学生実験 (C 課程・後期)

ソフトウェア実験

マルチタスクカーネルの作成

目 次

実験の目的

本実験は，オペレーティングシステムの基本的な機能について，動作を理解することを目的とする．

実験の内容は３つに分類される．１つは，アセンブラ言語と高級言語で記述されたプログラム間での制御やデータの授受の方法を理解することである．具体的には，高級言語として，Ｃ言語を扱う．Ｃ言語のライブラリを用い，そのライブラリが利用できる環境をオペレーティングシステム上に構築する．これにより，Ｃ言語のプログラムからアセンブラ言語で記述されたプログラムを呼び出す方法として，変数名の設定法，レジスタやスタックなどの利用法，コンパイルして１つのロードモジュールとする方法，などを修得する．２つめは，オペレーティングシステムの基本制御機能の１つとして，プロセスの並行制御方式を理解することである．具体的には，セマフォの機能をオペレーティングシステムとして実現する．セマフォのシステムコールを提供し，複数のプロセスからの利用が可能な環境を実現する．オペレーティングシステムとして，セマフォによるプロセス切替の機能も実現する．これにより，並行プロセスを制御する概念を修得する．最後に，排他制御による処理の同期方法を理解することである．具体的には，セマフォ機能を利用して，排他制御により処理の同期を図る応用プログラムを作成する．これにより，排他制御の基本概念を修得する．

試問とレポートについて

実験期間中に 2 回の試問とレポート提出が必要です．試問と同時にレポートを提出してください．1 回目の試問日は別途連絡します．2 回目の試問は実験の最終日です．2 回ともレポートは一人ずつ作成してください．レポート作成時の注意点は以下の通りです．

テーマ 1

1 回目のレポート (テーマ 1) では，次の点に注意してください．

- レポートには**最低限**以下の項目を記載．

  テーマ 1 についての説明 (教本のまる写しは不可)
  テーマ 1 の実験全体に占める位置付けの説明
  プログラムの説明
  プログラムのリスト
  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

- テーマ 2 の準備として，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，思考実験を行え．

- 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)

思考実験では，不自然ではないが実験遂行上都合の良いプログラム場所・タイミングでタスク切替えが行われるものとする。ユーザタスクプログラムの説明とリストを記載し，時分割のタスク切替えをきっかけとした実行タスク，ready キュー，セマフォキューの時間を追った状態変化の一覧を示しながら動作説明をすることによって，上の条件を満たしていることを示せ.

(テーマ 2 で作成するマルチタスクカーネルの動作確認に用いることを前提に作成すること. ただし、実際にターゲット上で動作を確認する必要はない．思考実験で採用した都合の良いタイミングによるタスク切替えの具現化が難しいからである．)

テーマ 2, 3

2 回目のレポート (テーマ 2, 3) では，次の点に注意してください.

- ユーザタスクの部分は 1 人に 1 つずつ別な物を作ること.
- レポートには**各テーマ毎に最低限**以下の項目を記載.

  当該テーマについての各説明 (教本のまる写しは不可)
  当該テーマの実験全体に占める位置付けの説明
  プログラムの説明 、プログラムのリスト
  　テーマ 2 (カーネル，user タスク)：担当部を明記し、担当部はより詳しく
  　テーマ 3 (カーネル変更部, user タスク)
  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

- プログラムファイルの提出.

  テーマ 3 の作業フォルダ内にモニタのオブジェクトファイルのリファレンスへの差替え状況を記した README.txt を作成した後、make clean を実行した後に作業フォルダ毎 zip 圧縮を行い、出来上がった zip ファイルを Moodle のファイル提出場所に提出する。zip ファイル名は T3-学籍番号-氏名.zip とする事.

実験最終日に，本実験についての簡単なアンケートを実施します．ご協力をお願いします.

第1章　テーマ１：C言語ライブラリの移植

1.1　はじめに

テーマ１では，実験で用いるターゲットの上でC言語プログラムが実行できるように，C言語処理系 m68k-elf-gcc をカスタマイズする．具体的には，m68k-elf-gcc に付随して提供されているライブラリのうち `read()`，`write()`，`printf()`，`scanf()` をターゲット上で使用できるように移植する．

このためには，以下のことがらについての理解を深める必要がある．

1. C言語プログラムで書かれたプログラムとアセンブリ言語で書かれたプログラムから１つの実行可能コードを作成する方法
2. C言語プログラムの起動方法
3. C言語の入出力ライブラリの構成法

1.2　基本的事項

1.2.1　プログラムの分割コンパイル・アセンブル

プログラム作成を行なう際には，１つの巨大なソースプログラムを作成して一気にコンパイル・アセンブルを行なう方法よりも，プログラム全体を目的毎や機能毎のより小さなプログラムファイル (モジュール) に分割し，個々のモジュール毎にプログラム作成を行なった後にモジュール毎の分割コンパイル・アセンブルを行なう方法を勧める．

この様な分割プログラミングの利点として，以下の点が挙げられる．

1. モジュール毎に分担する事によって，多人数によるソースプログラム作成の際の作業分担が容易．
2. プログラム中に不具合が発生した場合，モジュール毎に点検する事によって問題点の絞り込みが容易．
3. ソースプログラムに修正を加えた後の再コンパイル・アセンブルは，修正箇所を含んだモジュールのソースプログラムに対してのみ行なうだけで済み，プログラム全体をコンパイル・アセンブルし直す必要は無い．

分割プログラミングの短所としては，小さなプログラムモジュール片がいくつも作成され，手動でコンパイル・アセンブルするには手間がかかるという点が挙げられるが，この作業を支援するユーティリティーとして make コマンドがある．make コマンドについては他書を参照されたい．

以下に，m68k-elf-gcc による C 言語の分割コンパイルや m68k-elf-as によるアセンブリ言語の分割アセンブルを行なう際に必要となる事柄について述べる．

File A

```
char CommonArray[10];

int func1(int a)
{
    a < b
      ...
}

char func2(char b)
{
      ...
}
```

File B

```
extern int  func1(int  a);
extern char func2(char b);
extern char CommonArray[10];

main()
{
    int  tmp;
    char c;
      ...
    tmp = func1(10);
    c = func2('A');
      ...
}
```

図 1.1: C 言語プログラムのファイル分割

C 言語プログラムの分割コンパイル

C 言語のプログラムは，1 つの `main()` 関数 (Pascal 言語での本体に相当する) とその他の関数 (Pascal 言語での関数や手続きに相当する) で記述するのが普通である．Pascal では，呼び出される関数・手続きは呼び出す関数・手続きの前に記述する．C 言語では関数の記述順序については規定がない．しかし，C 言語において関数を呼び出す際には，それ以前に呼び出す関数を記述しておく方がよい．関数の呼び出しよりもソースファイルの後方でどうしても記述したい場合には，ソースファイルの始めで関数定義だけを行なう．

C コンパイラは 1 つのファイルを 1 つのコンパイル単位とする．1 つの大きなプログラムをいくつかのモジュールに分割してコンパイルする時には，プログラムをいくつかのファイルに分けて作成し，それをすべてコンパイルする．例を図 1.1 に示す．File A 中に int 型の値を返す関数 `func1()` と char 型の値を返す関数 `func2()` を記述し，File B 中の関数 `main()` でそれらを使用している．C コンパイラはこれらのファイルを別々にコンパイルするので，File B をコンパイルする際には関数 `func1()` と `func2()` が存在することを知らない．別のファイルでこれらの関数が記述されていることを知らせるために，普通ソースファイルの先頭に関数定義を書く．それが File B の ‘`extern`’ で始まる行である．この 2 行で関数 `func1()` と関数 `func2()` の存在をコンパイラに知らせる．また，関数だけでなく複数のコンパイル単位で使用する変数を使用する場合にも，配列 `CommonArray` の宣言のように ‘`extern`’ 宣言を使用する．

ソースファイルの中に別のファイルの内容を取り込む方法として，インクルードと呼ばれる方法がある．これは，取り込む側のソースファイルに以下の記述をすることで別のファイルを取り込む方法である．

例： `#include  "ファイル名"`

この指定は，“ファイル名” で表されるファイルの内容全体がプログラム中のその場所に単純に

def.h

```
#define  NEWLINE   '\n'
#define  TAB       '\t'
#define  NULL      '\0'
#define  FALSE     0
#define  PI        3.1416
```

main.c

```
#include  "def.h"

main()
{
      ...
}
```

図 1.2: インクルードによるファイルの取り込み

挿入されることを意味している．この方法では，プログラムを分割して別のファイルに保存しているが，コンパイルの時には 1 つのファイルのようにして扱われる．そのため，複数のプログラム単位で使用するマクロなどを定義したファイルを作成し，そのファイルを各プログラム単位でインクルードする等のようにして使用する．図 1.2 では，ファイル “def.h” 中で定義したマクロ (NEWLINE，TAB など) をファイル main.c 中の関数で使用できる．def.h のような複数のプログラム単位で使用するマクロなどを定義したファイルをヘッダファイルと読んでいる．

上に示したインクルードでは，ファイル名をダブルクォーテーションで括ったものを示したが，`#include <ファイル名>`のように<と> でファイル名を括ることもある．これら 2 種類の記述の違いはインクルードされるファイルをどのディレクトリで探すかの違いだけである．ダブルクォーテーションでファイル名を括った場合には，カレントディレクトリでその名前のファイルを探した後にシステムで定義されているディレクトリで探す．<と>でファイル名を括った場合には，カレントディレクトリでは探さずに直接システムで定義されたディレクトリでそのファイルを探す．stdio.h のようにシステムで定義されているヘッダファイルをインクルードする場合には，普通<と>で括る方のインクルードを使う．

アセンブリ言語プログラムの分割アセンブル

アセンブリ言語プログラム例 (TEST0) を図 1.3 に示す．モジュール TEST0 内のサブルーチン mst は変数 locala，localb を使用しないものとして，サブルーチン foo，bar，変数 locala，localb の組 (モジュール TEST1) とサブルーチン mst，変数 glob の組 (モジュール TEST2) に分割する事を考える．プログラムを分割してみると，モジュール TEST2 内のサブルーチン mst から TEST1 内のサブルーチン foo，bar を呼び出したり，モジュール TEST1 内のサブルーチンから TEST2 内の変数 glob をアクセスする必要が生じてくる．

この様な 外部モジュール内のサブルーチンや変数の使用を可能にする為に，.global/.extern ディレクティブを用いる．サブルーチン等のラベル名や変数名 (シンボル) を更に .global 宣言すると，宣言されたシンボルは外部定義の属性を与えられる事によって外部モジュールから利用可能となる．また，外部モジュールで .global 宣言されているシンボルを自モジュール内で .extern 宣言する事によって，外部モジュール内にあるサブルーチンや変数を自モジュール内で利用する事が出来る．

図 1.3 のアセンブリ言語プログラムを分割した例 (TEST1 と TEST2) を図 1.4 に示す．モジュール TEST1 からはモジュール TEST2 内の変数 glob を, TEST2 からは TEST1 内のサブルーチン名 foo，bar を必要としているので，TEST1 ではシンボル foo，bar，TEST2 ではシンボル glob

```
foo:        ...                 mst:        BRA.S   mstbr
        MOVE.L  locala,%D0      mstlp:  ...
        ADD.L   %D0,glob                JSR     foo
        ...                             JSR     bar
        RTS                             ...
                                mstbr:  TST.L   glob
bar:        ...                         BGT     mstlp
        MOVE.L  localb,%D0              RTS
        SUB.L   %D0,glob
        ...                     .section .data
        RTS                     glob:       .dc.l   16
                                locala:  .dc.l   1
                                localb:  .dc.l   2
```

図 1.3: アセンブリ言語プログラム例 (分割前，TEST0)

を.global 宣言すると共に，TEST1 では glob，TEST2 では foo，bar を.extern 宣言している．

.global/.extern ディレクティブはアセンブリ言語プログラム中の何処からでも使用する事が出来る． 詳細は m68k-elf-as のマニュアルを参照のこと．

1.2.2 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

m68k-elf-gcc/m68k-elf-as/m68k-elf-ld の関連

C 言語処理系およびアセンブリ言語処理系にはさまざまなものがあるが，本書では今回の実験で使用する処理系を取り上げて解説する．今回の実験では，以下の 3 つのシステムを使用する．

1. **m68k-elf-gcc コンパイラ**
   C で書かれたソースプログラムを m68k-elf-as で処理可能な 68000 アセンブリ言語ソースプログラムに翻訳する．
2. **m68k-elf-as アセンブラ**
   68000 アセンブリ言語で書かれたソースプログラムを m68k-elf-ld で処理可能な 68000 オブジェクトコードに変換する．
3. **m68k-elf-ld ローダ**
   68000 オブジェクトコードを実行可能コードに変換する．複数のオブジェクトコードファイルを 1 つの実行可能コードにリンクすることもできる．

したがって，これらのシステムを用いると，例えば以下のようなプログラム開発を行うことができる．

1. アセンブリ言語で書いた単一のプログラムを m68k-elf-as でオブジェクトコードに変換し，m68k-elf-ld で実行可能コードに変換する．
2. アセンブリ言語で書いた複数のプログラムを m68k-elf-as でそれぞれオブジェクトコードに変換し，m68k-elf-ld で単一の実行可能コードにまとめる．

```
        .global foo
        .global bar
        .extern glob
foo:        ...
        MOVE.L  locala,%D0
        ADD.L   %D0,glob
        ...
        RTS
bar:        ...
        MOVE.L  localb,%D0
        SUB.L   %D0,glob
        ...
        RTS

.section .data
locala:  .dc.l   1
localb:  .dc.l   2
```

```
        .extern foo
        .extern bar

mst:        BRA.S   mstbr
mstlp:  ...
        JSR     foo
        JSR     bar
        ...
mstbr:  TST.L   glob
        BGT     mstlp
        RTS

.section .data
.global glob
glob:  .dc.l   16
```

図 1.4: アセンブリ言語プログラム分割例 (TEST1(左)，TEST2(右))

