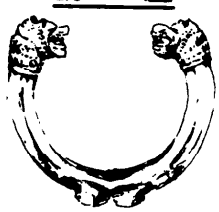
講　座



OS/2 における並行プログラムの作り方 (II)†

鷹　野　澄††

2. OS/2 カーネル上のマルチスレッド・プログラミング

今回は，プロセスの中で複数のスレッドを生成し並行処理するマルチスレッド・プログラムの作り方について述べる．マルチスレッド・プログラムは，プロセスを複数生成して並行処理するマルチプロセス・プログラムに比べて，スレッド間の切り替えや同期，データ転送などのオーバヘッドが少ないため，迅速な応答を必要とする並行プログラムの作成に適している．しかし一方，マルチスレッド・プログラムでは，プログラムコードやデータが共有されるため，アクセスの競合が発生しやすいというやっかいな問題もある．

2.1 OS/2 におけるスレッドの生成と終了

新たな並行処理を生成および終了する方法としては，Conway などが提唱した図-1 の fork(分岐)命令と join（結合）命令や，Dijkstra が提唱した図-2 の parbegin-parend 並行文などがよく知られている．OS/2 の場合は，図-3 に示したように，スレッドの生成は fork 命令に似た DosCreate Thread という API で行い，スレッドの終了はスレッド自身が DosExit という API を実行して終了するという方法が採用されている．join 命令のようなスレッドの結合は，スレッド間の同期とスレッドの終了という二つの処理により実現できるので，OS/2 が提供しているのは join 命令よりもさらにプリミティブなものである．

図-4 は，OS/2 で複数のスレッドを生成する簡単なプログラムの例である．この中の makechildth ルーチンに，スレッドの生成のための処理をまとめて示した．スレッドの生成では各スレッドに別別に専用のスタック領域を割り当て，それらが他のスレッドからアクセスされることのないように注意する必要がある．またスタック領域の大きさは，スレッド自身が使う分にさらに最低 2KB 以上余分に加えて十分大きくとる必要がある．ここでは C のランタイムルーチンの malloc により 3KB のスタック領域を動的に確保している．

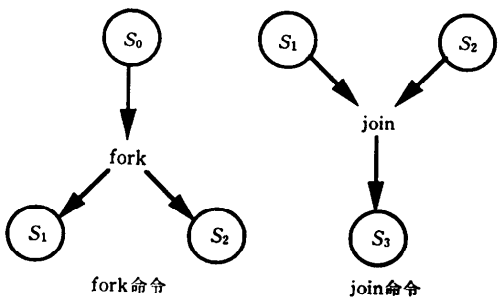
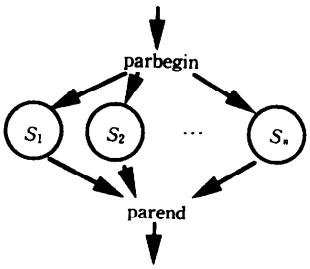


図-1 fork 命令と join 命令



parbegin S_1; S_2; …; S_n parend

図-2 parbegin-parend 並行文

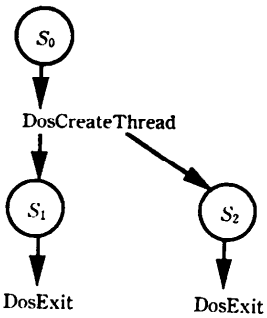


図-3 OS/2 のスレッドの生成と終了

† Concurrent Programming in OS/2 (II) by Kiyoshi TAKANO (Earthquake Research Institute, University of Tokyo).

