じになる必要がある. そのため, 中断時に計算機の状態を正しく保存し, 再開時にその状態を回復することが重要となる. 具体的には, 中断時の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し, 再開時に復元することで, タスクの継続実行を可能にする.

1. カーネル再入防止

タスクの切替えは, P命令による休眠状態への移行とタイマ割込みによって行われる. P 命令はユーザタスクのみが発行するため, カーネルタスクの実行中には発生しないが, タイマ割込みはカーネルタスクの実行中にも発生する可能性があり, 不適切なタイミングでの切替えにより管理情報が破壊される危険がある. この問題を防ぐため, カーネルタスクの実行中に発生したタイマ割込みではタスク切替えを行わず, 割込み前の状態に復帰する方法が一般的（UNIXの方式）だが, 今回の実験では, カーネルタスクの実行中はタイマ割込み自体を受け付けないことで対処する. 具体的には, P・V 命令処理のアセンブラルーチンの最初で割込みを禁止（走行レベル 7）し, 最後で元に戻すことで実現する. ただし, この方法ではタスクの連続実行時間の上限が厳密に一定とはならず, システムコール実行中はタスク切替えが遅れる可能性がある.

**2. テーマ2の実験全体における位置づけ**

テーマ2においては, マルチタスクカーネルの製作を通じて, タスクの並列実行をシミュレートし, OSが行うタスクスケジューリングの基本概念や共有資源に対する排他制御などを学ぶことに重点が置かれている. このように, テーマ2はOSの基本機能の習得を目的とした基礎フェーズであり, テーマ3において3つ以上のタスクを扱ったり, 2つのポートを使ってマルチタスキングを行えるように機能を拡張していくにあたっての重要な準備段階に位置づけられると考えられる.

**3. プログラムのリスト**

以下に, テーマ2において作成を行ったプログラムファイルmtk\_c.c, mtk\_c.h, mtk\_asm.s, test2.cのリストを示す.

**・mtk\_c.c**

#include <stdio.h>

#include "mtk\_c.h"

/\*======================================================

変数および配列の定義

========================================================\*/

SEMAPHORE\_TYPE semaphore[NUMSEMAPHORE]; /\* セマフォ \*/

TCB\_TYPE task\_tab[NUMTASK + 1];         /\* TCB配列 \*/

STACK\_TYPE stacks[NUMTASK];             /\* タスクスタック \*/

/\* 大域変数 \*/

TASK\_ID\_TYPE curr\_task;

TASK\_ID\_TYPE new\_task;

TASK\_ID\_TYPE next\_task;

TASK\_ID\_TYPE ready;

/\*======================================================

カーネルの初期化

init\_kernel:

引数 　: なし

返り値 : なし

担当　 : 若松

========================================================\*/

void init\_kernel()

{

  /\* TCB配列の初期化. TCB配列の各要素はTCB\_TYPE型の構造体 \*/

  for (int i = 0; i <= NUMTASK; i++)

  {                                /\* id = 0 は使わないが初期化は行っておく \*/

    task\_tab[i].status = EMPTY;    /\* status = EMPTY \*/

    task\_tab[i].next = NULLTASKID; /\* 次のタスクidはNULLTASKID \*/

    task\_tab[i].task\_addr = NULL;  /\* タスクアドレス = NULL \*/

    task\_tab[i].stack\_ptr = NULL;  /\* スタックアドレス = NULL\*/

    task\_tab[i].priority = 0;      /\* 優先度は0 \*/

  }

  /\* readyキューの初期化 \*/

  ready = 0; /\* 実行待ちタスクはない \*/

  /\* pr\_handler = TRAP #1 \*/

  \*(unsigned int \*)0x084 = (unsigned int)pv\_handler; /\* 関数名でアドレス参照 \*/

  /\* セマフォの値を初期化 \*/

  for (int i = 0; i < NUMSEMAPHORE; i++)

  {

    semaphore[i].count = 1;              /\* セマフォは専有されていない \*/

    semaphore[i].task\_list = NULLTASKID; /\* 初期化時点で次のタスクはない \*/

  }

}

/\*======================================================

ユーザタスクの初期化と登録

set\_task:

引数 : ユーザタスク関数へのポインタ(タスクの先頭番地)

返り値 : なし

担当：一瀬

========================================================\*/

void set\_task(void \*p)

{

  TASK\_ID\_TYPE i;

  for (int i = 1; i <= NUMTASK; i++)

  {

    if (task\_tab[i].status == EMPTY)

    {

      new\_task = i;

      break;

    } // タスクを走査し、空きスロット見つけたらそのIDをnew\_taskに代入

  }

  task\_tab[new\_task].task\_addr = p;                    // 空きスロットにユーザタスク関数のポインタを代入

  task\_tab[new\_task].status = OCCUPIED;                // スロットの使用状態を使用中に変更

  task\_tab[new\_task].stack\_ptr = init\_stack(new\_task); // init\_stackで初期化

  addq(&ready, new\_task);                              // readyキューにnew\_taskを登録

}

/\*==========================================================================

ユーザタスク用のスタックの初期化

init\_stack:

引数   : タスクID

返り値 : 初期化後にユーザタスク用SSPが指すアドレス(void \*型)

担当：若松、一瀬（共同）

===========================================================================\*/

void \*init\_stack(TASK\_ID\_TYPE id)

{

  int \*int\_ssp;

  int\_ssp = (int \*)&stacks[id - 1].sstack[STKSIZE]; // Sスタックの底+1のアドレスをポインタに代入

  \*(--int\_ssp) = (int)task\_tab[id].task\_addr; // 初期PCを積む

  unsigned short int \*short\_ssp = (unsigned short int \*)int\_ssp;

  \*(--short\_ssp) = 0x0000; // 初期SRを積む

  int\_ssp = (int \*)short\_ssp;

  int\_ssp -= 15;                                  // 15×4バイト分の領域を飛ばす（レジスタ群を積む）

  \*(--int\_ssp) = &stacks[id - 1].ustack[STKSIZE]; // 初期USPを積む

  return (void \*)int\_ssp; // 関数終了時のSSPが返り値

}

/\*==========================================================================

マルチタスク処理の開始

begin\_sch:

引数 : なし

返り値 : なし

担当 : 若松

===========================================================================\*/

void begin\_sch()

{

  curr\_task = removeq(&ready); /\* removeqで最初のタスクを設定 \*/

  init\_timer();                /\* init\_timerでタイマの初期化 \*/

  first\_task();                /\* first\_taskで最初のタスクを起動 \*/

}

/\*=============================================================

タスクキューの最後尾へのTCBの追加

addq:

引数 : キューへのポインタとタスクのID

返り値 : なし

担当：一瀬

===============================================================\*/

void addq(TASK\_ID\_TYPE \*que\_ptr, TASK\_ID\_TYPE id)

{

  if (\*que\_ptr == NULLTASKID)

  {                // キューの先頭のタスクが空なら,

    \*que\_ptr = id; // 先頭にタスクを登録

  }

  else

  {

    TCB\_TYPE \*task\_ptr = &task\_tab[\*que\_ptr]; // 先頭のタスクのポインタ設定

    while (1)

    {

      if ((\*task\_ptr).next == NULLTASKID)

      {                        // その次のタスクが空だったら,

        (\*task\_ptr).next = id; // タスクを登録

        break;

      }

      else

      {

        task\_ptr = &task\_tab[(\*task\_ptr).next]; // 次のタスクにポインタを移動

      }

    }

  }

}

/\*=============================================================

タスクキューの先頭からTCBの除去

removeq:

引数 : キューへのポインタ

返り値 : 先頭のID

担当：一瀬

===============================================================\*/

TASK\_ID\_TYPE removeq(TASK\_ID\_TYPE \*que\_ptr)

{

  TASK\_ID\_TYPE r\_id = \*que\_ptr; // 返却値（先頭のタスクのidを取得）

  if (r\_id != NULLTASKID)

  {                                       // キューの先頭が空でなければ

    TCB\_TYPE \*task\_ptr = &task\_tab[r\_id]; // 先頭のタスクのポインタ設定

    \*que\_ptr = (\*task\_ptr).next;          // 先頭から2番目のタスクを先頭にする

    (\*task\_ptr).next = NULLTASKID;        // 先頭のタスクのnextはNULLTASKIDにして、タスクを取り出す

  }

  return r\_id; // キューの先頭のタスクのidを返す

}

/\*=============================================================

タスクのスケジュール関数

sched:

引数 : なし

返り値 : なし

担当 : 若松

===============================================================\*/

void sched()

{

  next\_task = removeq(&ready); /\* readyキューからタスクを取り出しnext\_taskに代入 \*/

  while (next\_task == NULLTASKID)

  {

    while (1)