3. C 言語で書いた単一のプログラムを m68k-elf-gcc でアセンブリ言語プログラムに翻訳する．続いて，これを m68k-elf-as でオブジェクトコードに変換し，最後に m68k-elf-ld で実行可能コードに変換する．

4. C 言語で書いた複数のプログラムを m68k-elf-gcc でアセンブリ言語プログラムにそれぞれ翻訳する．続いて，これらのアセンブリ言語プログラムを m68k-elf-as でそれぞれオブジェクトコードに変換し，最後に m68k-elf-ld で単一の実行可能コードにまとめる．

5. C で書かれたプログラムを m68k-elf-gcc でアセンブリ言語プログラムに変換する．このアセンブリ言語プログラムと，もともとアセンブリ言語で書かれたプログラムをそれぞれ m68k-elf-as でオブジェクトコードに変換し，最後に m68k-elf-ld で単一の実行可能コードにまとめる．C ソースプログラム・アセンブリ言語ソースプログラムともに複数あっても同様の手順で開発できる．

ソフトウェア実験第 1 の簡易 OS 作成では，1 あるいは 2 の手順を用いてプログラムを作成した．今回は，5 の方法でプログラム開発を行う．

外部名参照

1.2.1 節で述べたように，C 言語プログラムを複数のファイルに分けて分割コンパイルした場合，それらの間で発生する外部名（関数名，大域変数名）の参照が，リンク時に実際のサブルーチンの先頭番地やメモリ上のデータ格納場所へと変換される．

このような外部名参照は，C ソースファイル間においては extern 宣言，アセンブリ言語ソースファイル間においては .extern ディレクティブによって明示される．これらの宣言／ディレクティ

ブは，コンパイラ／アセンブラに対して，それらのシンボルが外部ファイルで定義されていることを示すのに用いられる．コンパイラ／アセンブラは，それらのシンボルがリンク時に実アドレスに変換されなくてはならないということを示す情報をオブジェクトファイル内に残す．

では，プログラムの一部を C で，残りをアセンブリ言語で記述した場合は，それらの間の外部名参照をどのように解消すればよいのだろうか．このためには，C プログラムが外部名参照を残したままアセンブリ言語プログラムに翻訳される際の名前の変換規則を利用して，それを表す extern 宣言と.global/.extern ディレクティブを書いておけばよい．

今回の実験で使用する m68k-elf-gcc では，C プログラム中で用いられる関数・大域変数を次のような規則に基づいて変換する．

- 関数は，サブルーチンに翻訳する．そのサブルーチンのラベルは，関数名と同一になる．
- 大域変数の定義は，メモリ上でそれが保持される領域を確保するためのディレクティブに翻訳する．そのラベルは，変数名と同一になる．
- extern 宣言された関数名・変数名は上記の規則に基づくシンボルを.extern 宣言する．

図 1.5 に，簡単な C プログラムとそれを m68k-elf-gcc で処理した場合のアセンブリ言語プログラム（コンパイラの出力するコメントや不必要なセクション切替えは省略した）の例を示す．

```
int local;                          .section .text
extern int external;                        .global func2
extern void func1();                func2:
void func2()                                MOVE.L  external, local
{                                           JSR     func1
  local = external;                         RTS
  func1();
}                                   .section .bss
                                    .extern external
                                    .extern func1
                                    .global local
                                    local:  DCB.B   4,0
```

図 1.5: C プログラムとアセンブリ言語プログラムの例

したがって，C プログラムおよびアセンブリ言語プログラムの間で外部参照を行う場合は，それぞれで次のような注意をすればよい．

1. C プログラムからアセンブリ言語プログラムのサブルーチン・メモリを参照する場合
   C プログラム中では参照する関数名・変数名を extern 宣言して，通常通り使用する．アセンブリ言語プログラムの方では，C プログラム側で使用している関数名・変数名をラベルとして，サブルーチン・領域確保ディレクティブを書くとともに，これらのシンボルを.global 宣言する．
2. アセンブリ言語プログラムから C プログラムの関数・変数を参照する場合
   アセンブリ言語プログラム側では，シンボルを用いてサブルーチン呼び出しやメモリ参照を書くとともに，これらを.extern 宣言する．C プログラム側では，これらのシンボルの名前を関数・変数の名前として定義する．

関数呼び出し

1.2.2 節で述べたように，関数呼び出しはサブルーチンコールに翻訳される．しかし，C における関数の引数や戻り値などにも注意が必要である．

引数の渡し方

m68k-elf-gcc では，関数呼び出しの際の引数はおおむね以下のように処理された後，サブルーチンコールの命令が生成される．個々のデータ型による詳細はマニュアル参照のこと．

1. 引数は**右から左へ**順に評価されて，スタックに積まれる．
2. 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．
3. 配列以外の引数（構造体も含む）は，評価した値がスタックに積まれる．
4. 配列は配列のコピーではなくそのアドレスがスタックに積まれる．

サブルーチン側ではこれらの引数をスタックから取り出して処理に使用する．サブルーチン終了後不要になった引数をスタックから取り除く（スタックポインタを移動させる）ためのコードは呼び出し側のサブルーチン命令の直後に置かれる．

実際に C から呼ばれる関数をアセンブリ言語で書く際には，呼ばれる関数と同じ引数の並びを持つ簡単な C 関数を書いてみて，これをコンパイルした結果を見ながら（あるいは書き直しながら）アセンブリ言語サブルーチンを作るとよい．

アセンブリ言語プログラムから引数のある C 関数を呼び出す場合には，呼び出される C 関数をコンパイルした結果を見て，スタックに必要な情報を積んでからサブルーチンコールするようにすればよい．その C 関数をただ呼び出すだけの C 関数を作ってそのコンパイル結果を真似るのもよいであろう．

関数からの戻り方

C 関数に対応するアセンブリ言語サブルーチンから戻るとき，その戻り値が 32 ビットに収まる場合には D0 レジスタに置かれる．それ以外の場合は，呼出し側がスタックフレームを用意してその先頭アドレスを A1 レジスタで指定する場合があるので，指定した 領域に値をコピーするなど，いくつかのバリエーションがある．

通常は，C から呼ばれるアセンブリ言語サブルーチンが戻り値を持つ場合には，それを D0 レジスタに格納した状態で RTS するように書けばよいであろう．

詳細が必要な場合は，引数の授受のみを行うような C 関数をコンパイルし，コンパイラが生成するアセンブラソースを参照すればよい．

逆に，アセンブリ言語プログラムで C 関数を呼び出す場合には，D0 レジスタから戻り値を受け取るようにプログラムを記述する必要がある．

関数本体でのレジスタの使い方

m68k-elf-gcc でコンパイルされた C 関数はいくつかのレジスタの値を保存せずに使用する場合がある．したがって，アセンブリ言語プログラムから C 関数を呼び出す場合，関数から戻った時点では，呼び出し前の時点のレジスタの値は破壊されている可能性がある．必要なら事前に他のレジスタにコピーするか，スタックに積んでレジスタの値の保存を明示的に行なう必要がある．

C プログラムからアセンブリ言語サブルーチンを関数として呼ぶ場合には，そのサブルーチンの中ではレジスタの値をスタックに積むなどして，呼び出し側に戻る前に復元しておく必要がある．

1.3 C 言語処理系 m68k-elf-gcc のカスタマイズ [実験準備]

1.3.1 m68k-elf-gcc のカスタマイズのために変更／作成を要するファイル

ターゲットの上で C 言語プログラムが実行できる環境を構築するにあたり，ソフトウェア実験第 1 で作成した簡易 OS のハードウェア初期化機能およびシリアル入出力等のシステムコール実装を利用する．以下，簡易 OS をモニタと呼称し，モニタプログラムを mon.s とする．

この実験でこれから作成するシステムは、m68k-elf-gcc を特定のターゲットのためのクロスコンパイラ[1]としてカスタマイズするためには，システムに付いてくる以下のファイルを変更もしくは作り直さなければならない．

1. `crt0.s`
   BSS セクション初期化，スタックポインタ・ヒープポインタ初期化の処理などを記述するためのアセンブリ言語プログラムで，必要な処理を行った後，関数 `main()` を呼び出す．今回の実験では，事前に用意されているので，変更の必要はない．

2. `sbrk.s`
   ヒープ領域（関数 malloc() などでプログラムが使用する記憶領域）の管理のための関数に対応するアセンブリ言語ルーチン．今回の実験では使用しない．

3. `csys68k.c`
   関数 `main()` の起動やファイルのオープン・クローズなどの処理のための C の関数群．キーボードからの入力を画面にエコーバックするための処理は，この中の関数 `read()` に書いておくとよい．今回の実験では，事前に用意されているので，変更の必要はない．

4. `inchrw.s`
   標準入出力ライブラリに含まれる入力系のすべての関数を構成する元になる一文字入力関数 `inbyte()` が収められている．したがって，特定のターゲットにカスタマイズするためには，この関数をハードウェアの仕様に合わせて改造すればよい．今回は，この関数をモニタの `GETSTRING` システムコールを用いてアセンブリ言語で実現する．

5. `outchr.s`
   標準入出力ライブラリの中のすべて出力系関数を構成するもとになる一文字出力関数 `outbyte()` が含まれる．今回は，この関数をモニタの `PUTSTRING` システムコールを用いてアセンブリ言語で実現する．

[1] クロスコンパイラとは，プログラムを実行する CPU とは異なる CPU 上でコンパイルすることができるコンパイラである．本実験の場合，プログラム開発用のノート PC でコンパイルし，68000 互換 CPU 搭載のターゲットボード上で実行する．同様に今回使用するアセンブラもクロスアセンブラと呼ばれる．

1.3.2 m68k-elf-gcc における C プログラムの起動

C のプログラムにおいては，関数 `main()` が最初に起動される．m68k-elf-gcc によってコンパイルされた実行可能プログラムの `main()` 本体の実行されるまでの手順は次の通りである．

1. `crt0.s` のメインルーチンが 400 番地に配置され，ここからプログラムの実行を開始する．このルーチンでは BSS セクションを初期化し，スタックポインタ，ヒープポインタを初期化する．
2. 上記ルーチンの最後で `main()` を起動する．
3. C コンパイラがコンパイル時に `main()` の先頭に挿入した `__main()` 呼び出しにより，`__main()` を実行する．`__main()` は C の標準ライブラリ初期化を行う．具体的には，変数の初期化，入出力バッファの初期化，標準入出力ストリームのオープンを行う．
4. 以降，ユーザが記述した `main()` の内容が実行される．

ただし，今回の実験では，1. の部分が改造してあり，BSS セクションの初期化の後にモニタプログラムの初期化ルーチン (`mon.s` の該当部) を呼び出す[2]．そのため，図 1.6 に示すように，

- モニタプログラム (`mon.s`) の初期化開始位置に `monitor_begin` のラベルを設置
- モニタプログラムの初期化部分の終りに `jmp start` を挿入

の変更を施す．こうすることで，`crt0.s` での BSS セクションの初期化後，モニタプログラムの初期化ルーチンを実行し，再度 `crt0.s` の `start` ラベルに戻って，`crt0.s` の処理を継続することになる．また，`mon.s` では，`monitor_begin` は `.global` 宣言する必要があり，`start` は `.extern` 宣言する必要がある．

C で書かれたプログラムを実行する場合には，上のように改造したモニタ，システム附属の `crt0.s`，`csys68k.c`，さらに自作の `inchrw.s` と `outchr.s`，これらすべてから作成されるオブジェクトコードと main() を含む自分の C プログラムとを合わせて 1 つの実行可能コードを作成するとよい．この実行可能コードを，68000 システム起動後のプロンプトにしたがってロードすることになる．

このためには，ローダの起動オプションもしくはローダコマンドファイルを用いて，これらすべてのオブジェクトファイルをリンクする．これらの内容は，Makefile 中に既に記述されている．

```
.extern start
.global monitor_begin
monitor_begin:
        ...
        jmp start
```

図 1.6: mon.s の変更点

[2] `crt0.s` の先頭は本来 `start` ラベルで始まるが，今回の実験で使用する `crt0.s` では，BSS セクションを 0 で埋める処理の後に `start` ラベルを移動し，ラベル直前に `jmp monitor_begin` を挿入している．これは，前期に作成したモニタプログラムの初期化部分で BSS セクションにデータを書き込んでいる場合に，`crt0.s` の BSS セクションの初期化 (BSS セクションすべてに 0 を書き込む) で上書きしてしまうことを防ぐためである．

1.4 作成内容 [実験内容]

C 言語処理系を特定のターゲットのクロスコンパイラとしてカスタマイズするために重要な作業の 1 つは，入出力関数が使えるようにすることである．C ではすべての入出力関数が言語の組み込み関数ではなくライブラリ関数として言語処理系の「外」に作られている．したがって，移植の際には，標準入出力ライブラリがターゲットの上で動作するようにすればよい．

m68k-elf-gcc では，ライブラリ関数のオブジェクトコードは libc.a に収められている．これらのライブラリ関数のうち入出力に関するのは以下の 5 つの関数を用いて実装されており，これらをカスタマイズすれば m68k-elf-gcc のすべての入出力関数がそのまま使えるようになる．

- `close()` …ファイルのクローズ
- `create()` …ファイルの作成
- `open()` …ファイルのオープン
- `read()` …ファイルからの任意のバイト数のデータの読み込み
- `write()` …ファイルへの任意のバイト数のデータの書き込み

今回のハードウェアではシリアルポートを利用した `read()`, `write()` のみを使用する．

`read()`，`write()` はファイル `csys68k.c` の中にあり，これらの関数を見ると，`read()` と `write()` はそれぞれ以下の関数を用いて実装されていることがわかる．

- `inbyte()` …1 文字入力関数
- `outbyte()` …1 文字出力関数

したがって，シリアルポートに対する `read()`, `write()` を使えるようにするには，上記二つの関数をソフトウェア実験 I で作成したモニタの `GETSTRING` システムコールと `PUTSTRING` システムコールを用いて実装すればよい．また，これにより，`scanf()` や `printf()` も使えるようになる．なお，アルファベット 1 文字は C 言語では char 型で記述し，長さは 1 バイトである．