†† 東京大学地震研究所

```
/*--- maketh. c ----------------------------------*/
/* 複数のスレッドを生成するプログラムの例          */
/* [MS-C による作成例] cl maketh. c maketh. def  */
/* [モジュール定義ファイル maketh. def の例]      */
/*    NAME   MAKETH   WINDOWCOMPAT                */
/*    PROTMODE                                    */
/*    STACKSIZE 4096                              */
/* [使用例] maketh 5 (スレッドを5個生成)          */
/*------------------------------------------------*/
#include <stdio. h>
#include <stdlib. h>     /*    malloc            */
#include "sample2. h"    /* 例題用ヘッダ(図-8 参照)*/
              /*- スレッド間の共通変数 -----------*/
USHORT GO = FALSE;  /* 開始フラグ                 */
int    nchild = 0;  /* 生成したスレッドの数       */
ULONG  mutex = 0;   /* 排他制御用のセマフォ       */
              /*- 関数のプロトタイプ宣言 ---------*/
makechildth(PFN);        /* スレッド生成ルーチン  */
void far child(void);    /* スレッド処理ルーチン  */
/*==== メインプログラム (スレッド1) ==============*/
main(argc, argv) int argc; char **argv; {
    USHORT rc; int nt, i;
    if(argc!=2)          /* 引数をチェック        */
       printf("Usage: maketh number_of_child¥n");
    else {
       nt = atoi(argv[1]); /* nt=スレッドの数     */
       for(i=0; i<nt; i++) makechildth(child);
       nchild = nt;      /* 生成したスレッドの数  */
       GO = TRUE;        /* 開始フラグを ON       */
       /* すべての子スレッドが終了するまで待つ     */
       while(nchild>0) DosSleep(10L);
       printf("All threads end. ¥n");
    }
}
/*==== makechildth: スレッド生成ルーチン ==========*/
#define STKSIZ 3072      /* スタック領域の大きさ  */
makechildth(PFN entry) {
    USHORT rc; TID tid; char *sb;
            /*-- スレッドのスタック領域を確保 --*/
    if((sb=(char *)malloc(STKSIZ))==NULL)
          err("stack allocation failed.", -1);
    sb+=STKSIZ;      /* sb=スタックの底          */
            /*-- スレッドを生成 ----------------*/
    rc = DosCreateThread(
          entry,     /* スレッドのエントリ名     */
          &tid,      /* スレッド ID (出力)       */
          sb);       /* スタック領域             */
    if(rc) err("DosCreateThead failed.", rc);
    printf("Thread %d created. ¥n", tid);
}
/*==== err: エラーメッセージ出力および強制終了 =====*/
err(msg, rc) char *msg; USHORT rc; {
    printf("ERROR: %s RC = %d¥n", msg, rc);
    DosExit(EXIT_PROCESS, 1);
}
/*==== child: スレッド処理ルーチン (共通) ==========*/
char txt[ ]="Now, child thread running...¥r¥n";
void far child (void) { USHORT nw;
    while(!GO) DosSleep(10L); /*GOになるまで待つ*/
    /* +++ ここにスレッドの処理を書く +++ */
       DosWrite(1, txt, sizeof(txt) -1, &nw);
    DosSemRequest(&mutex, -1L); /*排他制御の開始*/
      nchild = nchild - 1;      /*nchild を -1  */
    DosSemClear(&mutex);        /*排他制御の終了*/
    DosExit(EXIT_THREAD, 0);    /*スレッドの終了*/
}
```

図-4 複数のスレッドを生成するプログラム maketh. c

図-4 の最後の child ルーチンは，生成されたスレッドの処理の例である．ここでは最初に変数 GO が TRUE になるまで待ち，最後のほうでスレッドの数を示す変数 nchild を 1 減らしてから DosExit (0, rc) でスレッドを終了している．これを DosExit (1, rc) とすると，自分自身だけでなくプロセス内のすべてのスレッドを終了させるという意味になる (err ルーチンの最後を参照)．DosExit の 2 番目の引数 rc はプロセス終了コードで，この DosExit でプロセスが終了するときにのみ意味をもつ．なお，OS/2 バージョン 1.1 からスレッド 1 だけ特別になり，スレッド 1 を終了させるとそのプロセス内のすべてのスレッドが強制的に終了するようになった．このため，すべてのスレッドが終了するまでスレッド 1 を終了させないようにする必要がある．図-4 の例では，main ルーチンの最後に示したように，スレッド 1 は実行中の child スレッドの数が 0 になるまでループして待っている．

2.2 スレッド間のアクセスの競合の問題

マルチスレッド・プログラムでは，プログラムやデータなどのメモリやオープンされているファイルなどの資源はすべてプロセスの所有となり，プロセス内のすべてのスレッドから共有される．このため，あるスレッドがデータを参照している途中で別のスレッドがそのデータを書き換えてしまうような，いわゆるアクセスの競合が発生しやすいのが問題である．特に問題となるのは次のようなケースであろう．

(1) 複数のスレッドが共有変数を更新する場合

たとえば，図-4 の child ルーチンの中で，nchild 変数を nchild=nchild−1 という文で更新しているのがこの例である．もしあるスレッドが nchild の値を取り出し −1 し，それを格納する直前に CPU が切り替わって別のスレッドがこの文を実行したとすると，二つのスレッドがこの文を実行したにもかかわらず nchild の値は −1 しかされないという不都合が生じてしまう．

(2) 複数のスレッドがリエントラントでないルーチンを使用する場合

リエントラント (reentrant, 再入可能) なルーチンとは，それが呼び出されている最中に CPU が切り替わって別のスレッドから呼び出されても，

先の呼び出しの処理になんらの影響が生じないものをいう．たとえば，図-5(a)のように内部変数をまったく使わないものや，(b)のように内部変数としてスタック上の動的変数のみしか使わないものはリエントラントである．しかし図-5(c)のように，メモリ上に静的におかれる static 変数を内部変数にもつようなものは，static 変数に対するアクセスが競合するためリエントラントではない．また，内部変数だけでなく外部の静的変数の参照や更新を行うものも，たいていの場合それらに対するアクセスが競合してリエントラントではない．

Cのランタイムルーチンについてマニュアルで調べてみると，浮動小数点ライブラリや printf, scanf のような入出力ライブラリなど，多くのものがリエントラントでないことに気づくであろう．このためマルチスレッド・プログラムでは，リエントラントでないルーチンの同時使用の問題は意外と深刻である．