      ;

  } /\* NULLTASKIDの場合は無限ループ \*/

}

void p\_body(int id)

{

  semaphore[id].count--; /\*セマフォの値をデクリメント\*/

  if (semaphore[id].count < 0)

  {

    /\*その結果セマフォが獲得できないなら\*/

    sleep(id); /\*セマフォ待ちのキューに入れ休眠状態へ\*/

  }

}

void v\_body(int id)

{

  semaphore[id].count++; /\*セマフォの値をインクリメント\*/

  if (semaphore[id].count <= 0)

  {             /\*その結果セマフォが空いたなら\*/

    wakeup(id); /\*そのセマフォを待っている先頭のタスクを実行可能状態へ\*/

  }

}

int sleep(int a)

{ /\* a := セマフォID\*/

  addq(&(semaphore[a].task\_list), curr\_task);

  sched();

  swtch();

}

int wakeup(int a)

{

  TASK\_ID\_TYPE task\_id = removeq(&(semaphore[a].task\_list));

  addq(&ready, task\_id);

}

**・mtk\_c.h**

#ifndef MTK\_C\_H

#define MTK\_C\_H

// カーネル関連の定数定義///////////////////////////////////////////

#define NUMSEMAPHORE 5

#define NULLTASKID 0

#define NUMTASK 5

#define STKSIZE 1024

// task\_tab.status 定数////////////////////////////////////////////

#define EMPTY 0

#define OCCUPIED 1

#define COMPLETED 2

// 構造体の定義//////////////////////////////////////////////////

typedef int TASK\_ID\_TYPE; /\* TASK\_ID\_TYPEの定義 \*/

typedef struct

{

  int count;

  int nst; /\* reserved \*/

  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、配列の定義///////////////////////////////////////////

extern TASK\_ID\_TYPE curr\_task;

extern TASK\_ID\_TYPE new\_task;

extern TASK\_ID\_TYPE next\_task;

extern TASK\_ID\_TYPE ready;

extern SEMAPHORE\_TYPE semaphore[NUMSEMAPHORE];

extern TCB\_TYPE task\_tab[NUMTASK + 1];

extern STACK\_TYPE stacks[NUMTASK];

// 関数のプロトタイプ宣言///////////////////////////////////////////

extern void outbyte(int ch, unsigned char c);

extern char inbyte(int ch);

/// マルチタスクカーネル関連の関数

extern void init\_kernel();

extern void set\_task(void \*p);

extern void \*init\_stack(TASK\_ID\_TYPE id);

extern void begin\_sch();

extern void addq(TASK\_ID\_TYPE \*que\_ptr, TASK\_ID\_TYPE id);

extern TASK\_ID\_TYPE removeq(TASK\_ID\_TYPE \*que\_ptr);

extern void sched();

extern void first\_task();

extern void swtch();

/// タイマ関連の関数

extern void hard\_clock();

extern void init\_timer();

///// モニタ内の関数

extern void set\_timer();

extern void reset\_timer();

/// セマフォ関連の関数

extern void p\_body(int a);

extern void v\_body(int a);

extern int sleep(int a);

extern int wakeup(int a);

extern void P(int id);

extern void V(int id);

extern void pv\_handler();

extern void skipmt();

#endif

**・mtk\_asm.s**

\*\*global変数の宣言

.global first\_task

.global swtch

.global hard\_clock

.global init\_timer

.global skipmt

.global P

.global V

.global pv\_handler

\*\*外部入力（大域変数）

.extern curr\_task

.extern next\_task

.extern ready

.extern task\_tab

\*\*外部関数

.extern addq

.extern sched

.extern p\_body

.extern v\_body

\*\*システムコール番号

.equ SYSCALL\_NUM\_GETSTRING, 1

.equ SYSCALL\_NUM\_PUTSTRING, 2

.equ SYSCALL\_NUM\_RESET\_TIMER,   3

.equ SYSCALL\_NUM\_SET\_TIMER, 4

.equ SYSCALL\_NUM\_SKIPMT, 5

.equ PV\_CALL\_P,         0

.equ PV\_CALL\_V,         1

.section .text

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\*\* ユーザタスク起動用ルーチン

\*\* first\_task

\*\* 入出力なし

\*\* 担当：一瀬

\*\*

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

first\_task:

    move.l  curr\_task, %d0             /\*curr\_taskの番号をd1に\*/

    movea.l #task\_tab, %a0             /\*task\_tab配列の先頭アドレスをa0に\*/

    mulu.w  #20, %d0                  /\*curr\_taskの番号に20を乗算し、d1に格納\*/

    add.l   %d0, %a0                   /\*a0にd1を加算し、curr\_taskの先頭アドレスを計算\*/

    movea.l %a0, %a1

    add.l   #4,  %a1                   /\*a0は該当のcurr\_taskのstack\_ptr(SSP）の先頭アドレス\*/

    move.l  (%a1), %sp                 /\*スーパーバイザーモードのspにSSPを回復\*/

    move.l  (%sp)+, %a2

    move.l  %a2, %USP

    movem.l (%sp)+, %d0-%d7/%a0-%a6    /\*レジスタ15本回復\*/

    rte

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\*\* タスクスイッチを実際に起こす関数

\*\* swtch

\*\* 入出力なし

\*\* 担当：若松

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

swtch:

    move.w  %SR,-(%sp)                 /\* SRを退避しrteでの復帰を可能に \*/

    movem.l %d0-%d7/%a0-%a6,-(%sp)     /\* 実行中タスクのレジスタ退避 \*/

    move.l  %usp, %a0

    move.l  %a0, -(%sp)

    move.l  curr\_task, %d1             /\* curr\_task -> d1 \*/

    mulu.w  #20, %d1                   /\* TCB配列の各要素は4\*5=20byte, タスクidに乗算 \*/

    movea.l #task\_tab, %a0             /\* task\_tab配列の先頭アドレス -> a0 \*/

    add.l   %d1, %a0                   /\* task\_tab配列内のcurr\_taskの先頭アドレスまで移動 \*/

    add.l   #4,  %a0                   /\* stack\_ptr(SSP）の先頭アドレス -> a0 \*/

    move.l  %sp, (%a0)                 /\* 現在のタスクのTCBにSSPを記録\*/

    move.l  next\_task, %d1             /\* next\_task-> d1 \*/

    move.l  %d1, curr\_task             /\* curr\_taskをnext\_taskで更新\*/

    mulu.w  #20, %d1                   /\* TCB配列の各要素は4\*5=20byte, タスクidに乗算 \*/

    movea.l #task\_tab, %a0             /\* task\_tab配列の先頭アドレス -> a0\*/

    add.l   %d1, %a0                   /\* task\_tab配列内のcurr\_taskの先頭アドレスまで移動\*/

    add.l   #4,  %a0                   /\* stack\_ptr(SSP）の先頭アドレス -> a0 \*/

    move.l  (%a0),%sp                  /\* スーパバイザモードのspにSSPを回復 \*/

    move.l  (%sp)+, %a0

    move.l  %a0, %usp

    movem.l (%sp)+,%d0-%d7/%a0-%a6     /\*次のタスクのレジスタを回復\*/

    rte

hard\_clock:

    movem.l %d0-%d7/%a0-%a6, -(%sp)

    move.l  #ready, %d0         |%d0 -> readyキューへのポインタ

    move.l  curr\_task, %d1          |%d1 -> タスクのID

    movem.l %d0-%d1, -(%sp)     |%d0, %d1をスタックに積んで

    jsr addq                |addqを実行

    adda.l  #8, %sp         |%d0, %d1の8バイト分を%spに加算

    jsr sched

    jsr swtch

    movem.l (%sp)+, %d0-%d7/%a0-%a6

    rts

init\_timer:

    movem.l %d0-%d2, -(%sp)

    move.l #SYSCALL\_NUM\_RESET\_TIMER, %d0    | タイマーをリセット

    trap #0

    move.l #SYSCALL\_NUM\_SET\_TIMER, %d0      | タイマーをセット

    move.w #10000, %d1                                                  | およそ１秒くらい

    move.l #hard\_clock, %d2                                        | 割り込み時に呼び出すルーチンをセット

    trap #0

    movem.l (%sp)+,%d0-%d2

    rts

skipmt:

    movem.l %d0-%d2, -(%sp)

    move.l #SYSCALL\_NUM\_SKIPMT, %d0

    trap    #0

    movem.l (%sp)+,%d0-%d2

    rts

P:

    link.w  %a6,#0

    movem.l %d0-%d1,-(%sp)  /\*レジスタ退避\*/

    move.l  #0,%d0          /\*PシステムコールIDの0をd0レジスタにセット\*/

    move.l  8(%a6),%d1      /\*スタックから取り出した引数（セマフォID）をd1レジスタにセット\*/

    trap    #1

    movem.l (%sp)+,%d0-%d1  /\*レジスタ復帰\*/

    unlk    %a6

    rts

V:

    link.w  %a6,#0

    movem.l %d0-%d1,-(%sp)  /\*レジスタ退避\*/

    move.l  #1,%d0          /\*VシステムコールIDの1をd1レジスタにセット\*/

    move.l  8(%a6),%d1      /\*スタックから取り出した引数（セマフォID)をd1レジスタにセット\*/

    trap    #1

    movem.l (%sp)+,%d0-%d1  /\*レジスタ復帰\*/

    unlk    %a6

    rts

pv\_handler:

    movem.l %a0-%a6/%d0-%d7, -(%sp)

    move.w  %SR,-(%sp)  /\*SRをスタックに退避\*/

    move.w  #0x2700,%SR /\*走行レベルを７にして割り込み禁止\*/

    movem.l %d1,-(%sp)  /\*レジスタd1をスタックに退避\*/

    cmpi.l  #0,%d0      /\*d0の値が0であるか比較\*/

    bne pv\_handler\_1    /\*d0が0でないならば分岐\*/

    jsr p\_body          /\*p\_body()を呼び出す\*/

    bra pv\_handler\_end  /\*復帰処理へ\*/

pv\_handler\_1:

    cmpi.l  #1,%d0      /\*d0の値が1であるか比較\*/

    bne pv\_handler\_end  /\*d0が1でないならば分岐\*/

    jsr v\_body          /\*v\_body()を呼び出す\*/

pv\_handler\_end:

    movem.l (%sp)+,%d1  /\*レジスタd1をスタックから復帰\*/

    move.w  (%sp)+,%SR  /\*SRをスタックから復帰\*/

    movem.l (%sp)+, %a0-%a6/%d0-%d7

    rte                 /\*割り込み終了\*/

**・test2.c**

#include <stdio.h>

#include "mtk\_c.h" // マルチタスクカーネル用ヘッダー

int N=3;

int K=10000;

volatile int nttask;

int task\_0(){

    while(1){

        printf(" task0 is runnnig\n");

        P(0);

        if(nttask == N){

            nttask=0;

            for (int i=0; i<N; i++){

                printf("  do V(1) ,%d times\n",i+1);

                V(1);

            }

            printf("tasks reset\n");

        }

        V(0);

        printf("jmp to skipmt\n");

        skipmt();

        printf("task0 is back\n");

    }

}

int task1(){

    while(1){

        printf(" task1 start\n");

        for(int k=0; k<K; k++){

            if(k%1000==0){

                printf("task1 ,%d times\n",k/1000);

            }

        }

        printf(" task1 is runnnig\n");

        P(0);

        nttask++;

        printf("nttask is %d now\n",nttask);

        V(0);

        printf("              task1 is finished\n");

        P(1);

    }

}

int task2(){

    while(1){

        printf(" task2 start\n");

        for(int k=0; k<2\*K; k++){

            if(k%1000==0){

                printf("task2 ,%d times\n",k/1000);

            }

        }

        printf(" task2 is runnnig\n");

        P(0);

        nttask++;

        printf("nttask is %d now\n",nttask);

        V(0);

        printf("              task2 is finished\n");

        P(1);

    }

}

int task3(){

    while(1){

        printf(" task3 start\n");

        for(int k=0; k<3\*K; k++){

            if(k%1000==0){

                printf("task3 ,%d times\n",k/1000);

            }

        }

        printf(" task3 is runnnig\n");

        P(0);

        nttask++;

        printf("nttask is %d now\n",nttask);

        V(0);

        printf("             task3 is finished\n");

        P(1);

    }

}

int main() {

    printf("Initializing kernel...\n");

    init\_kernel();

    printf("Kernel initialized.\n");

    semaphore[0].count =1;

    semaphore[1].count =0;

    nttask = 0;

    printf("Setting Task 0...\n");

    set\_task((char \*)task\_0); // タスクを登録

    printf("Task 0 set.\n");

    printf("Setting Task 1...\n");

    set\_task((char \*)task1); // タスクを登録

    printf("Task 1 set.\n");

    printf("Setting Task 2...\n");

    set\_task((char \*)task2); // タスクを登録

    printf("Task 2 set.\n");

    printf("Setting Task 3...\n");

    set\_task((char \*)task3); // タスクを登録

    printf("Task 3 set.\n");

    printf("Starting scheduler...\n");

    begin\_sch();

    // begin\_sch後のコードは実行されないはず

    printf("Scheduler started (this line should not be printed).\n");

    return 0;

}

/\* exit() defined in test\*.c \*/

void exit(int value) {

    \*(char \*)0x00d00039 = 'h';  /\* led0 への表示 (halt) \*/

    for (;;)

    ;               /\* 無限ループトラップで停止させる \*/

}

**4. プログラムの説明**

　以下に, 上記に示したプログラムファイル毎に, プログラムの説明を示す.

**mtk\_c.c**

**・カーネルの初期化：init\_kernel();**

この関数では, 引数を取らず, TCB 配列・readyキュー・セマフォの値の初期化を行い，pv\_handlerをTRAP #1の割り込みベクタに登録する. なお, TCB配列の各要素はTCB\_TYPE型の構造体である. ここでは, 後々スケジューリングなどに使用することも踏まえて, TCBの使用状態を記号定数としてヘッダファイルに宣言し, 可読性の工場に努めている.

**・ユーザタスクの初期化と登録：set\_task();　担当部**

引数には, ユーザタスク関数へのポインタ（タスク関数の先頭番地）void \*pを取る. まず, task\_tab[]を走査して空きスロットを見つけ, そのIDをnew\_taskに代入する. つぎに, 上で見つけたTCBの更新を行い, スロットを使用中に変更する. その後, init\_stack(); の値を, そのTCBのstack\_ptrに登録し, 最後にaddq(); を用いて, readyキューにnew\_taskを登録する.

**・ユーザタスク用のスタックの初期化：init\_stack(); 担当部**

引数には, タスクIDを取り, 初期化後のスーパバイザスタックポインタ（SSP）が指すアドレスを void\* 型で返す。以下に処理の詳細な流れを箇条書きで示す.

1. スタックのアドレスを設定

タスクidに対応するstacks[id - 1].sstackのスタック底（+1の位置）をポインタint\_ssp に格納する.

2. 初期プログラムカウンタ（PC）を積む

task\_tab[id].task\_addr（タスクの開始アドレス）をスタックにプッシュし, 初期PCとして設定する.

3. 初期ステータスレジスタ（SR）の設定

0x0000 をプッシュし, SR（ステータスレジスタ）を初期化する.

4. レジスタ領域の確保

レジスタ保存用に15×4バイト分スタックを確保（スタックポインタを進める）.

5. 初期ユーザスタックポインタ（USP）の設定

stacks[id - 1].ustack[STKSIZE]（ユーザスタックのトップ）をプッシュし, USP（ユーザタスク用スタックポインタ）を設定する.

6. スーパバイザスタックポインタ（SSP）の返却

初期化完了時の int\_ssp を void\* 型で返す.

処理の中で, \*(--ssp) = 値の形式を用いることで, スタックに値をプッシュする際に, スタックポインタをデクリメントしながら代入する動作を簡単に実装している. また, int型ポインタと unsigned short int 型ポインタを適切に使い分けることで, 4バイト（整数）や2バイト（ステータスレジスタ）といった異なるデータサイズを正しく格納できるようにしている.

**・マルチタスク処理の開始 begin\_sch();**

　この関数は, 引数を取らない. まず, removeq()関数で readyキューから最初のタスクを取り出し, curr\_taskに代入する. 次に，init\_timer(); でタイマを設定する. そして, first\_task(); で最初のタスクを機動する.

**・タスクキューの最後尾へのTCBの追加：addq(); 担当部**

　引数として, キューへのポインタとタスクのIDを受け取る. 実際に作成した処理の流れの詳細を以下に箇条書きで示す.

1. キューが空の場合（\*que\_ptr == NULLTASKID）

先頭のタスクが存在しないため, キューの先頭に id を登録する.

2. キューにタスクがすでに存在する場合

task\_tab[\*que\_ptr] でキューの先頭タスクの TCB（タスク制御ブロック）\*\*を取得し,

最後尾のタスクを探すために, next が NULLTASKID（次のタスクがない）までループを行う. 最後尾のタスクが見つかったら, そのnextにidを登録し, タスクをキューの最後尾に追加する.