モジュール構成図を図 1.7 に示す．例えば，`main()` から `printf()` を呼び出すと，`printf()` が `write()` を，その後順に `outbyte`，`PUTSTRING` システムコールが呼び出され，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れる．また，図中の `libc.a` などはファイル名であり，`printf()` など `()` で終わるものは C 言語の関数を表し，`outbyte:` など `:` で終わるものはアセンブラ関数 (サブルーチン) を表している．

1.4.1 inbyte() のアセンブリ言語による作成

`inbyte()` は，引数をとらず，戻り値としてシリアルポート 0 から読み込んだ 1 文字 (char 型) を返す関数である．アセンブリ言語で `inbyte` なるラベルを持つサブルーチンを作ればよい．サブルーチン `inbyte` は，その内部でモニタのシステムコール `GETSTRING` を呼び出せばよい．`inbyte()` は，`csys68k.c` の `read()` から呼び出される．

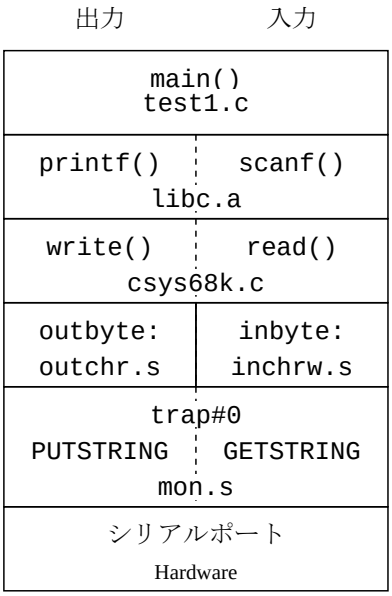

図 1.7: モジュール構成図

1.4.2 outbyte() のアセンブリ言語による作成

outbyte() は，引数として char 型のデータを取り，これをシリアルポート 0 に出力する関数である．戻り値はなくてよい．1.2.2 でも述べたように，関数が呼び出されるとき，char 型の引数は 32 ビットに符号拡張されてスタックに積まれる．サブルーチン outbyte では，出力すべき char 型の 1 バイトが格納されているスタックのアドレスをレジスタにセットし，モニタの `PUTSTRING` システムコールを発行すればよい．`outbyte()` は，`csys68k.c` の `write()` から呼び出される．

`inbyte()` や `outbyte()` は必ず 1 文字を入出力する関数とすること．つまり，`PUTSTRING` システムコールで 1 文字出力が成功しなかった場合はリトライする必要がある．これに限らず，システムコールを発行する場合はシステムコールの入出力および動作仕様を確認し，必要があればシステムコール発行後の評価を行うべきである．

1.4.3 コンパイルとテスト

コンパイルには make コマンドを用いる．make test1 コマンドによって，`crt0.s`, `mon.s`, `inchrw.s`, `outchr.s`, `csys68k.c`, `test1.c` ファイルがコンパイルされて，`test1.abs` ファイルが生成される．make コマンドは更新されたファイルのみコンパイルしなおす．すべてのファイルのコンパイルをやり直す場合は，make clean コマンドを実行した後，make test1 コマンドを実行する．make コマンドによるこれらの動作指示は，Makefile およびその内部で呼び出される Makefile.1, Makefile.2, Makefile.3 に記述されている．

入出力を行う簡単な C プログラムを書いてコンパイルし，モニタ・crt0・csys68k・inchrw・outchr のモジュールとリンクして，1 つの実行可能コードを作成する．これを 68000 側の IPL 機能を用いてロードし，入出力が正しく行われていることを確認する．

make test1 を実行した際の出力例を以下に示す.

```
guest@pcs0xxx% make test1
make  LIB_JIKKEN=...  -f Makefile.1
make[1]:  Entering directory ...
m68k-elf-as -m68000  -ahls=crt0.glis -o crt0.o crt0.s
m68k-elf-as -m68000  -ahls=mon.glis -o mon.o mon.s
m68k-elf-as -m68000  -ahls=inchrw.glis -o inchrw.o inchrw.s
m68k-elf-as -m68000  -ahls=outchr.glis -o outchr.o outchr.s
m68k-elf-gcc -c -g -O2 -Wall -m68000 -msoft-float -I.  -Wa,-ahls=csys68k.glis
    -o csys68k.o csys68k.c
m68k-elf-gcc -c -g -O2 -Wall -m68000 -msoft-float -I.  -Wa,-ahls=test1.glis
    -o test1.o test1.c
m68k-elf-ld -nostdlib --cref crt0.o mon.o inchrw.o outchr.o csys68k.o test1.o
    -o test1.hex -Map test1.map -T /m68k-5.0/lib/soft-jikken/ldscript.cmd
perl .../sconv.pl test1.hex > test1.abs
python .../gal.py test1.map crt0.glis mon.glis inchrw.glis outchr.glis csys68k.glis
    test1.glis > /dev/null
make[1]:  Leaving directory ...
guest@pcs0xxx%
```

注意 1: inchrw, outchr の実装が正常であっても、モニタの出来具合によって動作しない事があり得る．従って、正常動作しない場合はモニタの差し替えを検討項目に加えておく．モニタの差し替えは、配布ファイルの mon_r.o を mon.o にコピーする事で実現できる．(mon_r.o は mon_r.tgz 内にもある．) mon_r.o のシステムコール一覧は mon_r.txt に記述されている．

注意 2: システムコール発行時におけるモニタ内の作業領域の利用を禁じる．プログラムの見通しを良くする意味においても，ローカルな作業領域は各プログラム内で定義するべきである．更に今後のマルチタスク環境を見越した場合，関数呼出し側に固有の作業領域を検討する必要があるかも知れない．理由は各自検討せよ。

参考: 関数内部で一般変数を使った C プログラムのコンパイルのみ (gcc -S foo.c) によって得られるアセンブラコード (foo.s) 内で現れる命令 link, unlk の説明

mnemonic	action
link.w %a6,#nnn	#nnn:word value, %sp ← %sp−4, (%sp).l ← %a6, %a6 ← %sp, %sp ← %sp + #nnn
	access by #mmm(%a6) , #mmm ∈ [0,#nnn]
unlk	%a6 ← (%sp).l %sp ← %sp+4

第2章　テーマ２：マルチタスクカーネルの製作

2.1　はじめに

テーマ１の実験により，C 言語で書かれたプログラムをターゲットの上で実行することができるようになった．

テーマ２では，複数の「仕事」を C の関数として記述し，これを必要に応じて切り替えながら実行するマルチタスク処理の実験を行う．マルチタスク処理とは１つのシステムの上で複数のタスクを切り替えながら実行するような処理のことである．

タスクが切替えられるのは，以下の場合が考えられる．

1. 一般に，タスクが共有資源を利用する場合，排他制御をする必要があり，これを実現するためにタスク切替えが起こる．
2. 一つのタスクが CPU を独占して実行されることがないように，連続して実行できる時間の上限を与える場合，タスク切替えが起こる．

この両者に起因するタスク切替えを実現するために，タスクを 2 種類のキュー (待ち行列) に並べる．一つは ready キューで，自分の順番が来たならば，いつでも実行を再開できるタスク (実行待ちの状態にあるタスク) の待ち行列である．もう一つは，各共有資源毎のキュー (実際には後述する各セマフォが持つキュー) で，その共有資源が他のタスクにより使用されているために休眠状態に入っているタスクの待ち行列である．後者のキューに並んでいるタスクは，その共有資源が利用できる順番が来ても，すぐに再開されるのではなく，休眠状態から起こされるだけで，一旦 ready キューに並ばされ，実行待ちの状態になる．

まず，タスク切替えのために必要となる以下の基本事項

1. 特権命令の概念
2. P・V システムコールの概念
3. タスクを切替えながら実行するために必要な処理
4. タスク切替えのタイミングによるカーネル再入防止

を解説した後，具体的な処理の内容について解説する．

2.2　基本的事項

2.2.1　特権命令の概念

ある特別な作業に対して，OS には実行できるが利用者プログラムでは実行できないといった区別が必要になることがある．このような区別をするために以下のような２つのモードを設ける．

- スーパーバイザモード (supervisor mode)：全ての命令が実行可能
- ユーザモード (user mode)：一部の命令が実行不可能

スーパバイザモードでだけ実行でき，ユーザモードでは実行できない命令を特権命令と呼ぶ．このような異なる実行モードを実現するために，計算機のハードウェアにモード指示のためのビットが付加されていることが多い．今回の実験で使用する MC68VZ328 においても，ステータスレジスタの中にそのためのビットが用意されている．詳しくは前期テキスト「簡易 OS の開発」を参照すること．

オペレーティングシステムにおいてスーパバイザモードで実行すべきものには，例えば以下のようなものがある．(以降，スーパバイザモードで実行すべき命令列も特権命令と呼ぶ)

1. **入出力**
   全ての入出力命令は特権命令とされる．利用者プログラム側で入出力を実行したい場合は，システムコールを実行し，スーパーバイザモードで動作する OS に入出力作業を要求する．
2. **タスク切替えのためのタイマのセット・アンセット**
   利用者プログラムに制御を渡す前に OS がタイマのカウンタ設定とタイマ割り込み設定を行なっておく．タイマが割り込みを発生すると，利用者プログラムが動作中であっても OS に制御が渡される．このようなタイマの設定は特権命令でなければならない．

これらの操作は，前期に作成したモニタにおいては，TRAP 命令を利用したシステムコールとして実現されている．したがって，タスクがユーザモードで実行されているときにこれらのシステムコールを発すると，スーパバイザモードに切り替わり，必要な処理を行ったのち，ユーザモードに戻ってタスクの実行を再開することができる．

このような処置は，ユーザタスクが誤った処理を行ってシステムに重大な障害を与えるのを防ぐのに役立つ．

2.2.2 P・V 命令の概念

２つ以上のタスクが共有資源を同時に操作しようとすると，共有資源の完全性が保証されなくなる場合がある．このような資源は同時に複数のタスクがアクセスできないようにしなければならない．そのための制御が排他制御 (相互排除) である．

これを実現するために考案された代表的なものに，**セマフォ**とセマフォを用いた P・V 命令がある．セマフォの構成法はいくつかあるが，ここでは，セマフォがカウンタとキュー (リスト) から構成される場合を考える．カウンタはシステム初期化時には 1 にセットされていて，カウンタ値が 1 の場合は，そのセマフォ(あるいはそのセマフォが管理する共有資源) を使用しているタスクがないことを，0 以下の場合は，そのセマフォを使用しているタスクがあることを，0 未満の場合は，そのセマフォを使用しようとして待っているタスクがあることを意味する．また，キューはそのセマフォを使用しようとして待っているタスクの待ち行列である．セマフォ S を引数とした P 命令は，セマフォS を利用するための要求，セマフォ S を引数とした V 命令は，セマフォ S を開放する命令である．

あるセマフォ S を引数としてユーザタスクが P 命令を発行すると，セマフォ内のカウンタを 1 減らし，その結果のカウンタ値が，

- 負であれば (上記で減じる前の値は 0 以下であるからそのセマフォを利用しているタスクが存在することになる)，そのユーザタスクを休眠状態にするために，タスクをセマフォ内のキューに入れて他のユーザタスクに切替える．
- 0 であれば，なにもしない (つまりユーザタスクをそのまま続行する)．

一方，S を引数として V 命令を発行すると，セマフォ内のカウンタを 1 増やし，その結果のカウンタ値が，

- 0 以下であれば (上記で 1 増やす前は 0 未満であるから，そのセマフォを利用しようと待っているタスクが存在することになる)，そのセマフォの待ち行列からタスクを取り出して，ready キューに入れる．
- 正であれば，なにもしない．

たとえば，共有資源 C の使用前にその共有資源に対応したセマフォ S_c に対して，P 命令を発行した後，C を利用し，終了したときに S_c に対して V 命令を発行するようにユーザタスクを書くならば，P 命令は共有資源の使用前にその資源にロックをかける操作，V 命令は使用後にロックをはずす操作と考えることができる．また，あるタスク A がセマフォ S に対して P 命令を発行し，別のタスク B が S に対して V 命令を発行するようにすると，タスク A と B の間に処理の依存関係があることを示せる．つまり，P・V 命令の使用により，排他制御だけでなく，特定のタスクの実行を望みの時点で中断して他のタスクに切替えることも実現できる．今回実験で使用するには，セマフォが管理する共有資源がないため，上記の後者を試すことになる．

なお，P・V 命令は，タスク切替えやセマフォ内のキューの操作を伴うため，P・V 命令の処理はスーパバイザモード (つまり，特権命令として) 行わなければならない．したがって，P・V 命令は，OS のシステムコールとして実現されることになる．

2.2.3 タスクを切り替えながら実行するために必要な処理

マルチタスク処理を行う場合，タスクのプログラム中の文の途中で実行が中断され，あとで再開されることもある．このとき，再開後のタスクの実行結果が，これらを切り替えずに実行した場合と同じになるようにする必要がある．そのためには，次の 2 つの条件が必要になる．

1. タスクの実行が中断されるときには，その瞬間の計算機内部の「状態」が正しく保存（記録）される．
2. 中断していたタスクの実行が再開されるときには，1 で保存した「状態」が回復されて，中断したところから実行が再開される．

タスクの実行中の状態を保存し，あとで実行を再開するためには，少なくとも以下のような情報が保存されていなければならない．

- 実行が中断したときのプログラムカウンタの値
- 実行が中断したときのその他のレジスタの値
- 実行が中断したときのスタックの内容

2.2.4 タスク切り替えのタイミングによるカーネル再入防止

タスク切替えが行なわれるのは，P 命令により現行のタスクが休眠状態に入る場合と，タイマ割込みによる切替えとがある．カーネルタスクの走行中に P 命令が発行されることはない (P 命令はユーザタスクが発行する)．一方，タイマ割込みはカーネルタスクの走行中にも発生する可能性がある．このときタスク切替えを行なうと各種キューなどの管理情報の整合性を壊す危険がある．たとえば，カーネルタスクがユーザタスクの切替えを行なっている場合にタイマ割込みが起こり，タスク切替えを行なうと，ユーザタスクを管理している情報が破壊されることがある．

