講　座

OS/2 における並行プログラムの作り方 (III)†

鷹　野　澄††

3. OS/2 カーネル上のマルチプロセス・プログラミング

今回は，複数のプロセスを生成して並行処理するマルチプロセス・プログラムの作り方について述べる．マルチプロセス・プログラムでは，厳格なメモリ保護のもとで，プロセス間のメモリの共有や同期，データの転送，共有資源の管理などをいかにして行うかというプロセス間通信 (interprocess communication : IPC)の問題が重要である．また，デッドロック (deadlock) や飢餓状態 (starvation) の問題にも注意する必要がある．

3.1 OS/2 におけるプロセスの生成

ここでは，まず OS/2 におけるプロセスの生成方法について述べておこう．OS/2 では，DosExecPgm という API でプログラムを起動することにより新たなプロセスを生成する．このとき生成されたほうを子プロセス，生成したほうを親プロセスと言い，すべてのプロセスは基本的にこの親子関係で結ばれたプロセスツリー (process tree) を形成する．プロセスには，プロセス ID (process identifier) と呼ばれる，システム内でユニークな番号が割り当てられる．

DosExecPgm では，親プロセスと並行に実行可能な非同期プロセス (asynchronous process)，実行終了まで親プロセスを待たす同期プロセス (synchronous process)，親プロセスやそのセッションから切り離されて実行されるバックグラウンド・プロセス (background process) などを生成できる．マルチプロセス・プログラムでは，並行に実行できる非同期プロセスやバックグラウンド・プロセスがよく使われる．図-1 に，smserver. exe と smclient. exe という 2 つのプログラムを非同期プロセスとして起動する DosExecPgm の使用例を示す．この例では，親プロセスは，子プロセスの終了を待たずに終了しているが，DosCwait という API で子プロセスの終了を待つこともできる．DosCwait では，指定した子プロセスの終了のみを待つか，指定した子プロセスの下のプロセスツリー内のすべてのプロセスの終了を待つかを指定できる．

3.2 プロセス間のメモリ共有，同期ならびにデータ転送の問題

プロセス間では互いのメモリが保護されているため，マルチスレッド・プログラムのときのように，プロセス間でのメモリの共有や同期，データの転送などが簡単ではない．このため OS/2 では，本講座の第 1 回 (1990 年 10 月号) で解説したように，プロセス間通信機構として，共有メモリ，セマフォ，パイプ，キュー，シグナルなどの種々のツールが提供されている．また，DLL

```
/*--- smcreate.c ----------------------------------------------*/
/* [MS-Cによる作成例] cl smcreate.c smcreate.def  */
/* [モジュール定義ファイル smcreate.defの例]      */
/*   NAME  SMCREATE  WINDOWCOMPAT                 */
/*   PROTMODE                                     */
/*   STACKSIZE 4096                               */
/*--------------------------------------------------------------*/
#include "sample3.h"    /* 例題用ヘッダ(図-10参照)*/
#include "err.sub"      /* errルーチン            */
#define  FFL 128        /* 変数ffnの長さ          */
char ffn[FFL];          /* failed file名バッファ  */
RESULTCODES rcs;        /* 結果コードバッファ     */
main() { USHORT rc;
                        /* smserverプロセスの生成 */
  rc=DosExecPgm(ffn,FFL,EXEC_ASYNC,
      "smserver¥0¥0",0,&rcs,"smserver.exe");
  if(rc) err("DosExecPgm(smserver) failed.",rc);
  DosSleep(100L);       /* smserverの準備を待つ   */
  printf("smserver pid=%u¥n",rcs.codeTerminate);
                        /* smclientプロセスの生成 */
  rc=DosExecPgm(ffn,FFL,EXEC_ASYNC,
      "smclient¥0¥0",0,&rcs,"smclient.exe");
  if(rc) err("DosExecPgm(smclient) failed.",rc);
  printf("smclient pid=%u¥n",rcs.codeTerminate);
  printf("Now, smcreate terminated.¥n");
}
```

図-1　smserver. exe と smclient. exe を非同期プロセスとして起動する DosExecPgm の使用例

† Concurrent Programming in OS/2(III) by Kiyoshi TAKANO (Earthquake Research Institute, University of Tokyo).
†† 東京大学地震研究所