(3) 複数のスレッドが同じ資源を連続して利用する場合

OS/2 では同じ資源を複数のスレッドが同時に使用しても，DosWrite や DosRead などの入出力用のAPIで実際の資源に対するアクセスの制御が行われているので，個々の DosWrite や DosRead などの API についてアクセスの競合を心配する必要はない．しかし，あるスレッドが連続して使用している間に別のスレッドが割り込んで使用するというような，同じ資源を連続して利用する場合のアクセスの競合の問題は起こり得る．

```
        int iabs(int x) {
            if(x<0) return(-x);
            return(x);
        }
(a)  リエントラントなルーチンの例(1)
        int iabs(int x) {
            int y;
            y=x; if(y<0) y=-y;
            return(y);
        }
(b)  リエントラントなルーチンの例(2)
        int iabs(int x) {
            static int y;
            y=x; if(y<0) y=-y;
            return(y);
        }
(c)  リエントラントでないルーチンの例
```

図-5 リエントラントなルーチンとそうでないルーチンの例

以上のようなアクセスの競合を避けるには，ある資源に対する処理は一つのスレッドのみが行うようにするという方法と，ある資源に対する処理はスレッド間で排他的に実行されるように制御するという二つの方法のいずれかを行わなくてはならない．上の(2)と(3)のケースでは，あるリエントラントでないルーチンを使うスレッドを一つに限るとか，ある資源をアクセスするスレッドを一つにするというような前者の方法でたいていの場合アクセスの競合を避けることができる．しかし(1)のケースでは多くの場合後者の方法すなわちスレッド間の処理の排他制御が必要となる．

2.3 スレッド間の排他制御の方法

並行プログラムにおける排他制御の実現方法については，古くから相互排他問題 (mutual exclusion problem) とか共有資源の逐次実行制御の問題 (serialization problem of shared resources) として研究されており，Dijkstra のセマフォ (semaphore), Brinch Hansen と Hoare の危険領域 (critical region), Hoare の条件つき危険領域 (conditional critical region), Hoare が提案し Brinch Hansen が Concurrent Pascal 言語で実現したモニタ (monitor) などのツールがよく知られている．OS/2 では，この中でも最もプリミティブなツールであるセマフォが提供されている．OS/2 のセマフォは Dijkstra のセマフォとはだいぶ異なったもののようにみえるが，その基になる考え方は同じである．参考のために，付録1に Dijkstra のセマフォの概要を示しておく．

(1) OS/2 のセマフォの概要

OS/2 のセマフォは4バイトの変数で，最初は常に0に初期化して使用される．Dijkstra のセマフォは正や負の数値をもつが，OS/2 のセマフォの場合はセットとクリアという二つの値をもつ．RAM セマフォはスレッドやプロセスの共有メモリ上のセマフォであり，システムセマフォはファイル名と同様の ¥SEM¥path¥name. exe といった名前を指定して，DosCreateSem という API でシステム領域内に作成されるセマフォである．セマフォを操作する API は DosSemClear, DosSemRequest, DosSemSet, DosSemWait, DosSemSetWait, DosMuxSemWait の6つである．セマフォのアドレスはセマフォハンドルと呼ばれ，上記の API の引数として使用される．OS/2 バージョ

ン 1.1 で導入された FS RAM セマフォの場合は，4 バイトの RAM セマフォとそれを使用しているスレッドのプロセス ID やスレッド ID，再帰的排他要求の回数などを保持する 14 バイトの変数からなり，その操作用の API は DosFSRamSemClear と DosFSRamSemRequest の二つのみである．

マルチスレッド・プログラムではおもに RAM セマフォが使われるので，以下今回の例題ではすべて RAM セマフォを用いる．

（2） OS/2 のセマフォによる排他制御の方法

一つのセマフォ mutex を用いて，二つのスレッドの処理 S_1 と S_2 を排他制御する場合の例を次に示す．

```
セマフォの初期化：unsigned long mutex=0;
スレッド A：...DosSemRequest (&mutex,
                -1L); S1; DosSemClear
                (&mutex); ...
スレッド B：...DosSemRequest (&mutex,
                -1L); S2; DosSemClear
                (&mutex); ...
```

このように，排他的に実行したい処理の前後で同一のセマフォに対する DosSemRequest と DosSemClear を呼ぶことで，それらの処理の排他的制御が実現される．ここで DosSemRequest はセマフォの排他的な所有とセットを要求するもので，セマフォが別のスレッドによりセットされていたならクリアされるまで待つ．DosSemClear はセマフォをクリアし解放するものであるが，別のスレッドが DosSemRequest で待っていたならクリアと同時にその一つを受け付けるので，その場合はセットされたままとなる．DosSemRequest の 2 番目の引数は，セマフォがクリアされるまでの待ち時間（ミリ秒単位）を指定する 4 バイトの整数で，この値が負のときは無限に待つことを意味する．