これを避ける方法として，カーネルタスクの走行中に発生したタイマ割込みでは，ユーザタスクの切替えは行なわず，そのまま割込み前の状態に復帰するという方法が考えられる．これは，UNIX が採用している方法である[1]．

しかし，今回の実験では，より簡単に，カーネルタスクの走行中に発生したタイマ割込みを受け付けないようにすることでこれに対処する．今回の実験では，タイマ割込みを利用したタスク切替え以外のカーネルタスクは P・V 命令の処理だけであるから，P・V 命令の処理をする後述の アセンブラルーチン `pv_handler` の最初に割込み禁止 (走行レベル 7) にし，最後に走行レベルを元に戻せばよい．ただし，上記のような方法を取ると，タスクが連続して実行されている時間の上限は厳密に一定というわけではなくなる．たとえば現在走行中のタスク A の連続実行可能時間が過ぎようとしたちょうどそのとき，そのタスク A が，`putstring` システムコールを呼び出し，その処理をカーネル (モニタプログラム) が行なっているとすると，タイマ割込みによるユーザタスクの切替えは起こらず，システムコール終了後，再びタスク A の実行が続行されることになる．

2.3 MC68VZ328 におけるタスク切り替えの概要 [実験準備 1]

以下，本実験で使用する CPU である MC68VZ328 の上でのマルチタスク処理の方法を検討する．

複数のタスクを並行に走らせるとき，CPU 上で実行するタスクを切り換えるタイミングとしては，例えば次のようなものがある．

1. そのタスクに割り当てられた時間を終了した
   →次の順番が回ってくるまで待たされる

2. P 命令を発行したが，そのセマフォが利用可能でなかった
   →セマフォが利用可能になるまで待たされる

3. 入出力要求を出した
   →入出力が終了するまで待たされる

ここで，今回扱うのは，1,2 の要因によってタスクの切り換えを発生させる場合である．2.2.2 節で述べたように，P・V 命令は TRAP を用いたシステムコールとして実現される．また，前節で述べたように，タイマ割り込みによりユーザタスク A の実行を中断して他のタスクに切替える場合は，このタイマ割り込みはユーザタスク A の走行中に発生した割り込みであるため，このタイマ割り込みを，ユーザタスクが疑似的に発行したシステムコールとみなすことにより，P・V 命令とほぼ同様に実現できることがわかる．そこで，本節では，2 について考える．

[1]次に述べるタイマ割込み禁止という対処法と異なり，この方法では，一旦タイマ割込みを受け付けることにより，タイマ割込みが発生したことを記録でき，カーネルタスク終了時にそれを見て，必要ならば，タスク切替えをすることができる．

タスクAが実行中にP命令を発行するが，その引数のセマフォが利用可能でないため，中断される (スリープされる)．そのセマフォが利用可能になり，タスク A を実行させる順番が来たとき，元のタスクA を以前中断した点（の直後）から再開しなければならない．再開するためには，割り込みの際にどのような処理が必要になるだろうか．

まず，割り込みが発生したときのCPU内部の状態の変化を見てみる．簡単のため，どのタスクもスタックとレジスタにあるデータのみを操作するものと仮定する．

ユーザモードで実行されているタスクAが図2.1のような状態でユーザスタックを使用しながら動作しているとする．ここで割り込みが発生すると，MC68VZ328内部では次のような処理が自動的に行われる．

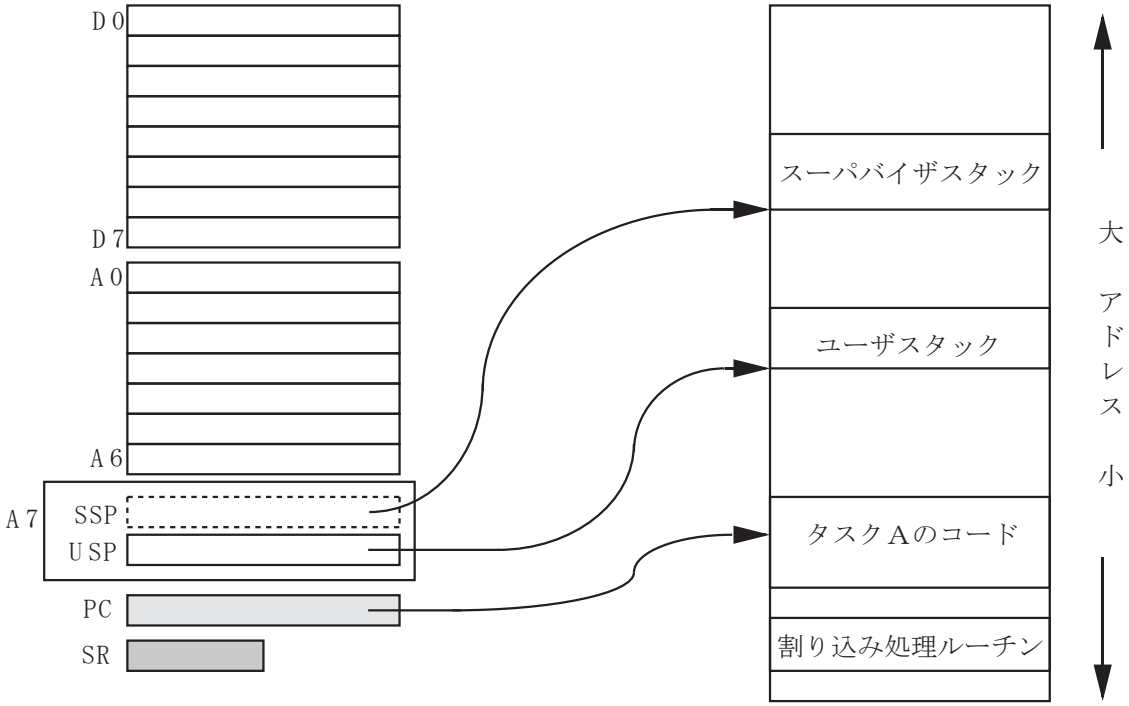

図 2.1: ユーザタスクA実行中の状態（割り込み直前）

1. **スーパバイザモードへの移行**
   TRAP命令が発行されることにより，MC68VZ328はスーパバイザモードに移行する．このとき，ステータスレジスタ (SR) の内容をCPU内部のシステムレジスタ（プログラムからはアクセスできない）にコピーし，SRのビット13（Sビットと呼ばれる）を”1”にセットする．このビットが立つと，スーパバイザモードと呼ばれる状態に移行する．このとき，スタックポインタはユーザモード用のUSPからスーパバイザモード用のSSPに切り替わる．

2. **例外ベクタのアドレスの生成**
   例外ベクタ番号を用いて，例外ベクタ (割り込み処理ルーチンの開始アドレス) を格納しているアドレスを割出し，そのアドレスをCPU内部のシステムレジスタ (上記のSRをコピーしたシステムレジスタとは別のシステムレジスタ) に置く．

3. **割り込み発生時のCPU内のデータの退避**

スーパバイザスタックに，プログラムカウンタ（PC）の値（実行中の命令の次の番地）をプッシュする．さらに，割り込み直前の SR の値もプッシュする．したがって，SSP の値は 6 だけ減る．

4. **例外ベクタの読み出し**
   例外ベクタを格納しているアドレス (これは，上記の 2 でシステムレジスタに格納されている) から例外ベクタを取り出して PC に格納し，そのアドレスの命令から割り込み処理を開始する．この時点でのシステムの状態を図 2.2 に示す．

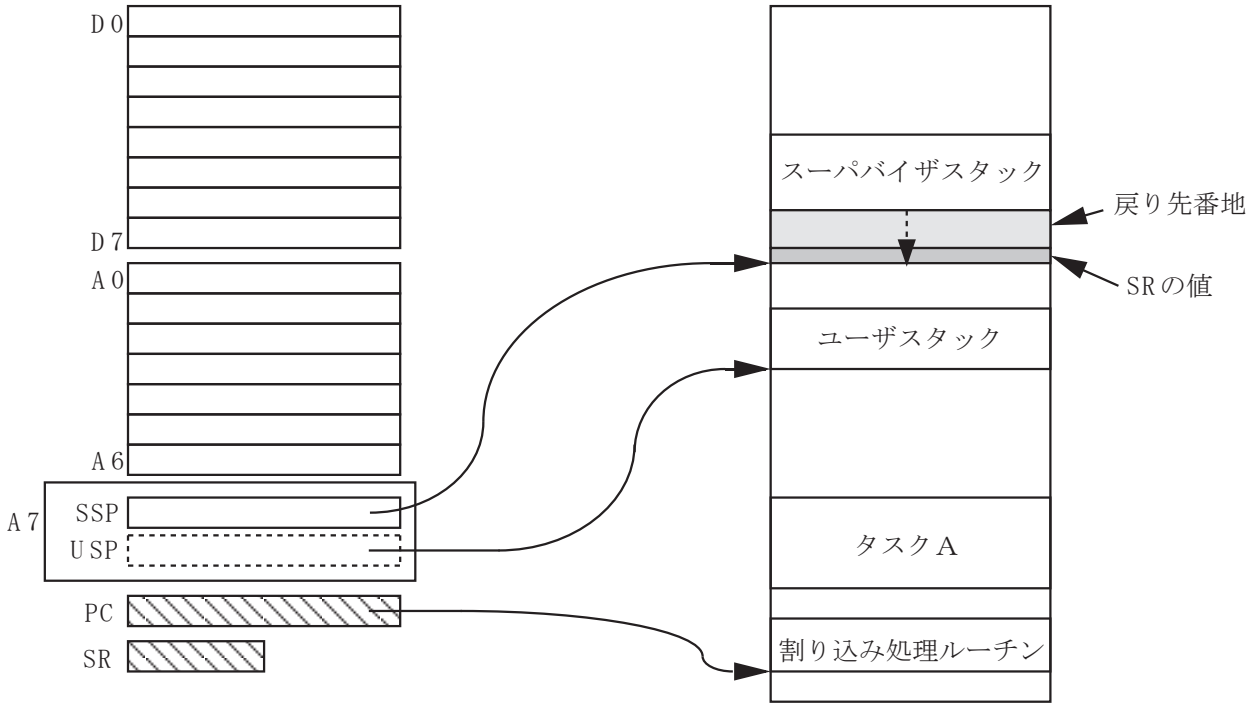

図 2.2: 割り込み処理開始時の状態

割り込み処理ルーチンの本体は，その開始アドレスが例外ベクタが示すアドレスとなるように配置されている．割り込み処理ルーチンの最後では，必ず RTE 命令を実行する．

5. **割り込み処理からの復帰**
   RTE 命令は次のような処理を行う．

   - SSP が指す位置から 2 バイトを SR に戻す．
   - SSP+2 番地から 4 バイトのデータ（戻り先番地）を PC に戻す
   - SSP を 6 増やす．

以上の割り込み処理では，スーパバイザスタックがユーザスタックと別に使用され SR と PC が保存されるため，ユーザスタックの状態とステータスレジスタ・プログラムカウンタの値はうまく回復される．

しかし，タスク A 実行中に発生した割り込み処理の直後に，他のユーザタスクを実行することなくタスク A の実行を再開する場合でも，中間で実行される割り込み処理は割り込みがかかった

時点のレジスタの値を破壊する可能性がある．さらに，実際に割り込み処理の後他のユーザタスク B に実行が移る場合には，タスク B が中断されタスク A に戻るタイミング次第では，USP の値やユーザスタック自体も，タスク A の実行を中断した時点の状態とは異なるものになっている可能性があり，タスク A の実行を元通りに再開することはもはや不可能である．

以上のような考察から，MC68VZ328 のタスク切り替えについて，次のようなことがわかる．

- **タスクごとのユーザスタックが必要である**
  各タスクは，実行途中の「状態」を保持していなければならない．したがって，各タスクに１つそのタスク専用のユーザスタックを設け，タスクを切り換える際にはユーザスタックも切り換えることによって，他のタスクのユーザスタックを破壊しないようにしなければならない．

- **スーパバイザスタックには全レジスタの値を保存する必要がある**
  タスク切り換え用の割り込み処理ルーチンが使用するスーパバイザスタックには，中断しているタスクの戻り先番地や SR の値だけでなく全レジスタの値を記録しておかなければならない．タスクを元通りに再開するためには全レジスタを割り込み直前の状態に戻してやる必要があるからである．また，ユーザスタックの切り替えと全レジスタ値の保存の２つの操作はハードウェアによって自動的に行われるものではないので，タスク切り換え用の割り込み処理ルーチンによって行わなければならない．

- **スーパバイザスタックも，タスクごとに，ユーザスタックとは別の位置に必要である**
  複数のタスクが中断状態にあるとき，それらを再開するために必要な情報（レジスタ値など）は各タスクによって異なるから，スーパバイザスタックも各タスクごとに必要である．切替える直前のスーパバイザスタックの上の方 (アドレスの小さい方) は，上から，USP の値，D0-D7/A0-A6 の値，戻り番地，SR の値であるが，その下には，実際にタスクスイッチを行なうまでにスーパバイザモードで行なわれたサブルーチンコールに伴う戻り番地等が積まれている．したがって，ユーザスタックポインタの指す最新の位置から続けてスーパバイザスタックをつくることはできず，ある程度離れた位置に別個に作る必要がある．

- **各タスクの SSP の値はスタックとは別に記録しておかなければならない**
  タスクごとのスーパバイザスタックに保存された種々の情報を回復して，中断しているタスクを再開するときには，そのタスクが中断され各情報を保存した後の SSP の値を入手しなければならない．したがって，各タスク中断時の SSP の値はスタックとは別の場所に記録しておく必要がある．

2.4　タスクスタックとタスクコントロールブロック [実験準備 2]