(dynamic-link library) もプロセス間通信や，自分用のプロセス間通信ツールの作成に利用できる．

このうち共有メモリとセマフォは，最もプリミティブなツールである．これらは多くの場合，データ転送などを実現するために，組み合わされて使用される．そこでここではまず，共有メモリとセマフォの使用法を個々に解説するのではなく，それらを組み合わせてプロセス間でデータ転送を行う例を示そう．

（1） 共有メモリとセマフォを用いたプロセス間のデータ転送のプログラム例

図-2 は，名前つき共有メモリとシステムセマフ

```
/*--- smserver.c:サーバプロセス ----------*/
/* [MS-Cによる作成] cl smserver.c smserver.def */
/* [モジュール定義ファイルの例は省略]          */
/*-------------------------------------------*/
#include "sample3.h"   /* 例題用ヘッダファイル */
#include "err.sub"     /* errルーチン          */
#define CtlZ   0x1a    /* ファイル終了文字     */
#define MSG_SIZE 81    /* メッセージの最大長+1 */
main() { USHORT nw; int s;
   char msg[MSG_SIZE];  /* 受信メッセージ      */
   sm_create();  /* 共有メモリとセマフォの生成 */
   while(TRUE) {
    sm_receive(msg,&s);     /* メッセージを受信  */
    DosWrite(1,msg,s,&nw); /* 標準出力に出力    */
      /* ファイル終了文字ならプロセスを終了     */
    if(msg[0]==CtlZ) {
     sm_delete(); /* 共有メモリとセマフォの解放 */
     DosExit(EXIT_PROCESS,0); /* プロセス終了 */
    }
   }
}
/*== 共有メモリとセマフォの変数 ==========*/
                /*   共有メモリの名前と大きさ  */
#define SM_NAME  "¥¥sharemem¥¥smline"
#define SM_SIZE   MSG_SIZE
SEL sm_sel;              /* 共有メモリのセレクタ */
char far * sm;           /* 共有メモリのポインタ */
              /*   システムセマフォの名前       */
#define SM_FULL   "¥¥sem¥¥smfull"
#define SM_EMPTY  "¥¥sem¥¥smempty"
HSYSSEM hfull, hempty; /*システムセマフォハンドル*/
/*== 共有メモリとセマフォの生成 ==========*/
sm_create() { USHORT rc;
   /*----- 名前つき共有メモリの割り当て ----------*/
   rc=DosAllocShrSeg(SM_SIZE,SM_NAME,&sm_sel);
   if(rc) err("DosAllocShrSeg failed.",rc);
   sm = MAKEP(sm_sel,0);  /* ポインタのセット */
   /*----- システムセマフォの作成と初期化 ----------*/
   rc=DosCreateSem(CSEM_PUBLIC,
                   &hfull,SM_FULL);
   if(rc) err("DosCreateSem(full) failed.",rc);
   rc=DosCreateSem(CSEM_PUBLIC,
                   &hempty,SM_EMPTY);
   if(rc) err("DosCreateSem(empty) failed.",rc);
   DosSemSet(hempty);  /* 初期値はバッファが空 */
}
/*== メッセージの受信 ================== */
sm_receive(msg,s) char *msg; int *s; { int i;
   DosSemWait(hempty,-1L);   /* emptyなら待つ */
   DosSemSet(hempty); /* すぐにemptyをセット */
     /*  バッファから1行メッセージを取り出す */
     i=0; while(sm[i]){ msg[i]=sm[i]; i++; }
     *s=i;
   DosSemClear(hfull); /* fullをクリア */
}