あるセマフォで排他制御されている処理の中から同じセマフォで排他制御されている処理を実行すると，再帰的な排他要求が発生する．**付録 1** に述べたように，Dijkstra のセマフォでは再帰的な排他要求を行うとデッドロックが生ずるのが問題であったが，OS/2 のセマフォの場合は DosSemRequest を再帰的に実行した回数がカウントされるだけでデッドロックは生じないようになっている．この場合，同じ回数の DosSemClear を実行しないとセマフォはクリアされない．

（3） その他の排他制御の方法

OS/2 にはこのほかに，排他制御したい部分を DosEnterCritSec と DosExitCritSec で囲むという方法が提供されている．これは DosEnterCritSec から DosExitCritSec の間，他のスレッドの実行を強制的に禁止するもので，それと関係なく並行に実行可能なスレッドの実行も禁止されてしまうのが欠点である．

2.4 スレッド間の同期の方法

次に，スレッド間の同期の実現方法について述べよう．一般に同期は，送信側からの事象（これを同期事象と呼ぶ）の通知を受信側で待つという互いの処理によって達成される．このとき受信側が常に先に待っているとは限らないので，通知された同期事象を受信されるまで保持する必要がある．スレッド間では，共有変数やセマフォを使って同期を行うことができる．

（1） 共有変数によるスレッド間の同期

一つの論理型の共有変数 ev を用いてスレッド間で同期をとる例を次に示す．

```
変数の初期化：int ev=FALSE;
送信スレッド：S1; ev=TRUE; ...
受信スレッド：...while (!ev) DosSleep(10L);
```

ここで，DosSleep(t) は t ミリ秒間（t は unsigned long 型）スレッドの実行を休止する API である．これにより，送信側スレッドの処理 S_1 が完了するまで受信側スレッドはループして待たされる．図-4 の中で，論理変数 GO を用いて，親のスレッドが生成したスレッドの処理を待たせているのがこの例である．

二つのスレッドが繰り返し同期しながら実行する場合は，二つの論理変数 s と t を次のように用いる．

```
初期化：int s=FALSE, t=TRUE;
スレッド A：while(TRUE) {
                while(!t) DosSleep(10L);
                t=FALSE;
                S1; s=TRUE; ...}
スレッド B：while(TRUE) {...
                while(!s) (DosSleep(10L);
                s=FALSE;
                S2; t=TRUE;}
```

以上のように共有変数による同期は簡単で分かりやすいが，受信側でループしながら待つ繁忙待

機 (busy-waiting) を必要とするのが欠点である．繁忙待機は CPU 時間を無駄に消費し，場合によってはシステム全体のレスポンスを落としてしまう．OS/2 では，上の例のように CPU を消費しないで待つ DosSleep がよく使われるが，一回の DosSleep の待ち時間を長くすると今度は受信側の処理の再開が遅くなるという問題が生ずる．

（2） セマフォによるスレッド間の同期

セマフォによる同期は，このような問題がないという点でより好ましいものである．次に一つのセマフォ synch を用いてスレッド間で同期をとる例を示す．

```
セマフォの初期化：unsigned long synch=0;
                  DosSemSet (&synch);
送信スレッド：S1; DosSemClear (&synch);
                  ...
受信スレッド：...DosSemWait (&synch,
                  -1 L);
```

ここで，DosSemSet (&synch) はセマフォ synch を単にセットするだけの API，DosSemWait (&synch, −1 L) はセマフォ synch のクリアを待つ API で，この 2 番目の引数は DosSemRequest のときと同じようにセマフォがクリアされるまでの待ち時間である．セマフォのクリアには，排他制御のときと同じ DosSemClear を使用する．なおこのほかに，セマフォをセットして待つ DosSemSetWait や複数のセマフォの中のどれか一つがクリアされるのを待つ DosMuxSemWait などの API も利用可能である．

二つのスレッドが繰り返し同期しながら実行する場合は，二つののセマフォ s と t を次のように用いる．

```
初期化：unsigned long s=0, t=0;
                DosSemSet (&s);
スレッド A：while (TRUE) {
                DosSemWait (&t,-1 L);
                DosSemSet (&t);
                S1; DosSemClear (&s); ...}
スレッド B：while (TRUE) {...
                DosSemWait (&s, -1L);
                DosSemSet (&s);
                S2; DosSemClear (&t); }
```

（3） 複数のスレッドが待っている場合の問題

Dijkstra のセマフォでは一度の V 操作で実行再開するスレッドは高々一つであるが，OS/2 のセマフォでは一度の DosSemClear でそれを待っているすべてのスレッドの実行再開が可能となる．そこで逆に，複数のスレッドが待っているときに，その中の一つだけを実行再開したい場合どうしたら良いかが問題となる．OS/2 の DosSemWait や DosSemSetWait はこのために，最初に実行再開されたスレッドがすぐにセマフォをセットしてしまえば，残りのスレッドは待たされたままとなるという仕様になっている．このような実行再開のしかたを OS/2 ではレベルトリガと呼んでいる．後の読み取り／書き込み問題のプログラム例では，これを利用して書き込みスレッドの排他的な実行を実現している．