前節までの考察から，今回の実験で製作するシステムでは，並行に動作させるタスクのそれぞれについて，ユーザスタックとスーパバイザスタックが別個に必要になることがわかった．また，スタックのどこまでデータが積まれているかを記録する部分がスタックとは別に必要になることもわかった．

そこで，今回の実験で作成するマルチタスクカーネルでは，管理するタスクごとに以下の２種類のデータ領域を用意する．

1. **タスクスタック**
   ユーザスタックの部分とスーパバイザスタックの部分とに分かれる．今回の実験では，どちらも固定長（1024 バイトあれば十分）の領域をタスク登録時に割り当てる．

2. **タスクコントロールブロック（TCB）**
   タスクの固有情報を蓄える．テーマ２のタスク切り換えに必要な情報は割り込み直後にレジスタの退避を終えたあとの SSP の値のみであるが，この他にも，

   - ユーザスタックの限界のアドレス（＝スーパバイザスタックの起点）
   - スーパバイザスタックの限界のアドレス
   - タスクの優先度
   - タスクの開始番地

   などのデータを記録すると便利なことがある．

タスクＡが実行中，タスクＢが停止中の場合のこれらのデータ領域の模式図を図 2.3 に示す．

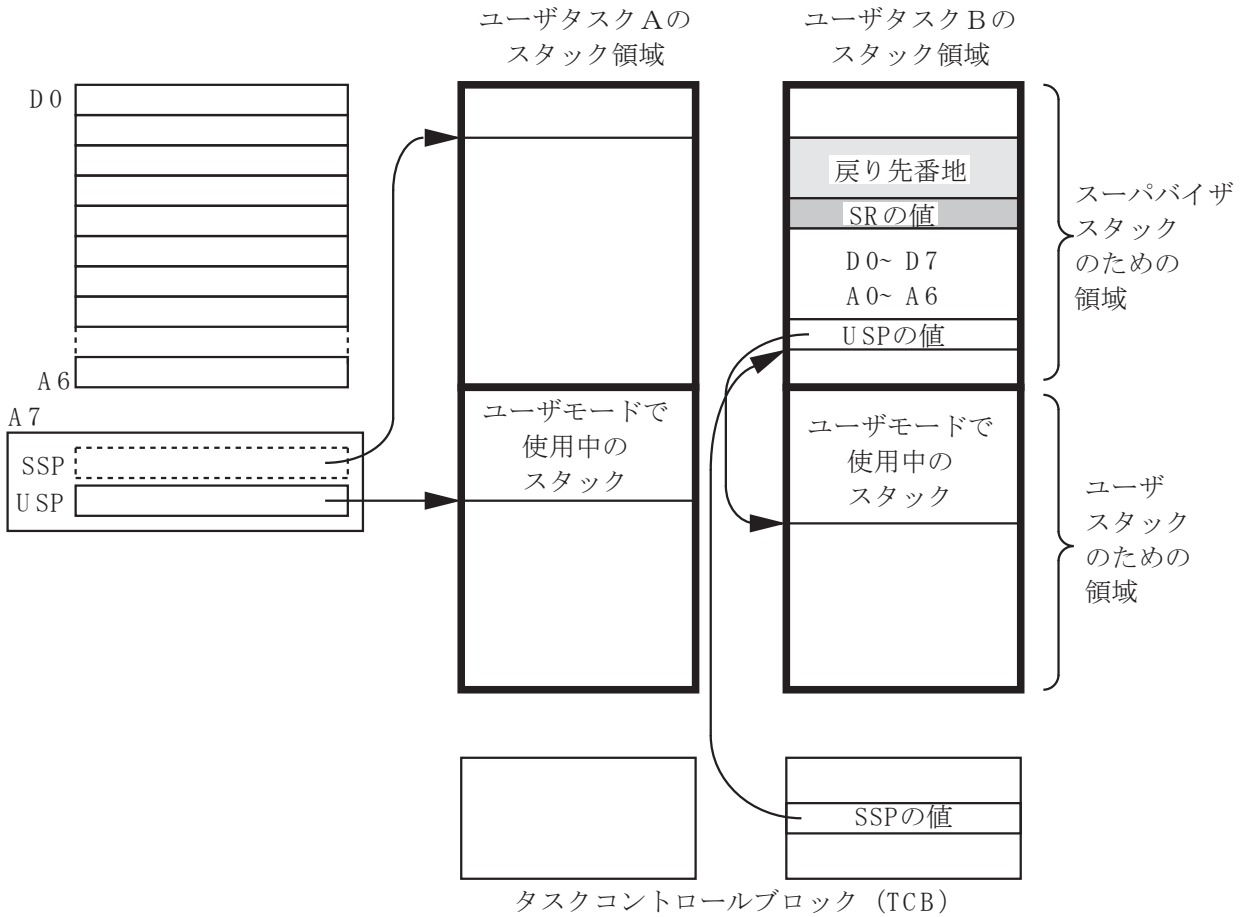

図 2.3: タスクＡ実行中の様子

図 2.3 の状態で割り込みが発生すると，現在実行中の命令の次の命令の番地およびそのときの SR の値がタスクＡのスーパバイザスタックに積まれて割り込み処理が開始される．割り込み処理ルーチンは次のような動作をすればよい．

1. タスクＡのレジスタ D0〜D7，A0〜A6，USP の値をタスクＡのスーパバイザスタックに積む．
2. 上記のレジスタ退避が終了したあとのタスクＡの SSP の値をタスクＡの TCB の中に登録する．
3. 次に実行すべきタスクを決定する．ここでは，タスクＢに決定したとする．
4. タスクＢの TCB の中から，タスクＢの SSP の値を回復する（スーパバイザスタックがタスクＡからタスクＢに切り換えられる）．
5. タスクＢのスーパバイザスタックから，D0〜D7，A0〜A6，USP の値を回復する（タスクＢのレジスタ群が回復されるとともに，ユーザスタックがタスクＢに切り換えられる）．
6. RTE 命令を実行する．

RTE 命令によって取り出される SR の値および戻り先番地は，もはやタスクＡのものではなくタスクＢのものである．5 の操作でレジスタ群やスタックはタスクＢのものに切り換えられているので，タスクＡの実行は中断され，タスクＢの実行が（前に中断した点の直後から）再開される．このときのスタックの状態を図 2.4 に示す．

2.5 作成内容 [実験内容]

通常の C プログラムとテーマ 2 の実験で使用する C プログラム (main 部) の構成の大まかな違いを図 2.5 に示す．どちらのプログラムも，`task1()`， `task2()` を実行することに違いはないが，`task1()`，`task2()` で記述される処理 (ユーザタスクと呼ぶことにする) の実行の形態が大きく異なっている．

図 2.5 左に示した通常のプログラムでは，関数 `task1()` および `task2()` を `main()` 内で呼び出して実行する．この場合，これらの関数は上から下に順次実行され，両者の間で切り替えは発生しない．

一方，図 2.5 右に示したマルチタスク実験用プログラムでは，それらのタスク関数を `main()` の中で呼び出していない．代わりに，これらを `set_task()` 関数によってユーザタスクとして登録し，関数 `begin_sch()` を呼び出してマルチタスキングを開始している．実験で製作するのは，主に，図 2.5 右の `set_task()` や `begin_sch()` のようなマルチタスク処理のための特殊な C 関数およびアセンブリ言語ルーチンである．これらの関数は，図に示したファイルとは別に作成し，最後にリンクする．上記の例では，これらの関数のヘッダが”`mtk_c.h`” に含まれているものとしている．

これらの C 関数およびアセンブリ言語ルーチンの詳細については後述する．

2.5.1 システムを構成するファイル群

本実験で作成するシステムは，C およびアセンブリ言語で実現しなくてはならない．また，カーネル（タスクの登録やスイッチを行う部分）とユーザタスクのように全く異なる性格のものも混在しているため，それらは別個のファイルにしておくほうが見通しがよい．

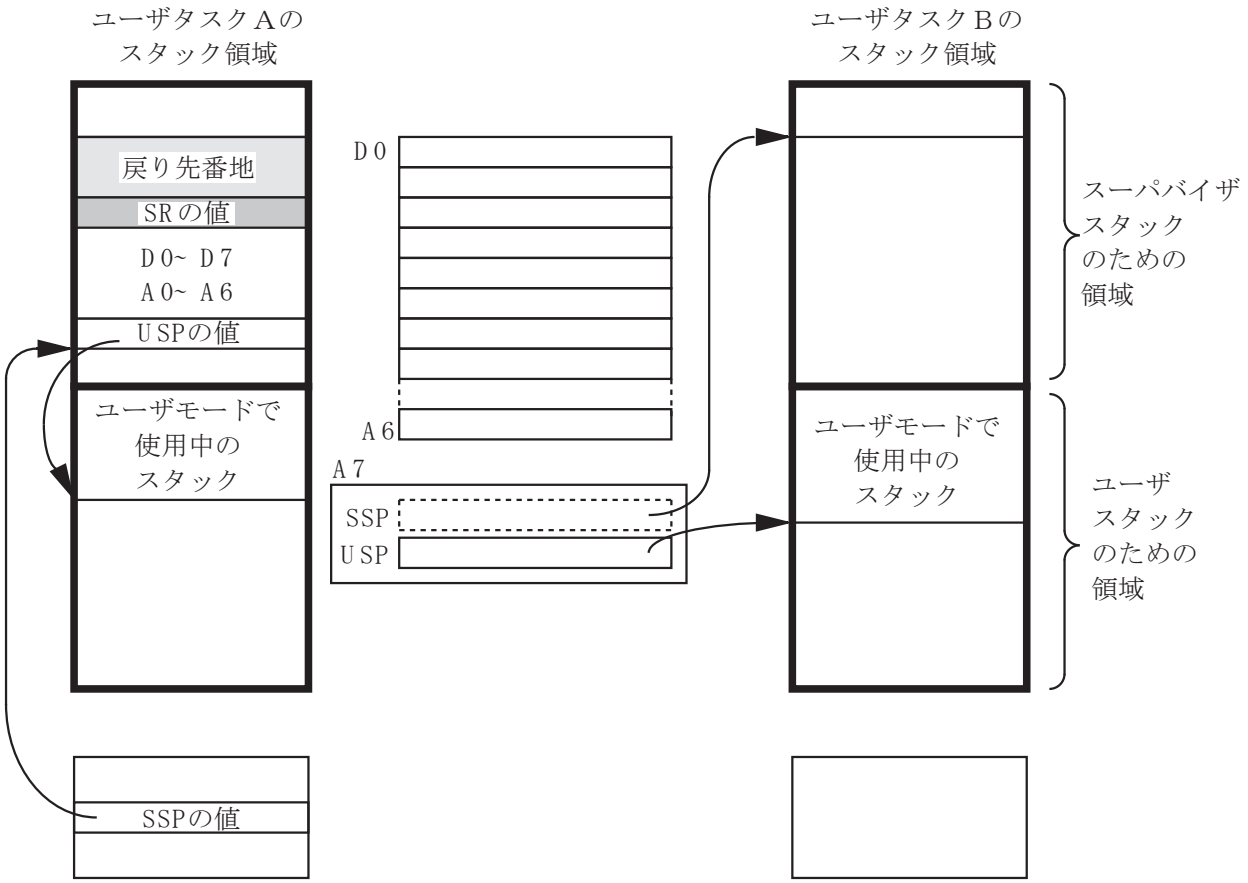

図 2.4: タスクB実行開始後の様子

そこで，C 言語の持つ分割コンパイルの機能を利用して，以下のようなファイルに分割した形で製作することにする．説明の都合上，実験監督者側で作成したサンプルシステム内での変数・関数名を使用しているが，これは適宜変更してさしつかえない．

マルチタスクカーネル部分のモジュール構成図を図 2.6 に示す．大きく 3 つの部分に分けることができる．図の左から順に，マルチタスク機能を提供する部分，タイマによるタスク切り替えを実現する部分，セマフォを実現する部分からなる．図中の `set_tiemr` と `reset_tiemr` は前期に作成したタイマ制御ルーチンである．

C で作成するもの

1. マルチタスク処理実験用システムメインプログラム（”`test2.c`”）
   関数 `main()` を含む．これがいわゆる「メインプログラム」であり，中では，以下のような関数を起動する．

   - カーネルの初期化：`init_kernel()`;
   - ユーザタスクの初期化と登録：`set_task()`;
   - マルチタスク処理の開始：`begin_sch()`;

```
#include <stdio.h>
   ...
#define MAX 1024
   ...
void task1()
{
   .../* タスク 1 定義 */
}

void task2()
{
   .../* タスク 2 定義 */
}
   ...
void main()
{
   .../* ハードウェア初期化 */
   task1(); /* task1 呼び出し */
   task2(); /* task2 呼び出し */
}
```

```
#include <stdio.h>
#include "mtk_c.h"
   ...
#define MAX 1024
   ...
void task1()
{
   .../* タスク 1 定義 */
}

void task2()
{
   .../* タスク 2 定義 */
}
   ...
void main()
{
   .../* ハードウェア初期化 */
   .../* システム設定 */
   set_task(task1);
   set_task(task2);
   /* マルチタスキング開始 */
   begin_sch();
}
```

図 2.5: 通常の C プログラム (左) とマルチタスク実験用 C プログラム (右)

2. ユーザタスク関数群（"user.c"および"user.h"）
   "user.c"にはユーザタスク task1(), task2() 等を記述する．これらのユーザタスク関数は引数をとらず，値も返さなくてよい．無限ループの形式で作ること．

   ヘッダファイル"user.h"はこれらのユーザタスク関数の"extern" 宣言をし，"test2.c"からインクルードする．今回はユーザタスク関数群を"test2.c"に含めてしまってもよい．その場合は"user.h"は不要である．

   ユーザタスク関数を無限ループにしないことも可能であるが，その場合はユーザタスクの終了処理を行う必要がある．終了処理の内容は，ユーザタスク終了時に USP がスタックの底を指していること，終了したタスクは再度実行する必要がないこと，この 2 点から考えてみるとよい．

3. マルチタスクカーネルのうちの C 言語による部分（"mtk_c.c"，および"mtk_c.h"）
   カーネルを構成する関数群を含む．この中には

   - カーネルの初期化：init_kernel();