/*== 共有メモリとセマフォの解放 ==========*/
sm_delete() {
   DosCloseSem(hfull); DosCloseSem(hempty);
   DosFreeSeg(sm_sel);
}
```

```
/*--- smclient.c:クライアントプロセス ----------*/
/* [MS-Cによる作成] cl smclient.c smclient.def */
/* [モジュール定義ファイルの例は省略]          */
/*-------------------------------------------*/
#include "sample3.h"   /* 例題用ヘッダファイル */
#include "err.sub"     /* errルーチン          */
#define CtlZ   0x1a    /* ファイル終了文字     */
#define MSG_SIZE 81    /* メッセージの最大長+1 */
main() { USHORT nr; int i; char c;
   char msg[MSG_SIZE]; /* 送信メッセージ       */
   sm_open(); /* 共有メモリとセマフォのオープン */
   while(TRUE) {  /* 標準入力より1行取り出す */
    for(i=0;i<MSG_SIZE-1;) {
     DosRead(0,&c,1,&nr);   /*  1文字入力     */
     if(nr!=1) if(i==0) c=CtlZ; else c='¥n';
     msg[i]=c; i++; if(c=='¥n'||nr!=1) break;
    }
    sm_send(msg,i);      /*  メッセージの送信  */
     /* ファイル終了文字によりプロセスを終了   */
    if(msg[0]==CtlZ) {
     sm_close(); /* 共有メモリとセマフォの解放 */
     DosExit(EXIT_PROCESS,0); /* プロセス終了 */
    }
   }
}
/*== 共有メモリとセマフォの変数 ==========*/
                /*   共有メモリの名前と大きさ  */
#define SM_NAME  "¥¥sharemem¥¥smline"
#define SM_SIZE   MSG_SIZE
SEL sm_sel;              /* 共有メモリのセレクタ */
char far * sm;           /* 共有メモリのポインタ */
              /*   システムセマフォの名前       */
#define SM_FULL   "¥¥sem¥¥smfull"
#define SM_EMPTY  "¥¥sem¥¥smempty"
HSYSSEM hfull, hempty; /*システムセマフォハンドル*/
/*== 共有メモリとセマフォのオープン ======= */
sm_open() { USHORT rc;
   /*----- 名前つき共有メモリのオープン ----------*/
   rc=DosGetShrSeg(SM_NAME,&sm_sel);
   if(rc) err("DosGetShrSeg failed.",rc);
   sm = MAKEP(sm_sel,0);  /* ポインタのセット */
   /*----- システムセマフォのオープン ----------*/
   rc=DosOpenSem(&hfull,SM_FULL);
   if(rc) err("DosOpenSem(full) failed.",rc);
   rc=DosOpenSem(&hempty,SM_EMPTY);
   if(rc) err("DosOpenSem(empty) failed.",rc);
}
/*== メッセージの送信 ================== */
sm_send(msg,s) char *msg; int s; { int i;
   DosSemWait(hfull,-1L);    /*  fullなら待つ   */
   DosSemSet(hfull);    /*  すぐにfullをセット  */
            /*  バッファにメッセージを格納      */
    for(i=0;i<s;i++) sm[i]=msg[i];
    sm[i]=0; /* 最後に0をセット */
   DosSemClear(hempty); /* emptyをクリア */
}
/*== 共有メモリとセマフォの解放 ==========*/
sm_close() {
   DosCloseSem(hfull); DosCloseSem(hempty);
   DosFreeSeg(sm_sel);
}
```