（4） 複数の事象を待つ方法

同期の問題の最後に，一つのスレッドが複数の事象を待つ方法について述べよう．まず複数の事象のうちのどれか一つを待つ場合は DosMuxSemWait を使うのが最も良い．DosMuxSemWait は最大 16 個までのセマフォのリストを指定し，その中のうち一つでもクリアされたなら戻ってくるという API である．DosMuxSemWait では，クリアされたセマフォがすぐにセットされても，クリアされたことが記憶されているので一度は必ず実行再開される．ただしクリアされた回数までは記憶されていないので，同じセマフォが何回クリアされても一度クリアされた場合と同じである．このような実行再開のしかたを OS/2 では，エッジトリガと呼んでいる．一方，複数の事象のすべてが通知されるまで待つ場合は適当な APIがなく，セマフォの数だけ DosSemWait を実行するのが最も効率のよい待ち方になる．たとえば，セマフォ p, q, r, s がすべてクリアされるまで待つ場合は次のようにする．

```
DosSemWait (&p, -1L);
DosSemWait (&q, -1L);
DosSemWait (&r, -1L);
DosSemWait (&s, -1L);
```

このような場合に DosMuxSemWait を使うのはあまり勧められない．また共有変数 p, q, r, s を使う場合は，

```
while (!p||!q||!r||!s) DosSleep (10L);
```

のような繁忙待機となる．

2.5 古典的な並行プログラミング問題

次に，古典的な並行プログラミング問題に対するプログラム例を示そう．古典的な並行プログラミング問題は，OS の設計者にとってはセマフォなどの基本的な同期のツールの機能の検証に有用であり，ユーザにとってはその正しい使い方をマスタするのに有用である．

（1） 生産者/消費者問題

生産者/消費者問題(producer/consumer problem)は，「生産者はデータを生産して消費者に渡す作業を繰り返し，消費者はデータを受け取って消費する作業を繰り返す」という作業を並行に実行させるものである．このとき，消費者スレッドはデータがまだ転送されていない場合待たねばならず，生産者スレッドはデータを転送するバッファがいっぱいになったら空きができるまで待たねばならない．簡単な例として，バッファが一つのみの場合を考えよう．

```
char b;
unsigned long full=0;
unsigned long empty=0;
DosSemSet (&empty);
```

ここで，セマフォ full はバッファにデータが入っているときにセットされ，セマフォ empty は逆に空のときにセットされる．生産者スレッドは，生産したデータを

```
DosSemWait (&full, -1 L);
DosSemSet (&full);
b=data;
DosSemClear (&empty);
```

によりバッファに格納し，消費者スレッドは，

```
DosSemWait (&empty, -1 L);
DosSemSet (&empty);
data=b;
DosSemClear (&full);
```