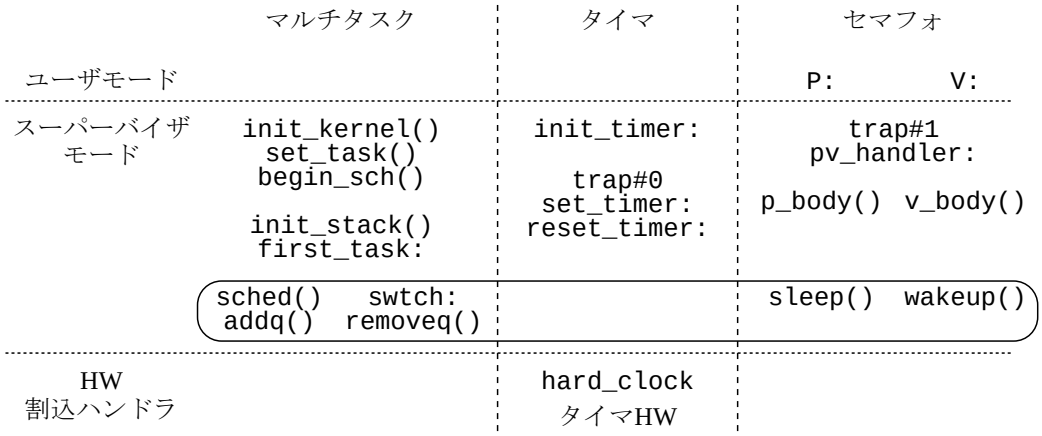

図 2.6: マルチタスクカーネル部モジュール構成図

- ユーザタスクの初期化と登録：set_task();
- ユーザタスク用のスタックの初期化: init_stack();
- マルチタスク処理の開始：begin_sch();
- タスクのキューの最後尾への TCB の追加：addq();
- タスクキューの先頭からの TCB の除去：removeq();
- タスクのスケジュール関数: sched();
- タスクを休眠状態にしてタスクスイッチをする: sleep();
- タスクを目覚めさせ実行可能状態にする: wakeup();

のような関数がある．また，アセンブリルーチンから呼ばれる関数

- P システムコール処理の本体：p_body();
- V システムコール処理の本体：v_body();

がある．

ヘッダファイル”mtk_c.h”は，”mtk_c.c”および”test2.c” の外部で定義される関数の”extern” 宣言，および，カーネル関連の大域変数の宣言を含んでいる．インクルードするファイルがどちらであるかによって，記憶割り当てを行う宣言と外部宣言とを使い分ける方法として，#ifdef〜#else〜#endif を用いる方法がある．詳細は C の参考書を調べること．

アセンブリ言語で作成するもの

マルチタスクカーネルのうちのアセンブリ言語による部分（”mtk_asm.s”）で，カーネルを構成する以下の５つのアセンブリルーチンを含む．

1. first_task
   最初のタスクの起動…C から呼ばれる関数（サブルーチン）として作成．

2. pv_handler
   タスクの切り換え…割り込み処理ルーチンとして作成．関数として呼び出されることはあってはならないが，C プログラムから関数として見えるようにしておくと，C プログラム内でこの関数の名前をこのルーチンの先頭番地として参照することができるので，例外ベクタに登録するのが容易である．

3. P
   P システムコールの入口…C から呼ばれる関数（サブルーチン）として作成．中では適切な値をレジスタに置き，TRAP #1 命令を実行する．

4. V
   V システムコールの入口…C から呼ばれる関数（サブルーチン）として作成．中では適切な値をレジスタに置き，TRAP #1 命令を実行する．

5. swtch
   タスクスイッチを実際に起こす関数

6. hard_clock
   クロック割り込みのルーチン．モニタのシステムコール TRAP #0 を利用して登録するので，rts で復帰するように書く．

7. init_timer
   クロック割り込みルーチン hard_clock をベクトルテーブルに登録するルーチン．モニタのシステムコール TRAP #0 を利用する．

2.5.2 C で記述されるカーネルの機能

本節では，カーネルのうち C で書かれる部分についてその機能を説明する．説明の都合上，実験監督者側で作成したサンプルシステム内での変数・関数名を使用しているが，これは適宜変更してさしつかえない．

大域変数

テーマ 2 のシステムで必要な大域変数は以下の通りである．

1. 現在実行中のタスクの ID：curr_task
2. 現在登録作業中のタスクの ID：new_task
3. 次に実行するタスクの ID：next_task
4. 実行待ちタスクのキュー：ready (型は TASK_ID_TYPE(後述) で先頭のタスクを指す)
5. セマフォの配列：semaphore[NUMSEMAPHORE]
6. タスクコントロールブロックの配列：task_tab[NUMTASK+1]
7. タスクスタックの配列：stacks[NUMTASK]

タスクの ID の型は C プログラム中では次のように定義し，TASK_ID_TYPE 型で変数の宣言をする．実際の型は int 型である．タスクの ID は，タスクに割り振られる番号で，1, 2, 3, …と登録順に振られる．ID が 1 のタスクは task_tab[1] と stacks[0] を利用する．このように二つの配列 task_tab[] と stacks[] のインデックスがずれている点に注意すること[2].

```
typedef int TASK_ID_TYPE;
```

セマフォは例えば次のように定義すればよい.

```
typedef struct
{
        int             count;
        TASK_ID_TYPE    task_list;
} SEMAPHORE_TYPE;

SEMAPHORE_TYPE  semaphore[NUMSEMAPHORE];
```

本テーマで用いるキューには ready キューと，各セマフォが持っているキューがある．現在実行中のタスクはどのキューにも登録されておらず，タスク切替えの際にいずれかのキューに登録される．また，いずれかのキューに登録されているタスクは，他のキューには登録されていない．したがって，一つのタスクは高々一つのキューにだけ登録されている．また，これまで『キューにタスクを登録する』という表現をしたが，タスクの情報は TCB が保持しているので，TCB 自体をキューに登録しても構わない．以上のことを考慮すると，ready キューと各セマフォが持っているキューを以下のように定義される配列 task_tab[](配列要素は一つの TCB に相当) を用いて実現できることがわかる.

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

TCB_TYPE        task_tab[NUMTASK+1];
```

後述するように関数 set_task() を用いてユーザタスクを登録する．登録された順番に従って，タスクには 1, 2, 3, … と ID が付与される．ID が id のユーザタスクに関する情報は task_tab[id] に記憶される．ここで，メンバ next は，キュー中の次のタスクの ID を示す．たとえば，セマフォが 3 つあるとして，各変数の値が次のようになっているとする.

```
ready=3
```

[2]タスク ID を 0, 1, 2, … とせずに 1, 2, 3, … としているのは，Linux，UNIX などの OS のプロセス番号が 1 からはじまることに似せているためで，0 からはじめても間違いではない.

```
semaphore[0].task_list=1
semaphore[1].task_list=4
semaphore[2].task_list=0

task_tab[1].next=7
task_tab[2].next=5
task_tab[3].next=2
task_tab[4].next=0
task_tab[5].next=0
task_tab[6].next=0
task_tab[7].next=0
```

このとき，これらの変数で以下のキュー (左が先頭で右が末尾) を表していることになる．また，キューは図 2.7 のように表せる．

```
ready :  3, 2, 5
semaphore[0] が持っているキュー :  1, 7
semaphore[1] が持っているキュー :  4
semaphore[2] が持っているキュー :  空
```

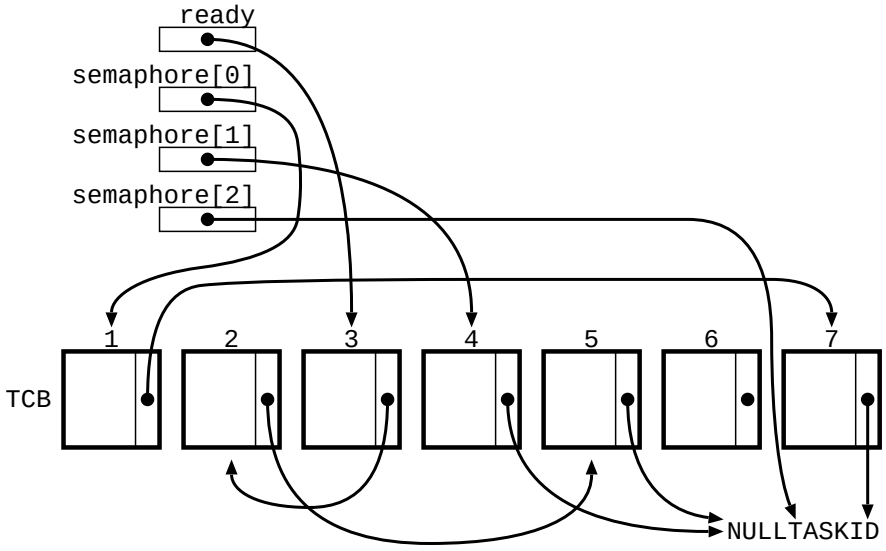

図 2.7: TCB とキュー

上記でわかるように，ID 0 は，キューの終端 (ready や セマフォの task_list の値が 0 のときは，キューが空) であることを表すために用いている．

上記の状態で，たとえば，semaphore[0] キューの末尾に ID が 6 のタスクを登録する場合，(必要があれば task_tab[6] の各要素の値を設定した後) そのキューの末尾を探し (semaphore[0].list の値 1 を見て，task_tab[1].next の値 7 を見て，task_tab[7].next の値 0 を見て，task_tab[7] が末尾であることがわかる)，task_tab[7].next を 6 に変更する．また，semaphore[2] キュー

の末尾に ID が 6 のタスクを登録する場合は，(必要があれば task_tab[6] の各要素の値を設定した後) 同様にこのキューの末尾を探すのであるが，semaphore[2].list の値が 0 であることから，このキューは空であることがわかり，単に，semaphore[2].list の値を 6 とする．どちらの場合も最後に task_tab[6].next の値を 0 としてこのタスクがキューの末尾であることを示す．

キューが空であることを示す ID 0 は，プログラムを見やすくする目的から，以下のように"mtk_c.h"の中で宣言し，プログラム中では 0 ではなく，NULLTASKID を用いる．

```
#define NULLTASKID   0  /* キューの終端 */
#define NUMTASK      5  /* 最大タスク数 */
```

許容できるタスクの最大数を意味する NUMTASK も同様である．今回の実験では，NUMTASK は 5 くらいで十分であろう．前述のように TCB の配列の宣言は，

```
TCB_TYPE        task_tab[NUMTASK+1];
```

としている，配列の大きさが NUMTASK より 1 大きいのは，タスクの ID が 1 から始まり，ID が id のタスク用の TCB を task_tab[id] としているからである．

priority や status はタスクの優先度や TCB の使用状態（未定義・使用中・実行終了など）を表す．優先度はテーマ 2 の実験では直接必要はないが，あとででスケジューリングアルゴリズムを改良する場合には使用する可能性がある．タスクの状態は適切な記号定数（例えば，UNDEFINED など）を"mtk_c.h"の中で宣言して使用すると見やすいプログラムになる．

タスクスタックは次のように定義すればよい．

```
typedef struct
{
        char    ustack[STKSIZE];
        char    sstack[STKSIZE];
} STACK_TYPE;