図-2 名前つき共有メモリとシステムセマフォによるプロセス間のデータ転送の例

ォを用いて smserver と smclient という2つのプロセス間でデータ転送を行う例である．smserver プロセスは，最初に ¥sharemem¥smline という名前つき共有メモリと ¥sem¥smfull および ¥sem¥smempty という2つのシステムセマフォを生成して初期化し (sm_create ルーチン参照)，次いでそこに転送されたデータを出力することを繰り返す．一方smclient プロセスでは，すでに生成されている名前つき共有メモリとシステムセマフォをオープンし (sm_open ルーチン参照)，その後標準入力から入力されたデータを転送することを繰り返す．データは，最大 80 文字からなるデータブロック（これをメッセージと呼んでいる）を単位として転送される．なお一般に，このようにメッセージを受信して処理するほうをサーバ (server)，サーバにメッセージを送信し処理を依頼するほうをクライアント (client) と呼ぶ．

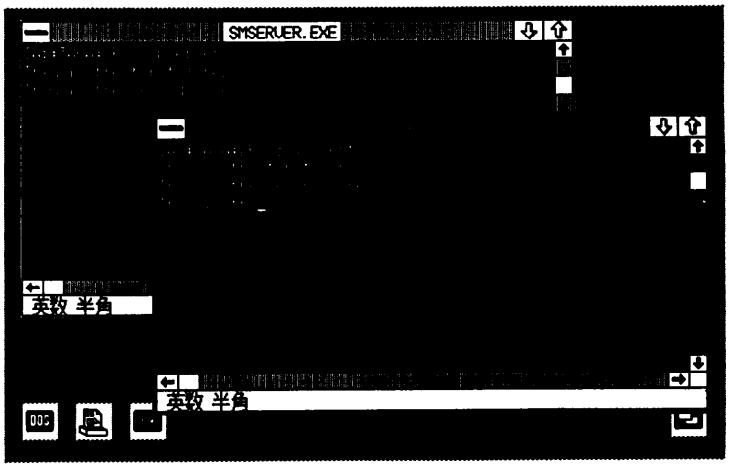


図-3 図-2 の smserver と smclient プロセスを別々のウィンドウ・セッション上で実行した例

図-3 は，この2つのプロセスを別々のウィンドウ・セッションで実行させた例である．smclient 側のウィンドウで入力したテキストが，1行ごとに smserver 側に転送されて表示される様子が一目で分かる．一方，前に示した図-1 は，この2つのプロセスを同一のセッション内で並行処理させるためのプログラムの例である．

図-2 では，簡単のために，メッセージ1個分の共有メモリを割り当てて，前回述べた単一バッファによる生産者／消費者問題の解を用いてプロセス間のデータ転送を実現した．それを N 個まで格納できるように拡張するのは読者への演習問題として残しておく．また，システムセマフォを使わずに，1つの共有メモリ上にバッファと RAM セマフォを置き，それだけでプロセス間のデータ転送を実現するというような変形も演習問題として適当であろう．

（2） DLL によるデータ転送用ツールの作成例

原理的には，共有メモリとセマフォさえあれば，プロセス間で任意のデータ転送が可能である．しかし，図-2 に示したように，個々のプロセスが別々にプログラムされるので，両方のプロセスの内容を見ないと何を行っているのか分かりにくく，間違いも生じやすい．そこでここでは，共有メモリとセマフォならびにそれらを操作するルーチンをまとめて DLL によりモジュール化し，独立なデータ転送用ツール（ライブラリ）を作成する方法について述べよう．

図-4 は，OS/2 の DLL で作成したプロセス間のデータ転送用ツールの例である．ここでは，図-2 の sm_xxx に相当する dl_xxx という名前の外から呼ばれるルーチンと，end_xxx という名前のプロセス終了ルーチンを定義している．ここで，dl_xxx の使用法は図-2 の sm_xxx と同じである．なお，図-4 では，図-2 とは別の共有メモリとセマフォの使用例を示すために，取得可能フラグを立てた名前なし共有メモリと DLL 内の共有データ上の RAM セマフォを用いている．名前なし共有メモリは，サーバから最初に dl_create ルーチンが呼ばれたときに，DosAllocSeg により割り当てられる．そしてクライアントが最初に dl_open ルーチンを呼んだときに，DosGetSeg により，その共有メモリに対するアクセス権が与えられる．

図-4 の dl_create, dl_open と図-2 の sm_create, sm_open を比較すると分かるように，図-4 では，ツールとして利用可能にするために若干の処理が追加されている．dl_create や dl_open の先頭では，created や connected という変数を調べて，2回以