によりバッファからデータを受け取り消費することを繰り返す．ここで，DosSemWait の後すぐに DosSemSet を行っているので，生産者スレッドや消費者スレッドがそれぞれ複数いても，バッファへの格納や取り出しは排他的に行われる．バッファを N 個に増やした場合の例を**図-6** に示す．バッファを N 個に増やした場合は，生産者スレッドと消費者スレッドが同時にバッファ操作を行わないような排他制御が必要となる．なおここでは

```
/*-------------------------------------------------*/
/* 生産者／消費者問題のプログラムの例              */
/* [MS-C による作成例] cl expc.c expc.def          */
/* [モジュール定義ファイル expc.def の例]          */
/*    NAME   EXPC   WINDOWCOMPAT                   */
/*    PROTMODE                                     */
/*    STACKSIZE 4096                               */
/* [使用例] expc                                   */
/*-------------------------------------------------*/
#include <stdio.h>
#include <stdlib.h>      /*    malloc             */
#include "sample2.h"     /* 例題用ヘッダファイル  */
#include "mkchild.sub"   /* makechildth ルーチン  */
#include "err.sub"       /* err ルーチン          */
/*- スレッド開始ルーチンのプロトタイプ宣言 ---------*/
void far producer(void);      /* 生産者スレッド   */
void far consumer(void);      /* 消費者スレッド   */
/*- スレッド間の共通変数 ---------------------------*/
#define  BSIZE  128      /* バッファサイズ        */
char     b[BSIZE];       /* バッファ              */
int      inp=0, out=0;   /* バッファのポインタ    */
int      b_cnt = 0;      /* バッファ内のカウンタ  */
ULONG full = 0;          /* バッファ満杯セマフォ  */
ULONG empty = 0;         /* バッファ空きセマフォ  */
ULONG mutex =0;          /* 排他制御用のセマフォ  */
/*==== メインプログラム（スレッド1）==============*/
main( ) { ULONG pausel = 0;  /* 完了待ちセマフォ */
    printf("Producer/Consumer Problem. ¥n");
    DosSemSet(&empty);   /* 最初はバッファは空   */
    makechildth(producer); makechildth(consumer);
    DosSemSetWait(&pausel, -1L);  /* 永久に待ち */
}
/*==== producer: 生産者スレッドの例 ==============*/
/* 標準入力から1文字入力しバッファに格納する      */
void far producer(void) { USHORT nw,nr; char c;
    while(TRUE) {
      DosRead(0, &c, 1, &nr);          /*1文字入力 */
      if(nr==0) c=0x1a;                /*nr==0: EOF*/
      DosSemWait(&full, -1L);          /*満杯なら待つ*/
      DosSemRequest(&mutex, -1L); /*排他制御開始*/
        /* 入力文字をバッファに入れカウンタを +1 */
        b[inp]=c; inp = (inp+1)%BSIZE; b_cnt++;
        /* バッファが満杯になったら full をセット */
        if(b_cnt==BSIZE) DosSemSet(&full);
      DosSemClear(&mutex);     /* 排他制御終了   */
      DosSemClear(&empty);     /* emptyをクリア  */
    }
}
/*==== consumer: 消費者スレッドの例 ==============*/
/* バッファから1文字取り出し標準出力に出力する    */
void far consumer(void) { USHORT nw; char c;
    while(TRUE) {
      DosSemWait(&empty, -1L); /* 空なら待つ    */
      DosSemRequest(&mutex, -1L);
                              /* 排他制御開始 */
        /* バッファから1文字取り出しカウンタを-1*/
        c=b[out]; out = (out+1)%BSIZE; b_cnt--;
        /* バッファが空になったら empty をセット */
        if(b_cnt==0) DosSemSet(&empty);
      DosSemClear(&mutex);     /* 排他制御終了   */
      DosSemClear(&full);      /* full をクリア  */
      DosWrite(1, &, c, 1, &nw);   /* 1文字出力  */
        /* ファイル終了文字ならプロセスを終了     */
      if(c==0x1a) DosExit(EXIT_PROCESS, 0);
    }
}
```

図-6 生産者／消費者問題のプログラム例

mkchild. sub と err. sub に図-4 の makechildth ルーチンと err ルーチンを入れて include で取り込んでいる.

以上から分かるように，生産者／消費者問題は，並行に実行されるプロセスやスレッド間のデータ転送の問題である．並行プログラムでは，データ転送は非常に一般的に行われるものであり，その形態についても図-6 のような FIFO (first-in first-out) 形のリングバッファや，LIFO (last-in first-out) 形のスタックなどいろいろある．OS/2 のパイプや待ち行列は，プロセス間やスレッド間のデータ転送用のツールとして提供されているもので，これらのツールを使えば図-6 のような複雑なプログラムを作らなくても簡単にデータ転送が可能である.

```
/*-------------------------------------------------*/
/* 読み取り／書き込み問題のプログラムの例          */
/* [MS-C による作成例]  cl exrw.c exrw.def         */
/* [モジュール定義ファイル exrw.def の例]          */
/*    NAME   EXRW   WINDOWCOMPAT                   */
/*    PROTMODE                                     */
/*    STACKSIZE 4096                               */
/* [使用例]  exrw                                  */
/*-------------------------------------------------*/
#include <stdio.h>
#include <stdlib.h>      /*   malloc               */
#include "sample2.h"     /* 例題用ヘッダファイル   */
#include "mkchild.sub"   /* makechildth ルーチン   */
#include "err.sub"       /* err ルーチン           */
#define  wmsg(m) DosWrite(1, m, sizeof(m)-1, &nw)
#define  idle(s) DosSleep(s*1000L)
                 /*- スレッド間の共通変数 ---------*/
ULONG  wait_ch = 0;      /* 開始待ち用のセマフォ   */
int    n = 0;            /* 実行中のスレッドの数   */
ULONG  me = 0;           /* 排他制御用セマフォ     */
/*- スレッド開始ルーチンのプロトタイプ宣言 ---------*/
void far reader(void);   /* writer スレッド        */
void far writer(void);   /* reader スレッド        */
/*==== メインプログラム (スレッド1) ==============*/
char title [ ]="Readers/Writers Problem. ¥r¥n";
main( ) { int i; USHORT nw; wmsg(title);
    DosSemSet(&wait_ch);       /* 開始待ちセット   */
    for(i=0; i<5; i++) makechildth(reader);
    for(i=0; i<3; i++) makechildth(writer); n=8;
    DosSemClear(&wait_ch);     /* 開始待ち解除     */
    while(n>0) DosSleep(10L);
}
/*==== reader: 読み取りスレッドの例 ==============*/
char rreq[ ]="R", rd[ ]="(", rext[ ]=")";
void far reader(void) { int i; USHORT nw;
    DosSemWait(&wait_ch, -1L); /* 開始待ち         */
    for(i=0; i<10; i++) { idle(2);
        wmsg(rreq); R_request( );  /* read 要求    */
        wmsg(rd); idle(2);         /* read         */
        wmsg(rext); R_exit( );     /* read 終了    */
    }
    DosSemRequest(&me, -1L); n--; DosSemClear(&me);
    DosExit(EXIT_THREAD, 0);
}
```

（2） 読み取り／書き込み問題

読み取り／書き込み問題 (readers/writers problem) は，ファイルへの読み書きにおける同時アクセスの制御に関連した問題である．ファイルの読み取りは複数のスレッドが同時に行ってもかまわないが，書き込みは他のスレッドの書き込みや読み取りを禁止して排他的に行われる必要がある．このような複雑なアクセスの実現方法を問うのがこの問題の主旨である.