この関数は, タスクキューの管理を行い, 新しいタスクを適切に最後尾へ追加する役割を持つ. キューが空の場合は, 新しいタスクを先頭に直接登録し, すでにタスクが存在する場合は, 最後尾のnextを更新して新しいタスクを接続する. while (1)ループを用いてキューの末尾を探索し, ポインタを適切に更新しながら処理を進めることで, リストの順序を維持する仕組みとなっている. これにより, タスクの実行順を保ちながら, 新しいタスクをスムーズにキューへ追加できるようになっている.

**・タスクのスケジュール関数：sched();**

　この関数は, 引数を取らない. removeq(); でreadyキューの先頭のタスクIDを取り出し，next\_taskに代入する. ただし, readyキューが空の場合は無限ループに入る.

**・Pシステムコールの本体：p\_body();**

引数には, セマフォIDを取る. セマフォの値をデクリメントし, セマフォが獲得できなければ, sleep(セマフォID)を実行し, 休眠状態に入る.

**・Vシステムコールの本体：v\_body();**

　引数には, セマフォIDを取る. セマフォの値をインクリメントし, セマフォが空けば, wakeup(セマフォID)を実行し, そのセマフォを持っているタスクを実行可能状態にする.

**・タスクを休眠状態にしてタスクスイッチをする：sleep(int a);**

　引数には, チャンネルchを取る. (ここでは, 作成の都合上簡単のためにint aとしている. ) スケジューラsched(); を起動して, 次に実行するタスクのタスクIDをnext\_taskにセットする. その後, タスクスイッチ関数であるswtchを呼び出し, タスクの切り替えを行う.

**・休眠状態のタスクを実行可能状態にする：wakeup(int a);**

　引数には, チャンネルchを取る. (ここでは, 作成の都合上簡単のためにint aとしている. ) まず，removeq(); でチャンネルchの待ち行列に繋がっている先頭のタスクを取り除き，その ID をTASK\_ID\_TYPE task\_id に代入する. 次に, 先程のtask\_idを利用して，addq(); でそのタスクをready キューに繋ぎ直して, そのタスクを実行可能状態にする.

**mtk\_c.h**

mtk\_c.hは, マルチタスクカーネルの構築に必要なタスク管理, スケジューリング, セマフォ制御, タイマ管理のための構造体, 変数, 関数を定義した重要なヘッダファイルである.

このヘッダファイルを使用することで, C 言語ベースのマルチタスク環境を効果的に実装できる. 以下に, このヘッダファイル内で定義されているものの説明を示す.

**・カーネル関連の定数の定義**

NUMSEMAPHORE：使用可能なセマフォの最大数

NULLTASKID：タスクキューが空であることを示す値

NUMTASK：許容できる最大タスク数

STKSIZE：タスクスタックのサイズ(1KBとしている）

**・task\_tab.status 定数**

EMPTY：タスクスロットが未使用

OCCUPIED：タスクスロットが使用中

COMPLETED：タスクが終了した状態

**・構造体の定義**

　ここでは, マルチタスクの制御に必要なデータを管理するため**構造体**が定義されている.

TASK\_ID\_TYPE：

タスクIDをint型で定義している.

SEMAPHORE\_TYPE：

count：セマフォのカウント(リソースの使用可能数の管理用).

nst：予約領域(今回は未使用）

task\_list：セマフォ待機キューの先頭タスクID (セマフォを待つタスクの管理用).

TCB\_TYPE：

task\_addr：タスクが実行する関数のアドレス(エントリーポイント）

\*stack\_ptr：タスクのスタックポインタ

priority：タスクの優先度

status：タスクの使用状態を表す (EMPTY, OCCUPIED, COMPLETED）

next：次のタスクID (キューとして管理するためのポインタ）

STACK\_TYPE：

ustack：ユーザモード用のスタック領域

sstack：スーパバイザモード用のスタック領域

**・大域変数, 配列の定義**

　ここでは, カーネルの状態やタスク情報を管理するグローバル変数が定義されている.

curr\_task：現在実行中のタスクを示す

new\_task：新しく登録されるタスクID

next\_task：次に実行するタスクID

ready：実行可能なタスクのキューの先頭ID

semaphore：セマフォ情報を保持する配列 (最大 NUMSEMAPHORE 個)

task\_tab：タスクの管理情報を格納するTCBテーブル (NUMTASK + 1)

stacks：タスクのスタック領域

**・関数のプロトタイプ宣言**

　カーネルの基本機能を提供するための関数が定義されている.

**mtk\_asm.s**

　このプログラムファイルには, マルチタスクカーネルのうちアセンブリ言語で構成される関数を記述している.

**・ユーザタスク起動サブルーチン：first\_task 担当部**

　このルーチンは, カーネルが使用しているスタックを現在のタスク (curr\_task) のスタックに切り替え, マルチタスク処理を開始するためのサブルーチンである. また, このルーチンはスーパバイザモードで実行される必要がある. 以下に, 詳細な処理の流れを示す.

1. curr\_taskのTCB（タスクコントロールブロック）アドレスを取得

curr\_task は現在のタスク ID（整数値）であり, これをタスクテーブルtask\_tabの適切なインデックスに変換します. task\_tabの各エントリ（TCB）のサイズは20バイトであるため, curr\_task の番号を20倍して適切なTCBアドレスを求めている.

2. SSP（スーパバイザスタックポインタ）の回復

curr\_taskのTCBのstack\_ptrに保存されていたSSPをSP（スーパバイザスタックポインタ）に復元する. これにより, 現在のタスクが実行を再開できる状態になる.

3. USP（ユーザスタックポインタ）の回復

スーパバイザスタックに保存されていたUSP（ユーザスタックポインタ）を復元する. これにより, タスクがユーザモードで実行するときに使用するスタックが正しく設定される.

4. 残りのレジスタの回復

movem.l 命令を使用して, スーパバイザスタックからd0-d7と a0-a6を復元する. これにより, タスクが中断された時点の状態を完全に再現できる.

5. ユーザタスクの起動

RTE（Return from Exception）命令を実行することで, カーネルモードからユーザタスクの実行を開始する. RTE は, 保存されたプログラムカウンタ（PC）とステータスレジスタ（SR）を復元し, タスクの実行を再開する.

**・タスクスイッチ関数：swtch**

　このルーチンは, 現在のタスクを中断し、次に実行するタスクへ切り替える処理を行う関数です。sleep() やタイマ割り込み (hard\_clock()) によって呼び出され、タスクのレジスタやスタックの状態を保存・復元することで、タスクの切り替えを実現します。以下に, 詳細な処理の流れを示す.

1. SR（ステータスレジスタ）を保存

RTE による復帰を可能にするため, 現在の SR をスタックに積む.

2. 現在のタスクのレジスタを保存

d0-d7, a0-a6, USPをスーパバイザスタックに退避する.

3. 現在のタスクのSSPをTCBに保存

curr\_taskのstack\_ptrに現在のSSPを記録する.

4. curr\_task を next\_task に更新

next\_taskに設定されている次のタスクIDをcurr\_taskに代入する.

5. 次のタスクの SSP を復元

新しいcurr\_taskのstack\_ptrから SSP を回復する.

6. 次のタスクのレジスタを復元

USP, d0-d7, a0-a6をスーパバイザスタックから復元する.

7. タスクの切り替えを実行

RTEにより, 次のタスクの実行を開始する.

**・タイマ割り込みルーチン：hard\_clock**

　このルーチンは, タイマ割り込みを行うルーチンである. まず, レジスタの対比を行い, addq(); で現タスクをreadyキューの末尾に追加する. そして, sched(); およびswtchを起動する.

**・クロック割り込みルーチン：init\_timer**

　このルーチンは, **タイマをセットし、一定周期ごとにハードウェア割り込みを発生させる**ためのルーチンである. まず d0-d2 レジスタをスタックに保存し、タイマ設定の影響を受けないようにする。その後、SYSCALL\_NUM\_RESET\_TIMER を d0 に設定し、trap #0 を実行してタイマをリセットする。続いて、SYSCALL\_NUM\_SET\_TIMER を d0 に設定し、d1 = 10000（約1秒の割り込み周期）と d2 = hard\_clock（割り込み時の処理ルーチン）を指定して、trap #0 によりタイマを有効化する。最後に d0-d2 のレジスタを復元し、rts で関数を終了する。

**・低難度Option：skipmt 担当部**

　このルーチンは, システムコール番号 5 (SYSCALL\_NUM\_SKIPMT) を使用し, trap #0 を実行することで特定のマルチタスク処理をスキップするルーチンである.