上生成したり2回以上オープンしないようなチェックをしている．また，それらの最後では，DosExitList によりプロセス終了ルーチンを登録している．OS/2 のプロセス終了ルーチンは，プロセスが終了するときに自動的に呼ばれるもので，共有メモリを解放したりセマフォをクリアするなどの終了処理を確実に行うために，DLL においてよく用いられる．プロセス終了ルーチンは，図-4

```
/*--- dlbuffer.c ----------------------------------------------------*/
/* [MS-Cによるdlbuffer.dllの作成例]                                  */
/* cl -Asnu -Gs dlbuffer.c dlbuffer.def                              */
/*      -Fedlbuffer.dll doscalls.lib[又はos2.lib]                    */
/* [implibコマンドによるdlbuffer.libの作成例]                        */
/* implib dlbuffer.lib dlbuffer.def                                  */
/* [モジュール定義ファイル dlbuffer.defの例]                         */
/*   LIBRARY  DLBUFFER                                               */
/*   EXPORTS                                                         */
/*     DL_CREATE                                                     */
/*     DL_RECEIVE                                                    */
/*     DL_DELETE                                                     */
/*     DL_OPEN                                                       */
/*     DL_SEND                                                       */
/*     DL_CLOSE                                                      */
/*-------------------------------------------------------------------*/
#include "sample3.h"   /* 例題用ヘッダファイル   */
#define CtlZ    0x1a   /* ファイル終了文字       */
#define MSG_SIZE 81    /* メッセージの最大長+1   */
#define SM_SIZE MSG_SIZE /* 共有メモリサイズ     */
#define BUFFER_USED     1001 /* エラーコード     */
#define NOT_CREATED     1002
#define NOT_OPENED      1003
#define BUFFER_DELETED 1004
#define CLIENT_ABORTED 1005
ULONG full=0,empty=0; /* RAMセマフォ          */
SEL  sm_sel;          /* 共有メモリのセレクタ  */
char far * sm;        /* 共有メモリのポインタ  */
int  msg_len;         /* 転送メッセージの長さ  */
int  created = FALSE; /* 生成済みフラグ        */
int  connected=FALSE; /* クライアント接続中    */
int  aborted = FALSE; /* クライアント強制終了  */
 /* Cライブラリ初期化ルーチンcrt0の取込み抑止 */
int _acrtused = 0;
            /*  プロセス終了ルーチンの宣言    */
void EXPENTRY end_server(USHORT);
void EXPENTRY end_client(USHORT);
/*--- rc=dl_create():バッファの生成(server) ----------*/
USHORT EXPENTRY dl_create() { USHORT rc;
  if(created) return(BUFFER_USED);
  created=TRUE; aborted=FALSE;
            /*  RAMセマフォの初期化          */
  DosSemClear(&full); DosSemSet(&empty);
     /*  取得可能な名前なし共有メモリの作成  */
  rc=DosAllocSeg(SM_SIZE,&sm_sel,SEG_GETTABLE);
  if(rc) { created=FALSE; return(rc); }
  sm = MAKEP(sm_sel,0);  /* ポインタのセット  */
            /*  プロセス終了ルーチンの登録    */
  rc=DosExitList(EXLST_ADD,end_server);
  if(rc) { created=FALSE; DosFreeSeg(sm_sel);
       return(rc); }
  return(0);
}
/*--- rc=dl_receive(msg,&n):メッセージの受信 -------*/
USHORT EXPENTRY
dl_receive(PBYTE msg,int far *n){ int i;
  if(!created) return(NOT_CREATED);
  DosSemWait(&empty,-1L); DosSemSet(&empty);
  if(aborted) { aborted=FALSE;
        return(CLIENT_ABORTED); }
     /*  バッファからメッセージを取り出す    */
  for(i=0;i<msg_len;i++) msg[i]=sm[i];
  *n = msg_len;
  DosSemClear(&full);
  return(0);
}
```

```
/*--- rc=dl_delete():バッファの解放(server) ---------- */
USHORT EXPENTRY dl_delete() {
  created=FALSE;
  DosFreeSeg(sm_sel); /*  共有メモリの解放      */
  DosSemClear(&full); /*  クライアントに通知    */
       /*  プロセス終了ルーチンの登録解除       */
  DosExitList(EXLST_REMOVE,end_server);
  return(0);
}
/*--- rc=dl_open():バッファのオープン(client) ------ */
USHORT EXPENTRY dl_open() { USHORT rc;
  if(!created) return(NOT_CREATED);
  if(connected) return(BUFFER_USED);
  connected=TRUE;
  rc=DosGetSeg(sm_sel); /* セレクタの使用許可  */
  if(rc) { connected=FALSE; return(rc); }
            /*  プロセス終了ルーチンの登録      */
  rc=DosExitList(EXLST_ADD,end_client);
  if(rc) { connected=FALSE; DosFreeSeg(sm_sel);
       return(rc); }
  return(0);
}
/*--- rc=dl_send(msg,n):メッセージの送信 -----------*/
USHORT EXPENTRY dl_send(PBYTE msg,int n){int i;
  if(!created) return(NOT_CREATED);
  if(!connected) return(NOT_OPENED);
  DosSemWait(&full,-1L); DosSemSet(&full);
  if(!created) return(BUFFER_DELETED);
            /*  バッファにメッセージを格納      */
  msg_len=n; for(i=0;i<n;i++) sm[i]=msg[i];
  DosSemClear(&empty);
  return(0);
}
/*--- rc=dl_close():バッファの解放(client) -------------*/
USHORT EXPENTRY dl_close() {
  connected=FALSE;
  DosFreeSeg(sm_sel);   /*  共有メモリの解放    */
       /*  プロセス終了ルーチンの登録解除       */
  DosExitList(EXLST_REMOVE,end_client);
  return(0);
}
/*-- end_server(tc):serverプロセス終了ルーチン -----*/
void EXPENTRY end_server(USHORT tc) {
  if(created) {
    created=FALSE;
    DosFreeSeg(sm_sel);  /* 共有メモリの解放     */
    DosSemClear(&full);  /* クライアントに通知   */
  }
  DosExitList(EXLST_EXIT,0);
}
/*-- end_client(tc):clientプロセス終了ルーチン -*/
void EXPENTRY end_client(USHORT tc) {
  if(connected) {
    connected=FALSE;
    DosFreeSeg(sm_sel);     /* 共有メモリの解放  */
    if(tc!=0/*TC_EXIT*/){  /*正常終了以外の場合*/
      aborted=TRUE;    /* abortedフラグを立てる*/
      DosSemClear(&full);   /* バッファを廃棄    */
      DosSemClear(&empty);/* サーバに通知       */
    }
  }
  DosExitList(EXLST_EXIT,0);
}
```