この問題に対しては，いくつかの解があり得る．一つは，読み取りスレッドの実行中は，新たな読

```
/*==== writer: 書き込みスレッドの例 ==============*/
char wreq[ ]="W", wrt[ ]="{", wext[ ]="}";
void far writer(void) { int i; USHORT nw;
    DosSemWait(&wait_ch, -1L); /* 開始待ち         */
    for(i=0; i<10; i++) { idle(3);
        wmsg(wreq); W_request( );  /* write 要求   */
        wmsg(wrt); idle(1);        /* write        */
        wmsg(wext); W_exit( );     /* write 終了   */
}
    DosSemRequest(&me, -1L); n--; DosSemClear(&me);
    DosExit(EXIT_THREAD, 0);
}
/*---- 共通変数の定義と初期化 ---------------------*/
int      nwriters =0;    /* writer の数            */
int      nreaders = 0;   /* reader の数            */
ULONG    readable = 0;   /* 読み取り可セマフォ     */
ULONG    writable = 0;   /* 書き込み可セマフォ     */
ULONG    mutex = 0;      /* 排他制御用のセマフォ   */
/*---- R_request, R_exit, W_request, W_exit --------*/
R_request( ) {           /* 読み取り要求           */
    DosSemWait(&readable, -1L);
    DosSemRequest(&mutex, -1L);
      nreaders++; DosSemSet(&writable);
    DosSemClear(&mutex);
}
R_exit( ) {              /* 読み取り終了           */
    DosSemRequest(&mutex, -1L);
      nreaders--;
      if(nreaders==0) DosSemClear(&writable);
    DosSemClear(&mutex);
}
W_request( ) {           /* 書き込み要求           */
    DosSemRequest(&mutex, -1L);
      nwriters++; DosSemSet(&readable);
    DosSemClear(&mutex);
    DosSemWait(&writable, -1L);
    DosSemSet(&writable);
}
W_exit( ) {              /* 書き込み終了           */
    DosSemRequest(&mutex, -1L);
      nwriters--; DosSemClear(&writable);
      if(nwriters==0) DosSemClear(&readable);
    DosSemClear(&mutex);
}
```

図-7　読み取り／書き込み問題のプログラム例

み取りスレッドは実行開始できるが書き込みスレッドは実行開始できないという reader 優先の解である．もう一つは，書き込みスレッドがあったらその後の読み込みスレッドの要求を待たせるという writer 優先の解である．これらの解は，優先しているほうの要求が連続すると，そうでないほうがずっと待たされるという問題を含んでいる．ここでは，writer 優先の解を図-7 に示した．

読み取り／書き込み問題は共有資源管理の方法に関する問題である．共有資源管理の方法としては，セマフォによる排他制御が一般によく使われるが，読み取り／書き込み問題はさらに並行処理を高めるために条件によっては同時アクセスを許すというより高度な管理方法の実現方式を問うものであるといえよう．

```
/*-------------------------------------------------*/
/* sample2. h: 例題で使用するヘッダファイルの例    */
/* なお OS/2 のソフトウェア開発キットに付属の      */
/* ヘッダファイルがある場合は以下の文だけでよい    */
/*       #define INCL_BASE                         */
/*       #include <os2. h>                         */
/*-------------------------------------------------*/
                    /*-- 使用するデータタイプの定義 --*/
#define LONG long
typedef unsigned long ULONG;   /* 符号なし long    */
typedef unsigned char BYTE;    /* バイト型         */
typedef BYTE far * PBYTE:      /* そのポインタ     */
typedef unsigned short USHORT; /* 符号なし整数     */
typedef USHORT far * PUSHORT;  /* そのポインタ     */
typedef USHORT TID;            /* スレッド ID      */
typedef TID far * PTID;        /* そのポインタ     */
typedef void (far *PFN) (void); /* 関数のポインタ  */
typedef void far * HSEM;       /* セマフォハンドル */
#define FALSE 0
#define TRUE 1
                    /*==== 使用する API の定義 ======*/
#define APIENTRY pascal far    /* API 定義指示子   */
USHORT APIENTRY DosCreateThread
                              (PFN, PTID, PBYTE);
#define EXIT_CODE USHORT
#define EXIT_THREAD 0
#define EXIT_PROCESS 1
#define RTN_CODE USHORT
void     APIENTRY DosExit(EXIT_CODE, RTN_CODE);
USHORT APIENTRY DosSleep(ULONG);
USHORT APIENTRY DosSemClear(HSEM);
USHORT APIENTRY DosSemRequest(HSEM, LONG);
USHORT APIENTRY DosSemSet(HSEM);
USHORT APIENTRY DosSemWait(HSEM, LONG);
USHORT APIENTRY DosSemSetWait(HSEM, LONG);
#define HFILE     USHORT
#define B_LEN     USHORT
#define PR_LEN    PUSHORT
#define PW_LEN    PUSHORT
USHORT APIENTRY DosRead
                    (HFILE, PBYTE, B_LEN, PR_LEN);
USHORT APIENTRY DosWrite
                    (HFILE, PBYTE, B_LEN, PW_LEN);
```