**・Pシステムコールの入口：P**

　このルーチンは, C言語からセマフォIDを引数として呼び出され, P操作（セマフォ取得）をシステムコール trap #1 を通じて実行するためのルーチンである. まずa6をスタックフレームとして確保し, d0-d1のレジスタをスタックに保存する. 次に, d0にPシステムコールのID（0）をセットし, スタックから取得したセマフォIDをd1に格納する. 次に, trap #1を実行してカーネルのPシステムコール(p\_body())を呼び出し, セマフォの状態に応じてタスクの実行を継続または休眠させる. 処理終了後, d0-d1のレジスタを復元し, a6 のスタックフレームを解除した後, rtsにより関数を終了し, C言語の呼び出し元へ戻る.

**・Vシステムコールの入口：V**

　このルーチンは, C言語からセマフォIDを引数として呼び出され, V操作（セマフォ解放）をシステムコール trap #1 を通して実行するためのルーチンである. タスクが使用していたセマフォを解放し, 待機中のタスクがあれば実行可能状態にする. まずa6をスタックフレームとして確保し, d0-d1のレジスタをスタックに保存する. 次に. d0にVシステムコールのID（1）をセットし, スタックから取得したセマフォIDをd1に格納する. trap #1 を実行してカーネルのVシステムコール(v\_body())を呼び出し, セマフォのカウントを増加させ, 待機中のタスクがあれば実行可能状態にする. 処理終了後, d0-d1のレジスタを復元し, a6のスタックフレームを解除した後, rtsにより関数を終了し, C言語の呼び出し元へ戻る.

**・TRAP #1 割り込み処理ルーチン：pv\_handler**

　このルーチンは, PまたはVのシステムコール(trap #1)が実行されたときに割り込み処理を行うルーチンであり, スーパバイザモードで実行される. C言語のpv\_handler()として直接呼び出してはならず, PまたはVのTRAP #1の実行により動作する.

D0レジスタには PまたはVのシステムコールの種類が, D1レジスタにはセマフォIDがセットされた状態で呼び出される. このルーチンは, D0の値に応じてp\_body(ID)または v\_body(ID)を呼び出し, セマフォ操作を実行する. 詳細な処理の流れを以下に示す.

1. レジスタを保存

割り込み処理の影響を受けないように, アドレスレジスタ(a0-a6)とデータレジスタ (d0-d7)をスーパバイザスタックに退避する. これにより, 処理が終了した後に元のタスクの状態を正しく復元できるようにする.

1. 割り込み禁止

SR（ステータスレジスタ）をスタックに保存し, SR=0x2700を設定して走行レベルを7に変更することで, 割り込みを禁止する. これにより, 割り込み処理の最中に別の割り込みが発生して処理が競合するのを防ぐ.

1. p\_body() または v\_body() の呼び出し

D0レジスタの値を確認し, 以下のように処理を分岐させる：

・D0 == 0 の場合, D1（セマフォ ID）を引数としてp\_body(D1)を実行する。

これはP操作（セマフォ取得）を行い, セマフォが使用可能でない場合はタスクを休眠状態にする可能性がある.

・D0 == 1 の場合, D1を引数としてv\_body(D1)を実行する.

これはV操作（セマフォ解放）を行い, 待機しているタスクがあれば実行可能状態に戻す.

1. 割り込み禁止の解除

SRをスタックから復元し, 元の割り込みレベルに戻す. これにより, 通常の割り込み処理が再開され, システムの割り込み管理が正常に動作する.

1. レジスタの復元

割り込み処理開始時にスタックに保存したレジスタを復元し, 元のタスクの状態を回復する.

1. 割り込み処理の終了

RTE（Return from Exception）命令を実行し, 割り込み処理を終了する. これにより, 処理が中断されたタスクまたは新たにスケジュールされたタスクが再開される.

**test2.c**

　このプログラムファイルには, セマフォを用いたタスクの同期を検証するためのテストプログラムが記述されている. これは, テキスト1.4.5節のtrap #0 のシステムコール番号5の機能追加実装及び2.5.6節の低難度 Option: skipmt() の実装を踏まえて, タスクの同期処理を確認できるように実装したものである. そのため, まずどのようにタスクの同期を実現したかについて説明を行う.

**セマフォ応用**

**タスクの同期処理**

本実装では, 複数のタスク(task\_k(), task\_0()⋯)をセマフォを用いて同期させる仕組みを構築する. ここでは, 各タスク(task\_k(), 1 ≤ k ≤ N)の終了順は順不同だが, 全タスクが停止した後に一斉に再開できるように実装を行う.

同期の仕組み

1. 共有変数とセマフォの準備
   * nttask（終了したタスク数）を共有変数として定義.
   * semaphore[0]（カウンティングセマフォ）でnttaskの更新を保護.
   * semaphore[1]（バイナリセマフォ）でタスクの待機・再開を制御.
2. タスクの動作
   * 各タスク(task\_k())は作業終了後, クリティカルセクション(P(0), V(0))で nttask を更新し, P(1)で待機状態に入る.
3. 監視タスク (task\_0()) の役割
   * nttask == N（全タスクが停止）を検出すると, nttask = 0にリセットし, V(1) をN回実行して全タスクを再開する.
   * skipmt()を実行し, CPU資源を節約.

この仕組みにより, 各タスクの終了順に関係なく, 全タスクの再開を統一的に管理することができる.

このtest2.c内では, main()でカーネルの初期化, セマフォの設定, タスクの登録を行い,スケジューラ (begin\_sch())を開始する. タスクの実行中, セマフォを用いてタスク間の同期を制御し, 全タスクが停止した後に一斉に再開できるようにしている.

以下に, スケジューラに登録しているタスクの詳細を示す.

task\_0（監視タスク）

他のタスク(task1, task2, task3)の進行を監視し, 全タスクが停止(nttask == N)したら, 再開 (V(1)) を行う, タスクの同期処理後, skipmt()を実行し, 計算資源を解放する.

task1, task2, task3（作業タスク）

各タスクは異なるループ (K, 2K, 3K回)を実行するように記述されており, 処理の負荷がそれぞれ異なる値に設定されている. 各ループ終了後, P(0)でクリティカルセクションを確保し, nttaskをインクリメント(nttask++)し, 実行終了を通知するような動作をする. そしてすぐにV(0)でクリティカルセクションを解放し, P(1)で待機状態に入る. 全タスクが終了するまでそれぞれ待機し, task\_0によってV(1)が呼ばれると, 再び動作を開始する.

　最後に, 実際にこのタスクを実行した際の結果を示す.

テキスト

自動的に生成された説明

テキスト

低い精度で自動的に生成された説明

テキスト

自動的に生成された説明

テキスト

中程度の精度で自動的に生成された説明

これを見ると, 処理の負荷の異なる3つのタスクがタイマ割り込みによって一定時間ずつ実行されているが, 処理が進むにつれ負荷の軽い処理から終了し, 残りのタスクのみが切り替えられ, すべてのタスクが終了した際に, 監視タスクが働き, 同期した後再開されており, 上に示した同期処理が実現できていることがわかる.

**5. プログラムにおいて注意したこと**

**1. スタックとタスクコントロールブロックの理解**

　テーマ2のマルチタスクカーネル作成や実際にタスクを用意してテストを行っていく中で, スタックとその情報の管理に関する理解は不可欠であり, テーマ2においてもっとも時間をかけた作業であった. 特に, タスクコントロールブロックの持つ情報などをスタックに積んでいく際のアドレス計算や, 積まれ方の仕様などは図を用いて念入りに情報を共有して, 班員全員が動作を理解して実装できるように注意した.

**2. セマフォに関する理解**

　マルチタスキングを実装するなかで, タスクの同期処理を実現するために, 本実験ではバイナリセマフォとカウンティングセマフォを使い分ける必要があった. もちろんセマフォの操作をするための関数を正確に作成するのはもちろんのこと, クリティカルセクションなど, タスクがどのようにセマフォを利用していくのかを構造建てて考える必要があり, これも班員で時間をかけて話し合い理解を行った.

**3. ヘッダファイルの整理**

　我々の班では, 作業を分野ごとに振り分け, 各々が違う関数を作成するような方法を取った. そのため, 各関数の引数や変数の定義などについての一貫性を保つためにヘッダファイルで適切にそして整理して宣言し, 矛盾ができるだけ発生しないように注意した.

**4. クラウドを利用した情報の共有**

　本実験では, 各々にPCが割り当てられ, それにボードを接続してデバッグやテストを行う必要があったため, 最新の実行環境を班員同士で常に共有しておかなければ, 誰かが解決した問題で他の人が詰まってしまうような事態が発生する危険があった. 我々の班では, 全員がGit hubのようなプラットフォームを使いこなせる訳ではなかったため, Onedriveの共有フォルダを通して, 修正・改善したコードの共有を行い, スムーズに実装が進むように注意した.