図-4 名前なし共有メモリと RAM セマフォを用いたプロセス間データ転送用 DLL モジュールの例

に示したように，DosExitList でいつでも登録や解除ができる．また，プロセス終了ルーチン自体は必ず DosExitList で終了することになっている．プロセス終了ルーチンの中では，呼ばれたときの引数の値によって，正常終了したのかそれとも強制終了されたのかを知ることができる (end_client 参照)．

DLL はその名が示すとおり，プログラムの実行開始時などに動的にリンクされるモジュールであるため，図-4 のプログラムは，それを使用するプロセスとは独立に開発することができる．またいったん作成した後に，図-4 のプログラムを書き換えて作り直しても，そのインタフェースが変わらなければ，それを呼んでいるプロセスのほうは，書き換えたりリンクし直す必要がない．このようなことができるのが，DLL の優れた特徴である．読者は，まず図-4 とそれを使用するプロセスを作成した後で，図-4 のツールを N 対 1 のクライアント／サーバ通信用に拡張したり，RAM セマフォのかわりに FS RAM セマフォを使用するなどの演習問題を行ってみると良いだろう．また，インタフェースを変更して，名前とハンドルを導入し，OS/2 が提供している名前つきパイプやキューのような，汎用的なツールに拡張するというのも興味深い演習問題となろう．

(3) OS/2 の名前なしパイプと名前つきパイプ

以上述べたように，DLL を使えばだれでもプロセス間通信用のツールを作ることができるが，DLL のプログラミングは必ずしも容易ではない．そこで OS/2 では，プロセス間の標準的なデータ転送用のツールとしてパイプやキューなどを提供している．

パイプは，共有メモリ上に置かれた FIFO (first-in first-out) 型のバッファで，その入出力に DosRead や DosWrite などのファイル入出力用 API を使用するのが特徴である．パイプは，UNIX オペレーティングシステムで最初に導入された．OS/2 には，UNIX で最初に導入したパイプと同様の名前なしパイプと，OS/2 バージョン 1.1 から導入された名前つきパイプの 2 種類のパイプが提供されている．

名前なしパイプは，

```
rc=DosMakePipe(&hread, &hwrite, SIZE)
```

という API で生成される．ここで SIZE では，共有メモリ上の FIFO バッファの大きさを指定する．その最大値は，共有メモリのセグメントの最大が 64KB であるため，それから制御用の変数などが置かれる 32 バイトのヘッダ領域の分を引いた値である．hread と hwrite は，パイプへの入力および出力用のファイルハンドルである．名前なしパイプのファイルハンドルは，UNIX におけるパイプの利用法にならって，通常は DosDupHandle という API で標準入力や標準出力などに複写し，子プロセスや孫プロセスに引き継がれて利用される．

一方，名前つきパイプは，クライアント／サーバ型のプロセス間のデータ転送に使用されることを想定しており，その名前さえ知っていれば任意のプロセスから利用できる．名前つきパイプについてはなじみが薄いと思われるので，**図-5** にその簡単な使用例を示した．名前つきパイプのサーバ側では，ファイル名と同様の ¥PIPE¥path¥name.ext という形式でパイプ名を指定して，DosMakeNmPipe という API で名前つきパイプを生成し，DosConnectNmPipe という API で，クライアント・プロセスから利用されるのを待つ (np_make ルーチン参照)．一方，クライアント・プロセスでは，通常のファイルとまったく同じように DosOpen で名前つきパイプをオープンして利用する (np_open ルーチン参照)．このとき，まだサーバ側の準備ができていなかったら，DosWaitNmPipe という API を使って待つこともできる．