図-8 今回例題で使用したヘッダファイルの例

2.6 ま と め

今回は，マルチスレッドプログラムの作成に関連して，OS/2 におけるスレッドの生成と終了の方法，スレッド間のアクセスの競合の問題，スレッド間の排他制御の方法，スレッド間の同期の方法について解説し，さらにスレッド間のデータ転送や共有資源管理に関して，有名な古典的問題である生産者／消費者問題と読み取り／書き込み問題のプログラム例を示した．古典的な問題の詳細については，前回の参考文献 12) や 13) などに詳しく載っているので参考にされたい．なお，今回の例題で用いたヘッダファイル sample 2. h の内容は図-8 のようなものである．この代りに付属の os2. h を使っても良いが，os2. h のバージョンにより若干のちがいがありエラーが出ることがある．そのような際には図-8 を参考にされたい．また OS/2 の 32 ビット対応のバージョン 2. 0 ではセマフォなどの仕様が変更されたので，バージョン 2. 0 以上では例題の書き換えが必要である．

付録 1 Dijkstra のセマフォの概要

（1） Dijkstra のセマフォの定義

Dijkstra のセマフォは整数変数 sem とそれに対する P(sem) と V(sem) の二つの操作からなる．ここで P(sem) と V(sem) は次のように定義される．

P(sem)：sem＝sem－1；
　　if (sem＜0) 待ち行列 Q(sem) の中で待つ；

V(sem)：sem＝sem＋1；
　　if(sem＜＝0) Q(sem) の中の一つを実行再開；

［補足］ セマフォはスレッド間の共有変数である．このため，P 操作や V 操作におけるセマフォの変更は，ハードウェア的な機構などを利用して排他的に行われる必要がある．

（2） Dijkstra のセマフォによる排他制御の方法

一つのセマフォ mutex を用いて二つのスレッドの処理 S_1 と S_2 を排他制御する場合の例を次に示す．

セマフォの初期化：mutex＝1；

スレッド A : ...P(mutex) ; S_1 ; V(mutex) ; ...

スレッド B : ...P(mutex) ; S_2 ; V(mutex) ; ...

この例のように，排他的に実行したい処理の前後で同一のセマフォに対する P 操作と V 操作を行うことで同様の操作を行っている任意の処理との間の排他制御が可能になる．

［補足］ Dijkstra のセマフォでは，あるセマフォにより排他的に実行されている処理の中から同じセマフォにより排他制御されている処理を呼び出すと，最初の P 操作の後続けて P 操作が行われて自分自身をデッドロックしてしまうという問題がある．

（3） Dijkstra のセマフォによる同期の方法

一つのセマフォ synch を用いて二つのスレッド間で同期をとる場合の例を次に示す．

セマフォの初期化 : synch＝0 ;

スレッド A (送信側) : S_1 ; V(synch) ; ...

スレッド B (受信側) : ...P(synch) ; S_2 ;

これにより，スレッド A の処理 S_1 の実行完了後にスレッド B の処理 S_2 が実行される．さらに，スレッド B の処理 S_2 の完了後に再びスレッド A の処理 S_1 をすることを繰り返すには，二つのセマフォを次のように用いればよい．

セマフォの初期化 : s=0 ; t=1 ;

スレッド A : while(e) {$P(t)$; S_1 ; $V(s)$; ...}

スレッド B : while(e) {...$P(s)$; S_2 ; $V(t)$; }

（平成2年11月13日受付）

鷹野　澄（正会員）

昭和27年生．昭和50年静岡大学工学部電気工学科卒業．昭和55年東京大学大学院工学系研究科電子工学専門課程博士課程修了．工学博士．昭和55年東京大学大型計算機センター助手，昭和58年東京大学地震研究所講師（地震予知観測情報センター）．情報理論，オペレーティングシステム，プログラミング言語，ネットワーク，データベースおよび地震予知情報システムなどの研究・開発・運用に従事．著書「MS-DOS」「OS/2」．情報システム研究会幹事．電子情報通信学会，IEEE，ACM，人工知能学会，地震学会など各会員．