STACK_TYPE      stacks[NUMTASK];
```

STKSIZE は"mtk_c.h"の中で宣言しておくとよい．今回の実験では，１Ｋバイトほどで十分であろう．

カーネルの初期化 init_kernel()

引数なし．以下のような処理を行う．

1. TCB 配列の初期化：すべて空タスクとする
2. ready キューの初期化：空（タスク ID=0）とする
3. P・V システムコールの割り込み処理ルーチン (pv_handler) を TRAP #1 の割り込みベクタに登録する
4. セマフォの値を初期化する

ユーザタスクの初期化と登録 set_task()

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）を取る．以下のような処理を行う．

1. タスク ID の決定：
   `task_tab[]` の中に空きスロットを見つけ (0 番は除く)，その ID を `new_task` に代入する．
2. TCB の更新：
   上で見つけた TCB に，`task_addr`，`status` を登録する．
3. スタックの初期化：
   関数 `init_stack()` を起動する．関数 `init_stack()` の戻り値を TCB の `stack_ptr` に登録する．
4. キューへの登録：
   `ready` キューに `new_task` を登録する．

C では，配列の名前はその配列のアドレスを意味する．TCB へ登録するスタックの位置情報はこれらの機能を用いて表すことができる．

ユーザタスク用のスタックの初期化 init_stack()

タスク ID を引数としてとる．戻り値に初期化が完了した時点でのユーザタスク用 SSP が指すアドレス (`void *` 型) を返す．引数を id とすると，以下の処理を行なう．

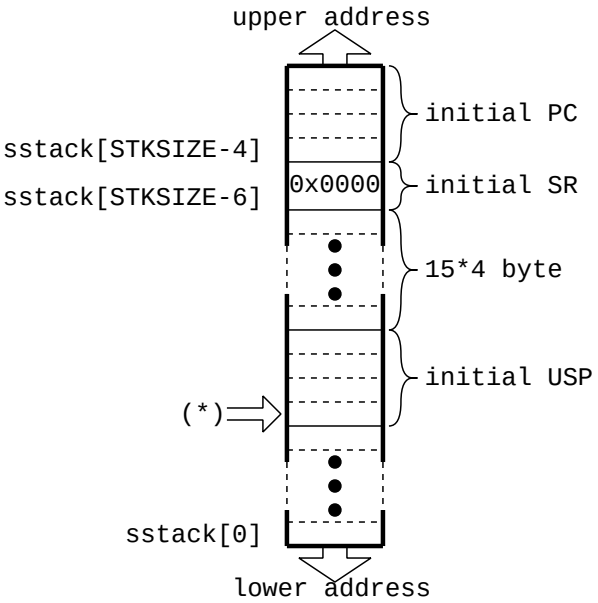

図 2.8: スタックの初期化

1. stacks[id - 1] の sstack を図 2.8 のように設定する．図中の「initial(初期)PC」の部分にはタスクの実行開始アドレス task_tab[id].task_addr を設定する.「initial SR」の部分には 0x0000 を，15×4 バイト分の領域を飛ばして,「initial USP」の部分はユーザスタックトップ stacks[id - 1].ustack[STKSIZE] を設定する.

2. 図 2.8 の (*) のアドレスを戻り値として返す.

なお，int 型へのポインタ ssp を宣言しておくと，ssp の値が現在のスーパバイザスタックのトップを指しているならば，4 バイトの値をプッシュすることは，*(--ssp) = 値で実現できる．これを利用すると，上記の操作は簡単である．また，2 バイトの値をプッシュするときは，unsigned short int 型へのポインタを宣言しておき，これを利用すると良い.

マルチタスク処理の開始 begin_sch()

引数なし．以下のような処理を行う.

1. 最初のタスクの決定：
   キュー ready から removeq() によってタスクを 1 つ取り出し，curr_task に代入する

2. タイマの設定:
   関数 init_timer() を呼び出し，タスクスイッチを行うためのタイマを設定.

3. 最初のタスクの起動：
   関数 first_task() を起動して最初のタスクに制御を移す．これは m68k-elf-gcc コンパイラによってアセンブリ言語サブルーチン first_task の呼び出しに変換される.

タスクのキューの最後尾への TCB の追加 addq()

引数にキューへのポインタとタスクの ID を取り，その TCB をキューの最後尾に登録する.

タスクのキューの先頭からの TCB の除去 removeq()

引数にキューへのポインタを取り，先頭のタスクをキューから取り除いてその ID を返す.

P システムコールの本体 p_body()

P 命令のうちの 2.2.2 節で述べたセマフォ値とタスクリストの操作を行う関数.

1. セマフォの値を減らす

2. セマフォが獲得できなければ sleep(セマフォの ID) を実行して，休眠状態に入る.

V システムコールの本体 v_body()

V 命令のうちの 2.2.2 節で述べたセマフォ値とタスクリストの操作を行う関数.

1. セマフォの値を増やす
2. セマフォが空けば，wakeup(セマフォの ID) を実行して，そのセマフォを待っているタスクを一つ，実行可能状態にする.

タスクのスケジュール関数 sched()

タスクのスケジュール関数 (スケジューラ) は ready 状態のタスクから，ready 状態になった順序，タスクの優先度により次に run 状態にするタスクを決める関数である．今回はタスクの優先順位を考慮していないため，単に ready キューの先頭のタスク ID を取り出し，next_task にセットする単純なものでよいものとする．ready キューの操作は removeq を用いる.
ただし、取り出した next_task が NULLTASKID の場合は、無限ループに入るようにすること.

タスクを休眠状態にしてタスクスイッチをする sleep(ch)

1. チャンネル ch (ここではセマフォだけが休眠状態に陥る原因なので，セマフォ ID としていい) の待ち行列に現タスクをつなぐ.
2. スケジューラ sched を起動して，次に実行するタスクの ID を next_task にセットする.
3. タスクスイッチ関数 swtch(後述) を呼び出し，タスクを切り替える.

休眠状態のタスクを実行可能状態にする wakeup(ch)

チャンネル ch (ここではセマフォだけが休眠状態に陥る原因なので，セマフォ ID としていい) の待ち行列ににつながっている 1 つのタスクを実行可能状態の行列 (ready) につなぎなおす．ここでは，タスクスイッチは起こらない.

2.5.3 アセンブリ言語で記述されるカーネルの機能

本節では，カーネルのうちアセンブリ言語で書かれる部分についてその機能を説明する．説明の都合上，サンプルシステム内でのサブルーチンラベル名を使用しているが，これは適宜変更しても構わない.

ユーザタスク起動サブルーチン first_task

このルーチンは，カーネルが使用しているスタックを curr_task で指されるタスクのそれに切り換えてマルチタスク処理を開始するためのサブルーチンで，C の begin_sch() の中で 1 度だけ起動される．RTE 命令で終わるため，呼び出し側には戻らない．起動される時点ではスーパバイザモードであることが必要である.

1. TCB 先頭番地の計算：
   curr_task の TCB のアドレスを見つける．

2. USP，SSP の値の回復：
   このタスクの TCB に記録されている SSP の値およびスーパバイザスタックに記録されている USP の値を回復する．

3. 残りの全レジスタの回復：
   スーパーバイザスタックに積まれている残り 15 本のレジスタの値を回復する．

4. ユーザタスクの起動：
   RTE 命令を実行する．

P システムコールの入口 P

このサブルーチンは C プログラムから引数（セマフォID）付きで呼び出されるので，スタックの扱いには 1.2.2 で述べたような注意が必要である．

P システムコールの ID（0 とする）を D0 レジスタに，スタックから取り出した引数を（セマフォID）D1 レジスタに，それぞれセットして `TRAP #1` 命令を実行する．終了後は `RTS` する．

V システムコールの入口 V

このサブルーチンは C プログラムから引数（セマフォID）付きで呼び出されるので，スタックの扱いには 1.2.2 で述べたような注意が必要である．

V システムコールの ID（1 とする）を D0 レジスタに，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，それぞれセットして `TRAP #1` 命令を実行する．終了後は `RTS` する．

TRAP #1 割り込み処理ルーチン pv_handler

このルーチンはアセンブラルーチンの `P`，`V` 中の `TRAP #1` 命令が実行されたときの，割り込み処理ルーチンであり，スーパバイザモードで実行される．このルーチンはプログラムから `JSR` 命令あるいは C 関数 `pv_handler()` の形で呼び出してはならない．

このルーチンは，D0 レジスタに P，V システムコールの種類を，D1 レジスタにセマフォID がセットされて呼び出される．D0 の値に応じて C 関数 `p_body()` あるいは `v_body()` を呼び出す．両関数とも引数にセマフォID (割り込み時の D1 が保持) を取るので，これをスタックに積んだ後サブルーチンコールをする必要がある．タスクスイッチは `p_body()` 中で起こる可能性がある．

以下に処理の概要を示す．

1. 実行中のタスクのレジスタの退避：
   このルーチン内で使用するレジスタをスーパバイザスタックに積む．退避し忘れたレジスタがあると，予期せぬエラーとなるので注意すること．

2. 割り込み禁止:
   SR の値をスタックに退避した後，走行レベルを 7 にするため，SR に 0x2700 を書き込む．

3. C の関数 p_body() もしくは v_body() の呼び出し：
   割り込み時の D0，D1 の値を参照して，D1 を引数 (セマフォID) としてスタックに積む．D0 が 0 なら p_body() を，1 なら v_body() を呼び出す．タスクスイッチは p_vody() の中で起こる可能性がある．

4. 割り込み禁止の解除:
   SR の値をスタックから復帰する．

5. レジスタの復帰:
   最初に退避したレジスタをスタックから復帰する．

6. 割り込み処理を終了：
   RTE を実行する．

タスクスイッチ関数 swtch

このルーチンは，sleep と，タイマ割り込みルーチン hard_clock (後述) の二通りの方法で呼び出される．

- sleep から呼び出される場合:
  ユーザタスク，P 命令，TRAP 命令，pv_handler，sleep，swtch の順に呼び出される．

- hard_clock から呼び出される場合:
  ユーザタスク，タイマ割り込み，hard_clock，swtch の順に呼び出される．

一方，次に実行されるタスクは sleep と hard_clock で中断されたタスクに加えて，次のタスクが含まれる．

- set_task 以降，初めて実行されるタスク．

swtch は通常の C 関数のように JSR で呼び出されるが，最後に復帰する (= タスク切り替えが起こる) ときは RTE を実行する．これは，次に実行させるべきタスクが初めて実行されるタスクでも，既に実行を始めたタスクでも同様に扱うための処置である．既に実行を始めたタスクは RTS で呼び出し元 (sleep，hard_clock) に戻ってかまわないが，タスクが最初の起動の場合は first_task と同様に RTE で swtch を終了する必要がある．

そこで swtch の先頭で，呼び出された直後の SR の値をスタックに積んでおく．これにより，これまで実行したことがあるタスクも RTE で swtch の呼び出し元に戻ることが可能になる．つまり，タスクスイッチ後のタスクが今まで実行したことがあるタスクでも初めてのタスクでも，そのタスク用のスーパバイザスタックの上部は順に、USP，D0〜D7/A0〜A6，SR，戻り番地 (実行が初めてのタスクはタスク開始番地、その他は，sleep または hard_clock 内の番地) となり，USP，D0〜D7/A0〜A6 を順に回復し，RTE で終了すれば良いことになる．

以下の手順の 3 までが現在実行中のタスクの中断処理で，それ以降は次に実行するタスクの再開処理である．

1. SR をスタックに積んで，RTE で復帰できるようにする．

2. 実行中のタスクのレジスタの退避：
   D0〜D7，A0〜A6，USP をタスクのスーパバイザスタックに積む．

3. SSP の保存:
   このタスクの TCB の位置を求め， SSP を正しい位置に記録する．

4. curr_task を変更:
   curr_task に next_task を代入する．swtch の呼び出し前にスケジューラ sched を起動しているため，next_task には次に実行するタスク ID がセットされている．

5. 次のタスクの SSP の読み出し：
   新たな curr_task の値を元に TCB の位置を割り出して，その中に記録されている SSP の値を回復する．これにより，スーパバイザスタックが次のタスクのものへ切り換わる．

6. 次のタスクのレジスタの読み出し：
   切り換わったスーパバイザスタックから USP，D0〜D7，A0〜A6 の値を回復する．

7. タスク切り替えをおこす:
   RTE を実行する．

タイマ割り込みルーチン hard_clock

このルーチンは割り込み駆動であるが，前期作成のタイマ用ハードウェア割込み処理インタフェースから呼び出されるため，RTS で復帰する．

1. 実行中のタスクのレジスタの退避：
   このルーチン内で使用するレジスタをスーパバイザスタックに積む．タイマ割り込みで実行されるルーチンであるため，退避し忘れたレジスタがあると予期せぬエラーとなり非常にデバッグが困難である．十分に注意すること．

2. addq() により，curr_task を ready の末尾に追加．

3. sched を起動 (次に実行されるタスクの ID が next_task にセットされる)．

4. swtch を起動．

5. レジスタの復帰:
   最初に退避したレジスタをスタックから復帰する．

クロック割り込みルーチン init_timer

前期に作成したタイマ制御ルーチンを用いて，タイマによるハードウェア割り込みが発生するように作成する．割り込み周期は任意である．参考値として，Linux などでは 10ms 周期である．タイマによるタスク切替えの動作確認時は 1 秒程度の人が知覚できる周期が適当だと考える．

2.5.4 コンパイルとテスト

1.4.3 節と同様にコンパイルには make コマンドを用いる．make test2 コマンドによって，mon.s, crt0.s, csys68k.c, inchrw.s, mtk_asm.s, mtk_c.c, outchr.s, test2.c ファイルがそれぞれコンパイルされて，test2.abs が生成される．

make test3 コマンドは make test2 とほぼ同様で，異なるのは test2.c の代わりに test3.c がコンパイルされること，test3.abs が生成されることの 2 点である．

make test2 を実行した際の出力例を以下に示す．