**6. プログラムに発生した問題とその原因，解決方法**

**addq()、removeq() の有効活用**

　我々の班では, 作業を「マルチタスク機能」「タイマ」「セマフォ」の3つに分け, それぞれの担当チームが開発を進めた. その結果, 基本的に同じ担当部門内のメンバーのみで意見を交わしながらコードを作成することになり, 他の担当部門との連携が少なくなってしまった. そのため, 各チームが作成したコードを統合してテストを行った際, 異なるチームが定義した変数や関数のやり取りがうまくいかず, プログラムが正しく動作しなかった.

　特に, タスクの実行・休眠を制御する sleep()・wakeup() 関数が機能せず, コードを調査したところ, タスクの状態を適切に更新する addq()・removeq() を使用せずに, 複数の変数を用いて手動でタスクの登録や new\_task の更新を行っていたことが原因であると判明した. この実装では, タスク管理の整合性が取れず, エラーが発生していた.

　この問題を解決するために, 自分を含めたマルチタスク機能を担当するメンバーを交えて話し合い, コードを大幅に整理・削減し, addq()・removeq() を適切に活用する形に修正した. その結果, タスク管理の効率が向上し, プログラムの動作が安定し, テストも正常に動作するようになった.

　以下に, 修正を行う前後での sleep()・wakeup() のコードを示す.

**・修正前のsleep()・wakeup()**

int sleep(int a){

    /\* a := セマフォID\*/

    int b = semaphore[a].task\_list; /\* b := セマフォ[a]の先頭タスクID\*/

    if(b == NULLTASKID){

        semaphore[a].task\_list = curr\_task; /\* セマフォ[a]が空のとき現在のタスクをセマフォの先頭へ \*/

        task\_tab[curr\_task].next = NULLTASKID;

    }else{

        int c = task\_tab[b].next;

        while(c != NULLTASKID){ /\* セマフォキューの末尾を辿り、たどり着いた時、 \*/

            b = c; /\* b= 末尾のタスクのID \*/

            c = task\_tab[b].next; /\* c= NULLTASKID \*/

        }

        task\_tab[b].next = curr\_task; /\* セマフォキューの末尾に現在のタスクを登録 \*/

        task\_tab[curr\_task].next = NULLTASKID;

    }

    sched();

    swtch();

}

int wakeup(int a){

    /\* a := セマフォID \*/

    int b = semaphore[a].task\_list; /\* b:= セマフォ[a]の先頭タスクID \*/

    if(b != NULLTASKID){

        int c = task\_tab[b].next; /\* c := セマフォ[a]の二番目のタスクID \*/

        task\_tab[b].next = NULLTASKID;

        semaphore[a].task\_list = c; /\* セマフォ[a]の先頭のタスクIDをcに設定 \*/

        if(ready == NULLTASKID){

            ready = b; /\* readyキューが空のとき先頭のタスクIDをbにする \*/

        }else{

            int d = task\_tab[ready].next;

            while(d != NULLTASKID){ /\* readyキューの末尾をたどり、たどり着いた時、 \*/

                c = d; /\* c= 末尾のタスクのID \*/

                d = task\_tab[c].next; /\* d= NULLTASKID \*/

            }

            task\_tab[c].next = b; /\* readyキューの末尾にbを登録 \*/

        }

    }

}

**・修正後のsleep()・wakeup()**

int sleep(int a)

{ /\* a := セマフォID\*/

  addq(&(semaphore[a].task\_list), curr\_task);

  sched();

  swtch();

}

int wakeup(int a)

{

  TASK\_ID\_TYPE task\_id = removeq(&(semaphore[a].task\_list));

  addq(&ready, task\_id);

}

**7. 考察**

　テーマ2を通して, TCBやセマフォ, タイマ割り込みを用いてマルチタスクカーネルを作成した. 今回は, タスクの切り替えを一定の周期でのタイマ割り込みをもとに行い, タスクの順番はFIFOで決定されることから, Round-Robinスケジューリング方式をとっていると言える.

　その中でも, セマフォは重要な役割を果たしていたといえる. 特に, 共有リソースに対する排他制御を実現するP操作とV操作を活用することで, リソース競合を防ぎつつ, タスク間の同期を確実に行うことができた。一方で, セマフォを適切に管理しない場合にはデッドロックのリスクがあることも確認できた.

また, TCBを用いたタスク管理では, タスクの状態遷移を明確に管理でき, タスク切換えの際には, 現在のタスクの情報を保存し, 次のタスクの情報を復元するプロセスを円滑に実現できた. これにより, タスク切り替えを効率よく実現しており, 今回のマルチタスキングの根幹を支えていると実感する結果となった.

さらに, Round-Robinスケジューリング方式では, タイマの割り込み周期がシステムに大きな影響を与えていると考えられる. 具体的には, 間隔を短く設定すると切り替えが頻発しオーバーヘッドが増加する一方, 長く設定しすぎるとタスクの応答性が低下すると考えられる. そのため, これ以降の実験では, このトレードオフを考慮し, システムに最適な時間を選定することが重要であると考えられる.

このように, 今回の実験を通じて, セマフォやTCBを活用したRound-Robinスケジューリング方式のマルチタスキングは, 効率性と公平性を両立する優れた手法であるといえる. またそれと同時に, リソース管理やタスク設計において多くの工夫が求められることも事実であるため, より複雑なマルチタスクシステムの設計や応用にむけてこの学びを活かしていきたい.

**テーマ3：応用**

**1. テーマ3についての説明**

テーマ3では, テーマ2で作成したマルチタスクカーネルを拡張し, 2つのRS232Cポートを用いたI/O処理を並列実行可能な環境を構築する. これにより, 2つのポートを用いたデータ通信やマルチユーザー環境のシミュレーションが可能となる.

具体的には以下を実装・拡張する.

1. I/O関数の改造

inbyte()やoutbyte()をそれぞれのポートに対応させ, 独立したI/O処理が可能になるよう修正する. UART1をRS232Cポート0, UART2をRS232Cポート1に割り当てる.

1. ファイルディスクリプタによるポート指定

read()や write()の第1引数であるファイルディスクリプタ（fd）を拡張し, どのポートでI/Oを行うかを指定可能にする.

1. fdopen()の実装

各ポートに対応するストリームを適切に割り当てることで, 複数のポートを扱うI/O操作を効率化する.

**2. テーマ3の実験全体における位置づけ**

テーマ3の目的は以下の2点に集約される.

1. マルチタスクの機能拡張

複数のポートを同時に制御することで, より実践的なマルチタスク環境を構築する.

1. 排他制御の応用

複数のタスクが同時にI/Oリソースを使用する場合, セマフォを用いて適切に排他制御を行い, データ競合を防ぐ.

また, このテーマの後には2ポートを活用したアプリケーションの開発も行う. 例えば, 2人用の対戦ゲームや双方向の通信システムなどが考えられる. これにより, マルチタスク処理の実装だけでなく, 実際のシステム開発における応用力も養う.

**3. プログラムのリスト（新規作成部）**

**・inchrw.s**

.global inbyte

.include "equdefs.inc"

.text

.even

inbyte:

    link.w  %a6, #-4

    movem.l %d1-%d3/%a0, -(%sp)

inbyte\_loop:

    move.l  #SYSCALL\_NUM\_GETSTRING, %d0

    move.l  %sp, %a0

    adda.l  #28, %a0

    move.l  (%a0),%d1

    move.l  %a6, %d2

    sub.l   #1,  %d2

    move.l  #1, %d3

    trap    #0

    cmpi.l  #0, %d0

    beq     inbyte\_loop

    move.b  'A', LED0

    move.b  '0', LED1

    move.b  -1(%a6), %d0

    movem.l (%sp)+, %d1-%d3/%a0

    unlk    %a6

    rts

**・outchr.s**

.global outbyte

.include "equdefs.inc"

.text

.even

outbyte:

    movem.l %d1-%d3/%a0, -(%sp)

outbyte\_loop:

    move.l  #SYSCALL\_NUM\_PUTSTRING, %d0

    movea.l %sp, %a0

    adda.l  #20, %a0

    move.l (%a0), %d1

    move.l  %sp, %d2

    addi.l  #27, %d2

    move.l  #1, %d3

    trap    #0

    cmpi.l   #0, %d0

    beq outbyte\_loop

    movem.l  (%sp)+, %d1-%d3/%a0

    rts

**・csys68k.c**

#include <stdio.h>

#include "mtk\_c.h" // マルチタスクカーネル用ヘッダー

#include <stdarg.h>

#include <fcntl.h>

#include <errno.h>

extern void outbyte(int ch, unsigned char c);

extern char inbyte(int ch);

int read(int fd, char \*buf, int nbytes)

{

  char c;

  int i;

  int ch;

  if (fd != 0 && fd != 3 && fd != 4)

  {

    return EBADF;

  }

  else if (fd == 0 || fd == 3)

  {

    ch = 0;

  }

  else

  {

    ch = 1;

  }

  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;

  int ch;

  if (fd != 1 && fd != 2 && fd != 3 && fd != 4)

  {

    return EBADF;

  }

  else if (fd == 1 || fd == 2 || fd == 3)

  {

    ch = 0;

  }

  else

  {

    ch = 1;

  }

  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;

  }

}

**・test3.c内mapping関数**

//---------------------mapping関数-------------------

void mapping()

{

    com0in  = fdopen(3, "r");

    com0out = fdopen(3, "w");

    com1in  = fdopen(4, "r");

    com1out = fdopen(4, "w");

}

**4. プログラムの説明**

**・inchrw.s**

テーマ3では, 複数ポートからの入出力を受け付けるために, read()とwrite()を呼び出す際に, 第1引数であるfdを用いて, ポートの指定を行う. そのため, その内部で呼び出されるinbyteに関してもfdに対応した形に変更する必要があり, fdをもとにシステムコールtrap #0で指定されたポートを利用する形に変更を行った.