```
guest@pcs0xxx% make test2
make  LIB_JIKKEN=...  -f Makefile.2
make[1]:  Entering directory ...
m68k-elf-as -m68000  -ahls=crt0.glis -o crt0.o crt0.s
m68k-elf-as -m68000  -ahls=mon.glis -o mon.o mon.s
m68k-elf-gcc -c -g -O2 -Wall -m68000 -msoft-float -I.  -Wa,-ahls=csys68k.glis
    -o csys68k.o csys68k.c
m68k-elf-as -m68000  -ahls=inchrw.glis -o inchrw.o inchrw.s
m68k-elf-as -m68000  -ahls=mtk_asm.glis -o mtk_asm.o mtk_asm.s
m68k-elf-gcc -c -g -O2 -Wall -m68000 -msoft-float -I.  -Wa,-ahls=mtk_c.glis
    -o mtk_c.o mtk_c.c
m68k-elf-as -m68000  -ahls=outchr.glis -o outchr.o outchr.s
m68k-elf-gcc -c -g -O2 -Wall -m68000 -msoft-float -I.  -Wa,-ahls=test2.glis
    -o test2.o test2.c
m68k-elf-ld -nostdlib --cref crt0.o mon.o csys68k.o inchrw.o mtk_asm.o mtk_c.o
    outchr.o test2.o -o test2.hex -Map test2.map -T .../ldscript.cmd
perl .../sconv.pl test2.hex > test2.abs
python .../gal.py test2.map crt0.glis mon.glis csys68k.glis inchrw.glis mtk_asm.glis
    mtk_c.glis outchr.glis test2.glis > /dev/null
make[1]:  Leaving directory ...
```

マルチタスクカーネルがうまく動作しているかどうかを確認するために，簡単なユーザタスク 2 つ以上を交互に切り替えながら実行する．P 命令によってユーザタスクの実行を中断するためには，その引数のセマフォの値が 0 以下になっていなければならない．また，そのタスクがあとで実行可能になるためには，少なくとも，休止中のタスクがつながっているセマフォに対して 1 回は V 命令が発行されなければならない．テーマ 1 のレポート時に作成した思考実験用のユーザタスク，またはより簡易で動作テストが十分可能なユーザタスクを別途作成して動作テストを行うこと．

デバッグは次のステップで行うと効率が良いと思われる．

D1:タイマ割り込み OFF の状態で最初のタスクが動作するか
　タスク登録 (保存) と最初のタスク復帰でのスタック構造の一致

D2:hard_clock 先は RTS のみ、タイマ割り込みを ON にした状態で最初のタスクが動作するか
　タイマ割り込み処理が及ぼす影響、モニタ差し替えの検討

D3:hard_clock 先を有効にし、タイマ割り込みを ON にした状態で 2 巡目のタスク動作が起こるか
　swtch の実装、curr_task の保存状況、ready キュー操作

D4:マルチタスク環境下でセマフォ表示タスクを動作させながらセマフォの動作確認
　P, V, trap #1 動作状況、セマフォキュー操作

D5:セマフォによるタスク切り替え
　タスク順序制御としてのセマフォ利用

2.5.5 注意事項，その他

テーマ 2 における注意事項等を列挙する．必要があれば参照の事.

1. 割り込み処理中の走行レベルに注意.
2. begin_sch() と swtch: のタスク復元手順は同じ.
3. キュー操作関数を用いず独自にキュー操作を行うと，キュー構造を破壊する元になる.
4. 動作中のタスク以外は何らかのキューに入る．キュー構造からタスクを漏らさないこと.
5. moodle の SW フォルダにソフトウェア実験関連ファイルがあり，そこに 68000 Instruction set (PDF) がある．USP の操作に関連する命令は，PDF 内検索で USP を検索する.
6. C プログラムから絶対アドレスにアクセスする方法
   - LED0 への書き込み: *(char *)0x00d00039 = 'A';
   - トグル SW の状態読み: tswstat = *(char *)0x00d00041;

   この方法で割り込みベクタの書き換えも可能．ただし，関数へのポインタや型変換が必要.

第3章　テーマ３：応用

テーマ１，２が終了したら，これらを用いて例えば以下のようなプログラムの作成を考えよう．

1. ３つ以上のタスクを順に実行する．
2. RS232C ポートを２つ使って，ポート０へ出力するタスクと，ポート１へ出力するタスクとを，マルチタスクで実行する．
3. RS232C ポートを２つ使って，一方のポートからの入力を受け取り何らかの出力を他方のポートへ行うタスクを複数作成し，セマフォにより排他制御しながら実行する．

ここで，複数の RS232C ポートを用いる場合には，テーマ１で移植したライブラリがシリアルポート０専用となっているので，そのままでは使用できない．このため，`inbyte()`，`outbyte()` の改造，および，`csys68k.c` に含まれる関数 `read()`，`write()` などを改造する必要がある．
以下の点に注意して改造を行えば，２つのポートを同じ入出力関数で使い分けることができる．

- ユーザタスクから `read()` と `write()` を呼び出す際に，第１引数 `fd` で入出力するポートを指定することができる．
- RS232C ポート０は UART1，ポート１は UART2 と呼ばれる．どちらもほぼ同様の制御を行えばよい．UART1 に関する詳細は前期ソフトウェア実験のテキストを参照すること．UART1 と UART2 を比較すると，
  - 割り込みマスクレジスタ (IMR)，割り込みステータスレジスタ (ISR)，割り込みペンディングレジスタ (IPR) はいずれも，Bit 2 が UART1 に対応し，Bit 12 が UART2 に対応する．
  - UART2 も UART1 と同じ構成のレジスタを持つ．UART1 のレジスタが 0xFFF900 から始まるのに対して，UART2 のレジスタは 0xFFF910 から始まる．
  - 割り込みベクタ: UART2 はレベル 5(変更可能．MC68VZ328 マニュアル参照)．UART1 はレベル 4(前期テキスト参照)．
- モニタ内にあるキュー (文字入出力用のバッファや制御変数 (キュー位置ポインタ)) は RS232C ポート 0 用なので、ポート 1 用のキューを新たに複製し、指定ポートに応じて切替える。

3.1　RS232C ポートへのストリーム割り当て

テーマ 3 において RS232C ポート 0, ポート 1 を両方使用可能にした場合、テーマ 2 までは read(), write() の関数内で無視していた read(), write() の第 1 引数 fd を利用する事によって、入出力対象の RS232C ポートを指定する事が出来る。その代わりに、テーマ 2 までは使用可能だったスト

リーム入出力関数 scanf(), printf() が使えなくなり、低レベル関数 read(),write() を使わざるを得なくなる事態に陥っている。ここでは fd をより適切に使用し、ストリーム割当を行なう事でストリーム入出力関数を使用可能にする。

3.1.1 ファイルディスクリプタとファイルポインタ

read(), write() の第 1 引数 fd は、ファイルディスクリプタ (file descriptor) またはファイルハンドル (file handle) と呼び、ファイルやデバイス等を open() で開いた際に入出力操作の識別子として渡される非負数である。C ライブラリにおいて、ファイルディスクリプタの 0,1,2 はそれぞれ標準入力、標準出力、標準エラー出力として予め割り当てられている。しかしこの割り当ては必ずしも堅持する必要はない。

また、ファイルやデバイス等を fopen() で開いた際に渡される FILE 構造体へのポインタはファイルポインタと呼ぶ。stdin, stdout, stderr は標準入力、標準出力、標準エラー出力に対して予め割り当てられているファイルポインタである。FILE 構造体には入出力ストリームを制御するパラメータが格納されており、ファイルディスクリプタも FILE 構造体内に格納されている。

scanf() はファイルポインタ stdin に対する入力操作、printf() はファイルポインタ stdout に対する出力操作が行なわれ、最終的に scanf() はファイルディスクリプタ 0 に対する read()、printf() はファイルディスクリプタ 1 に対する write() に展開される。

標準入出力以外のファイルディスクリプタは open() によってカーネルから割り当てられ、open() 時のフラグも管理すべきものであるが、実験用カーネルに open() は実装されていない。従って、動作可能な範囲内で簡略化された実装を行なう。

一例として **UART1 へのファイルディスクリプタを 3,UART2 へのファイルディスクリプタを 4 に割り当て、入出力ともに同じファイルディスクリプタを用いる事とする。**この場合、標準入出力指定とは別に UART ポートを指定した入出力が可能となる。

3.1.2 fdopen() の利用と fcntl() の実装

ファイルディスクリプタで入出力できる対象 (RS232C ポート) を scanf()/printf() のストリーム入出力関数でアクセス出来るようにする為には、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必要がある。fdopen() はこの目的の為の C ライブラリ関数である。

```
FILE *fdopen(int fd, const char *mode);
```

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である。読み込み用の場合は"r", 書き込み用の場合は"w" を *mode* として指定する。入出力ストリームの割当が正常に行なわれた場合、fdopen() は NULL 以外の FILE 構造体へのポインタを返す。

RS232C 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として、以下の大域変数を準備する。

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

fdopen(fd,mode) は内部で低レベル関数 fcntl(fd,F_GETFL) を呼び出し、ファイルディスクリプタ fd の入出力属性を確認する。得られた属性が fdopen() の引数で指定された入出力モードを許

容できれば正常、許容できなければ fcntl() の EBADF 戻り値を返して異常終了する。EBADF は errno.h で定義される。fdopen() は C ライブラリ内に実装されているが、低レベル関数 fcntl() は実装されていないので、新たに作成する必要がある。

fcntl() は以下の型を取る。

```
#include <stdarg.h>
#include <fcntl.h>
int fcntl(int fd, int cmd, ...);
```

fcntl() は実装上引数が可変なので stdarg.h をインクルードする必要があるが、*cmd*=F_GETFL の場合は 2 引数 (*fd* と *cmd*) しか取らない。fcntl.h では fcntl() 内で用いる数値 (F_GETFL, O_RDWR 等) が定義されている。

fcntl() 内は、*cmd* の値が F_GETFL の場合のみ簡略的な実装を行なう。本来ならば *fd* に対する open() 時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ので、常に O_RDWR (入出力可能) を返すようにする。*cmd* の値が F_GETFL 以外の場合、単に 0 を返す様にしておく。

3.1.3 ファイルディスクリプタから実デバイスへのマッピング

ファイルディスクリプタを介した入出力は、下層において RS232C ポート番号にマッピングする必要がある。テーマ 2 までは、全ての入出力指定を無条件に RS232C ポート番号 0 にマッピングした場合に相当する。今回の例の場合、**ファイルディスクリプタ 0,1,2,3 を RS232C ポート 0 に、ファイルディスクリプタ 4 を RS232C ポート 1 にマッピングする。**

実デバイスへのマッピングを行なう場所として、read()/write() 関数内を採用する。inbyte()/outbyte() 関数内も考えられるが、これらの関数はエラーを返す事が考慮されていない。また、read()/write() 関数内の方がデバイスマッピング判定頻度を減らせる事に加え、マッピング操作を C 言語で記述する事が出来る。

読み書き可能性を考慮したデバイスマッピング判定は以下の通りとなる。read() 内でのデバイスマッピング判定は、**ファイルディスクリプタ 0,3 を RS232C ポート 0 に、ファイルディスクリプタ 4 を RS232C ポート 1 にマッピングし、他はエラー EBADF にする。**write() 内でのデバイスマッピング判定は、**ファイルディスクリプタ 1,2,3 を RS232C ポート 0 に、ファイルディスクリプタ 4 を RS232C ポート 1 にマッピングし、他はエラー EBADF にする。**

3.1.4 作業のまとめ

上記の仕様を満たす fcntl() を実装する。更に、fdopen() を用いてファイルディスクリプタにストリームを割り当てる関数を作成し、main() の比較的早い段階で呼び出す。read()/write() 関数内でのデバイスマッピングの実装により、標準入出力に対する入出力操作は RS232C ポート 0 に振り分けられるので、scanf(),printf() はテーマ 2 までと同様に使用できる。RS232C ポートを指定した入出力は、ファイルポインタを指定する入出力関数 fscanf(),fprintf() を用いる事によって scanf(),printf() と同等の操作が行なえる。

3.2 USB-シリアルアダプタを介したシリアル接続

現在 RS232C によるシリアルポートは時代後れの (=レガシー)I/O ポートとなっており，RS232C を標準搭載した PC は減少している．拡張ドックで RS232C に対応するノート PC もあるが，Linux 環境では拡張ドックがサポートされない場合が多い．この様な状況下でも RS232C によるシリアル通信を行う為に，USB-シリアルアダプタを介したシリアル接続に対応した．以下にその使用方法を示す.

3.2.1 接続，デバイス確認

USB-シリアルアダプタを PC の USB ポートに接続すると，PC の OS(linux) 側で USB デバイスの認識・登録が行われる．その結果は以下の手段等で確認できる．

1. /var/log/messages にログが記録されるので，コマンドライン端末から、“tail /var/log/messages” を実行し，“pl2303 converter now attatched to ttyUSB#” の文字列を確認する．(# は，1 つ目のアダプタでは “0”，2 つ目では “1”)
2. プルダウンメニューから，“システム→管理→システムログ (システムログビューア)” で /var/log/messages の内容を確認．ログ内で確認すべき文字列は 1. と同じ．
3. コマンドライン端末から、“cat /proc/tty/driver/usbserial” を実行し，USB-シリアルデバイス一覧表を得る．表の左端の数字がデバイスファイル /dev/ttyUSB# の # に相当する．
4. コマンドライン端末から、“ls /dev/ttyUSB*” を実行し，デバイスファイル /dev/ttyUSB# の存在を確認する．

上記の確認方法で得られる USB-シリアルデバイス名 ttyUSB の直後の数字 (ttyUSB0 ならば “0”) を確認する．この数字は同一デバイスの認識順であり，USB ポートの順番ではないと思われる．

3.2.2 通信端末ソフトの起動と利用法

前節で確認された USB-シリアルデバイス種類名 ttyUSB とその番号を用いて，通信端末ソフトを起動する．デバイス名が **ttyUSB0** ならば “**m68k-termUSB0**”, **ttyUSB1** ならば “**m68k-termUSB1**” を実行する．使用方法は，シリアルデバイスが異なる以外は **m68k-term** (デバイス名は **ttyS0**) と同一である．

即ち，USB-シリアルアダプタでシリアルポートを 2 つに増やして実機の COM1,COM2 に接続し，2 つのコマンドライン端末で異なるシリアルデバイスに対応する通信端末ソフトを起動すれば，1 つの PC で 2 つのシリアル入出力に対応できる様になる．ただし入力はマウスで focus されたコマンドライン端末の側しか行えないため，2 つのシリアルポートへの同時入力を行うためには 2 台の PC を実機に接続する方法が推奨される．

付 録A　例外エラーメッセージと対応

1. 詳細な情報が表示される例外

   Bus Error bus time-out 判定
   Read-only/Write-protected の領域に書き込もうとした.
   User mode で Supervisor-only の領域にアクセスしようとした.
   ROM,RAM,I/O 等が未実装のアドレス空間にアクセスしようとした.

   Address Error アドレスエラー
   奇数番地のメモリアドレスに .b 以外でアクセスしようとした.

   これらの例外については，別途 Access Info (読み/書き等), Access Address (アクセスしようとしたアドレス), IR, SR, PC (それぞれ例外発生時の Instruction Register, Status Register, Program Counter の値) が IPL によって追加表示されるので，デバッグの支援情報となる．原因が見当たらない場合，3. の可能性もある．

2. 稀に起こりうる例外

   Divide-by-Zero DIVS, DIVU 命令で 0 で割る操作を行った.

   Privilege violation User mode で特権命令を実行しようとした.

   Spurious Interrupt 割り込み処理中に Bus Error が生じた.

   CHK instruction CHK 命令でテスト数値が指定範囲を越えた.

   TRAPV instruction TRAPV 命令でオーバーフローを検出した.

   Trace SR の T-bit を 1 にしてトレースモードに切り替わった.

   Line 1111 emulation FPU が無いのに浮動小数点命令を実行しようとした.

   Spurious Interrupt 以外は，特定の命令や操作に起因している．実際に対応した命令・操作をプログラムしているならば例外検出による正常なプログラム停止なので，そのまま継続するか，発生原因の命令・操作を止める．プログラムした覚えが無い場合は，3. へ移る．

   Spurious Interrupt の場合，割り込み処理中の Bus error まで判明している．同時に **Bus Error** が生じた場合はそれも参照してみる．該当部が無さそうならば，3. へ移る．

3. 通常起こらない例外

   Illegal Instruction 不正な命令

   Line 1010 emulation '0xA' の疑似命令テーブルを用意していない

   本来プログラムしていない命令が実行されている．スタックの収支の不一致を主な原因として PC にプログラムの存在しないアドレスが格納され，暴走したものと考えられる．