具体的には, %a0にスタックポインタの値を保存し, そこに退避させたスタックや戻り先のPCの分の領域を考慮したオフセットを加算し, 第1引数が格納されている場所から確実に値をとりだし, %d1格納するという命令を記述している.

**・outchr.s**

　outbyteも上記の理由から, 同様にfdに対応した形へと変更を行った.

具体的にはここでも, %a0にスタックポインタの値を保存し, そこに退避させたスタックや戻り先のPCの分の領域を考慮したオフセットを加算し, 第1引数fdが格納されている場所から確実に値をとりだし, %d1格納するという命令を記述している. そして, このままシステムコールにチャンネルとして渡されるという流れになっている.

また, 新たに%a0を退避させているために, 実際のデータの格納領域を示すためのオフセットも変更されている.

**・csys68k.c**

**readの変更点**

　ここでは資料に基づき, 読み書き可能性を考慮したデバイスマッピング判定をもとにポートを選択できるように, 変更を行っている.

　具体的には, fdが0または3の場合はシリアルポート0を, fdが4の場合はシリアルポート1を使用し, その他のfdに対してはEBADF（無効なファイルディスクリプタエラー）を返すように実装している. また, ポート選択のために新たにchという変数を定義し, inbyteに引数として渡す値として利用している.

**writeの変更点**

　writeも同様に, 資料に基づいたデバイスマッピング判定を導入し, fdが1, 2, 3の場合はシリアルポート0を, fdが4の場合はシリアルポート1を使用し, その他のfdではEBADFを返すようにしている. またここでも, ポート選択のために新たにchという変数を定義し, outbyteに引数として渡す値として利用している.

**fcntlの新規実装**

　fcntl()は関数引数の個数が可変であるため, 最初にstdarg.hをインクルードしたうえで, cmdがF\_GETFLの場合に, O\_RDWR(入出力可能)を返し, その他には0を返すように実装をしている. これは, 本来ならばfdに対するopen()時にのフラグをカーネルに問い合わせる必要があるが, 実験カーネルでは管理していないためである.

**・test3.c内mapping関数**

　この関数では, 入出力ストリームをファイルディスクリプタに割り当てるためのCライブラリ関数であるfdopen()を用いて, FILE構造体へのポインタとしての大域変数を準備するための関数である. fdopen()は, FILE \*fdopen(int fd, const char \*mode); と定義されており, fdはファイルディスクリプタ, modeはfdに対する入出力モードを示しており, 読み込みの場合は”r”, 書き込みの場合は”w”を指定する. ここでは, RS232Cに対応するものとして,

com0in： UART1からの読み込み

com0out： UART1への書き込み

com1in： UART2からの読み込み

com1out： UART2への書き込み

と準備している.

**5. プログラムにおいて注意したこと**

　プログラムで注意した点は, inbyte・outbyteを変更する際に, read()・write()の第1引数であるfdの値を取って来る際のスタックの扱いである. 資料にあるように引数は, スタックに技から左へと積まれ, さらにchar(8ビット), short(16ビット)のデータは符号拡張命令 (EXT)を用いて32ビットデータに符号拡張したのちスタックに積まれるといった処理が施されるためそれを考慮したオフセット計算が必要であった. そのため, 班員で図を書きながら確認をし合って値を決めて, 確実に値を取ってこられるようにした.

　また, mapping()やfcntl()をどこに記述し, extern宣言を行えばいいのかも注意して確認し, プログラムが問題なく動作するように記述した.

**6. プログラムに発生した問題とその原因，解決方法**

　上記に示した通り, 引数の処理のされ方には様々なルールがあったが, はじめはこれを理解していなかったため, 2ポートを使った入出力が全くできていなかった. そして改めて資料を読み込んでいくとこのルールの存在に気づき, 修正することができた. 修正の途中でも様々な値を試して, そのたびにLEDを使ったデバッグでどこまで実行できているか, またどんな引数が渡されているかのテストをしながら確認することで, 正確なオフセットを求めることができた.

**7. 考察**

　テーマ3では, inbyte, outbyte, read(), write()などの改良, mapping(), fcntl()の作成を通して, 2ポートで入出力を行う事ができるようになった. これにより複数の画面を使用してタスクを実行していくことができるようになり, テーマ２で実現したマルチタスク性を十分に発揮するプログラムを動かすことが可能になったと言える.

**個人課題**

1. **個人課題の概要**

個人課題では, 簡易的なトランプのHigh&Lowゲームの実装を行った. 実際に, 実装した内容は以下の通りである.

* + 1~13までのカードが, シャッフルされて山札として設定される.
  + はじめの1枚をめくり数字を確認する
  + 2人のプレイヤーは山札の次の数字が先程のカードより大きいか小さいかを予想し, それぞれHighであれば1, Lowであれば0を入力する
  + 山札をめくり, 予想があたったものにポイントが加算される.
  + 指定したラウンドまで繰り返し, ポイントが多いほうが勝者となる.

　この課題内での, マルチタスク性を活かした部分は, プレイヤーが回答をする部分で, まず両者の入力を同時に受け付けたうえで, そろってラウンドを重ねていくために, 両者ともに解凍が終わるまで, 結果判定・及び次のラウンドに進まないようになっている. また, 乱数を生成するタスクを独立に用意し, はじめの手札をシャッフルする部分で活かしている.

**2. プログラムのリスト**

**・test3.h**

#ifndef TEST3\_H

#define TEST3\_H

// カード枚数 (1〜13)

#define TOTAL\_CARDS 13

// 「数字カード」だけを持つシンプルな構造体

typedef struct

{

    int number; // 1〜13

} CARD;

// グローバルな山札の配列 (13 枚)

extern CARD cards[TOTAL\_CARDS];

// カードの初期化 (1〜13 を設定)

void init\_card();

// カードのシャッフル

void shuffle\_card();

#endif // TEST3\_H

**・test3.c**

#include <stdio.h>

#include "mtk\_c.h"

#include "test3.h"

CARD cards[TOTAL\_CARDS];

// -------------------- グローバル変数 --------------------

int round\_count;

int top\_index;

int score1;

int score2;

int guess1;

int guess2;

int is\_finish;

int winner;

// -------------------- 乱数生成 --------------------

volatile unsigned int rand\_counter = 2;

void random\_generator()

{

    while (1)

    {

        rand\_counter = (rand\_counter \* 1103515245 + 12345) & 0x7FFFFFFF;

    }

}

unsigned int get\_random()

{

    return rand\_counter;

}

// -------------------- カード初期化 --------------------

void init\_card()

{

    for (int i = 0; i < TOTAL\_CARDS; i++)

    {

        cards[i].number = i + 1;

    }

}

void shuffle\_card()

{

    for (int i = TOTAL\_CARDS - 1; i > 0; i--)

    {

        unsigned int r = get\_random() % (i + 1);

        int temp = cards[i].number;

        cards[i].number = cards[r].number;

        cards[r].number = temp;

    }

}

// -------------------- ラウンド表示 --------------------

void first\_display\_round()

{

        fprintf(com0out, "----------------------\n");

        fprintf(com0out, "Round %d\n", round\_count);

        fprintf(com0out, "Current Number: %d\n", cards[top\_index].number);

        fprintf(com0out, "Player0 Score=%d, Player1 Score=%d\n", score1, score2);

        fprintf(com1out, "----------------------\n");

        fprintf(com1out, "Round %d\n", round\_count);

        fprintf(com1out, "Current Number: %d\n", cards[top\_index].number);

        fprintf(com1out, "Player0 Score=%d, Player1 Score=%d\n", score1, score2);

}

// -------------------- Player 0 --------------------

void player0(){

    P(0);

    while (1)

    {

        fprintf(com0out, "Player 0: 次のカードは High(1) か Low(0) か? (Score=%d): ", score1);

        int buf1;

        fscanf(com0in, "%d", &buf1);

    if (buf1 == 1){ //High

        guess1 = 1;}

    else if (buf1 == 0){    //Low

        guess1= -1;}

    else{

        guess1 = 1;     //default High

        }

        V(0);

        V(2);

        P(3);

    }

}

// -------------------- Player 1 --------------------

void player1(){

    P(1);

    while (1)

    {

        fprintf(com1out, "Player 1: 次のカードは High(1) か Low(0) か? (Score=%d): ", score2);

        int buf2;

        fscanf(com1in, "%d", &buf2);

    if (buf2 == 1){ //High

        guess2 = 1;}

    else if (buf2 == 0){    //Low

        guess2= -1;}

    else{

        guess2 = 1;     //default High

        }

        V(1);

        V(2);

        P(3);

    }

}

// -------------------- 結果判定 --------------------

void judgment()

{

    while (1)

    {

        P(0);

        P(1);

        P(2);

        P(2);

        int next\_number = cards[++top\_index].number;

        if (next\_number > cards[top\_index - 1].number)

        {

            if (guess1== 1)

                score1++;

            if (guess2 == 1)

                score2++;

        }

        else if (next\_number < cards[top\_index - 1].number)

        {

            if (guess1 == -1)

                score1++;

            if (guess2 == -1)

                score2++;

        }

        round\_count++;

        fprintf(com0out, "----------------------\n");

        fprintf(com0out, "Round %d\n", round\_count);

        fprintf(com0out, "Current Number: %d\n", cards[top\_index].number);

        fprintf(com0out, "Player0 Score=%d, Player1 Score=%d\n", score1, score2);

        fprintf(com1out, "----------------------\n");

        fprintf(com1out, "Round %d\n", round\_count);

        fprintf(com1out, "Current Number: %d\n", cards[top\_index].number);

        fprintf(com1out, "Player0 Score=%d, Player1 Score=%d\n", score1, score2);

        if (round\_count >= 5 || top\_index >= TOTAL\_CARDS)

        {

            is\_finish = 1;

            winner = (score1 > score2)   ? 0

                     : (score1 < score2) ? 1

                                             : -1;

            fprintf(com0out, "\*\*\* Game Over! \*\*\*\nWinner: %s\n",

                    (winner == -1)  ? "Draw"

                    : (winner == 0) ? "Player 0"

                                    : "Player 1");

            fprintf(com1out, "\*\*\* Game Over! \*\*\*\nWinner: %s\n",

                    (winner == -1)  ? "Draw"

                    : (winner == 0) ? "Player 0"

                                    : "Player 1");

            while(1){

            }

        }

        V(3);

        V(3);

        V(0);

        V(1);

    }

}

//---------------------mapping関数-------------------

void mapping()

{

    com0in  = fdopen(3, "r");

    com0out = fdopen(3, "w");

    com1in  = fdopen(4, "r");

    com1out = fdopen(4, "w");

}

// -------------------- main関数 --------------------

int main()

{

    mapping();

    init\_kernel();

    init\_card();

    shuffle\_card();

    first\_display\_round();

    semaphore[0].count = 1;

    semaphore[1].count = 1;

    semaphore[2].count = 0;

    semaphore[3].count = 0;

    set\_task((char\*)random\_generator);

    set\_task((char\*)player0);

    set\_task((char\*)player1);

    set\_task((char\*)judgment);

    begin\_sch();

    return 0;

}

**3. プログラムの説明**

**・グローバル変数**

ゲームの状態を管理

int round\_count：ラウンド数

int top\_index：現在のカードの位置

プレイヤーの管理

int score1, score2：プレイヤーのスコア

int guess1, guess2：プレイヤーの予想(High:1/Low:0)を管理

フラグと勝者

int is\_finish：終了フラグ

int winner：勝者の情報

**・random\_generator()**

　この関数は, 擬似乱数を生成するための関数である. 無限ループ内で計算を繰り返し, グローバル変数rand\_counterに擬似乱数を生成して格納している. 乱数を生成する仕組みについては, 線形合同法（Linear Congruential Generator, LCG）と呼ばれる乱数生成アルゴリズムを用いており, 無限ループ内で乱数を生成し続けるため、常に最新の乱数を取得できる仕組みになっている. しかしこのプログラム内では, 最初のシャッフルの際に1回呼び出すだけとなっている.

**・get\_random()**

この関数は, グローバル変数rand\_counterに格納されている最新の擬似乱数を取得して返す関数である. 別タスクとして動作するrandom\_generator関数によって生成された乱数を利用するためのインターフェースを提供している.

**・init\_card()**

　この関数は, カードゲームに使用するカードのデッキを初期化するための関数である. forループを使用して, cards配列内の各要素に1から順に番号を順番に割り当る用になっている.

**・shuffle\_card()**

この関数は, カードデッキ（cards配列）をランダムに並び替えるためのシャッフル機能を提供するものである. init\_card関数で順番に初期化されたカードを, 乱数を利用してランダムな順序に並び替えることで, ゲーム用のデッキを作成します. 詳細な動作は以下の通りである.

1. デッキの末尾からシャッフル

逆順ループ（for文）を使用して, 配列の末尾から順番にシャッフルを行う. 現在のインデックスiより小さい範囲でランダムな位置rを選択し, i番目のカードとr番目のカードを交換する.

2. ランダムなインデックスの生成

関数get\_random()を使用して乱数を取得し, それをi+1で割った余り（% (i + 1)）を計算して, 0からiまでのランダムなインデックスrを生成する.

3. カードの入れ替え

cards[i].numberとcards[r].numberの値を交換することで, シャッフルを実現する.

　このシャッフルの方法は, Fisher-Yates法に基づいており, 配列のサイズ（TOTAL\_CARDS）に応じて動的に動作するため, カードの枚数が変更されても問題なく対応可能な仕組みになっている.

**・first\_display\_round()**

　この関数は, ゲームの各ラウンド開始時に現在の状況をプレイヤーに表示するための機能を提供している. ラウンド数, 現在のカードの番号, 各プレイヤーのスコアをそれぞれのシリアルポート（com0outとcom1out）を通じて出力する.

**・player0(), player1()**

　これらの関数は, それぞれPlayer 0とPlayer 1の入力処理を行うタスクとして設計されている. プレイヤーが「次のカードの値が前のカードより高い（High: 1）か、低い（Low: 0）か」を予想し、その結果を変数guess1またはguess2に記録する. また, セマフォを使用してタスク間の同期を行い, ゲームの進行を制御する.

**・mapping()**

　省略

**・main()**

　この関数は, カードゲームの初期設定とタスクの登録を行い, タスクスケジューラを起動する役割を担っている. 全体的な初期化処理と, ゲームを開始するための準備を行う. 経じめに, セマフォ2のカウントを0にすることで, 判定タスクが待機状態になるようにしている.

**・タスク間の流れとセマフォ操作**

以下に, 1ラウンドでのタスク間の流れとセマフォ操作の順序を示す.

1. Player 0の入力

* P(0)を実行してセマフォの待機を解除し, Player 0が入力を開始.
* 入力完了後：
  + V(0)：Player 0の入力完了を通知.
  + V(2)：判定タスクを実行可能にする.

2. Player 1の入力

* P(1)を実行してセマフォの待機を解除し, Player 1が入力を開始.
* 入力完了後：
  + V(1)：Player 1の入力完了を通知.
  + V(2)：判定タスクを実行可能にする.

3. 判定処理

* 判定タスクはP(0), P(1)、P(2)（2回）を実行して, 両プレイヤーの入力が完了していることを確認.
* 次のカードを判定し, スコアを更新.
* ゲーム進行状況を出力.
* 次のラウンドの準備として, V(3)を2回呼び出して, 両プレイヤーが次のラウンドで入力を再開可能にする. また, V(0)とV(1)を呼び出して, 次のラウンドで各プレイヤータスクが入力を開始可能にする.

**4. 個人課題まとめ**

個人課題を通じて, 期待どおりに動作しない場面が多く, 原因を探ることに多くの時間を費やした. そのため, ゲームのルールを変更したり, 機能を削減することになり, マルチタスク性を最大限に発揮した制作物にはならなかった. しかし, セマフォを用いたマルチタスクの仕組みやその重要性について理解を深めることができた点は, 大きな収穫だと感じている.

今後, さらに機能を追加したり, より本格的なトランプゲームに近づけた実装を行う際には, 優先度ベースのタスクスケジューリングや非同期I/Oの導入が非常に有用だと考えている. これらを実現することで, 効率的で拡張性の高いマルチタスクシステムが構築できると考えられ, 今後の課題として引き続き取り組みたい.