名前つきパイプでは，データ転送形態として，バイト方式とメッセージ方式の 2 通りが用意されている．バイト方式は，名前なしパイプの場合と同様の方式で，バイト単位に読み書きが行われる．しかしメッセージ方式の場合は，メッセージすなわち一度の DosWrite で書き込まれたデータブロックごとに，DosRead で読み出しが行われる．メッセージ方式の場合，名前つきパイプの中でメッセージの区切りが記憶されている．

また，名前なしパイプでは，hwrite で書き込み hread で取り出すというように，ファイルハンドルのデータの流れは 1 方向であったが，名前つきパイプの場合は，DosMakeNmPipe や DosOpen が返す 1 つのファイルハンドルを，流入のみ，流出のみ，双方向の 3 通りに使用できる．図-5

の例では，1つのファイルハンドルを使って，DosReadで相手プロセスからのメッセージを受信し，DosWriteでそのお返しのメッセージを送るという双方向通信が行われている．さらに，名前つきパイプは，1対1のクライアント／サーバ間だけでなく，複数のクライアントからのデータを受け付けるN対1やN対Mのクライアント／サーバ間のデータ転送にも利用できる．またさらに，LANマネジャのもとでは，¥¥server-¥pipe¥nameというような名前で，別のパソコン上のサーバと通信することも可能である．OS/2の名前つきパイプは，このように，ローカルなプロセス間通信からLAN上でのリモート・プロセス間通信まで，幅広く利用可能な強力なプロセス間通信ツールとなっている．

3.3 プロセス間の共有資源管理の問題

プロセス間の共有資源管理の問題は，前回述べたスレッド間の共有資源に対するアクセスの競合

```
/*--- npserver.c:サーバプロセス ----------*/
/* [MS-Cによる作成] cl npserver.c npserver.def */
/* [モジュール定義ファイルの例は省略]          */
/*-------------------------------------------*/
#include "sample3.h"   /* 例題用ヘッダファイル */
#include "err.sub"     /* errルーチン          */
#define CtlZ    0x1a   /* ファイル終了文字     */
#define MSG_SIZE 81    /* メッセージの最大長+1 */
main() { USHORT nw; int s;
  char msg[MSG_SIZE]; /*  受信メッセージ      */
  np_make(); /* 名前つきパイプの生成と接続待ち */
  while(TRUE) {
    np_receive(msg,&s);     /* メッセージを受信 */
    DosWrite(1,msg,s,&nw); /* 標準出力に出力   */
      /*  ファイル終了文字ならプロセスを終了  */
    if(msg[0]==CtlZ) {
      np_free();          /* 名前つきパイプの解放 */
      DosExit(EXIT_PROCESS,0); /* プロセス終了 */
    }
  }
}
/*== 名前つきパイプの生成と接続待ち ======= */
          /*  名前つきパイプの名前と大きさ    */
#define NP_NAME   "¥¥pipe¥¥npbuffer"
#define NP_SIZE   MSG_SIZE*2
HFILE   hp;             /*  パイプハンドル     */
  /* OPEN_MODE = PIPE_ACCESS_DUPLEX |
     PIPE_NOINHERIT | PIPE_WRITEBEHIND          */
#define OPEN_MODE  0x0082
  /* PIPE_MODE = インスタンス数 + PIPE_WAIT |
     PIPE_READMODE_MESSAGE | PIPE_TYPE_MESSAGE*/
#define PIPE_MODE   1+0x0500
#define TOUT        1000L
np_make() { USHORT rc;
  /*----- 名前つきパイプの作成 ---------- */
  rc=DosMakeNmPipe(NP_NAME,&hp,OPEN_MODE,
        PIPE_MODE,NP_SIZE,NP_SIZE,TOUT);
  if(rc) err("DosMakeNmPipe failed.",rc);
  /*----- クライアントのオープン待ち ----------*/
  rc=DosConnectNmPipe(hp);
  if(rc) err("DosConnectNmPipe failed.",rc);
}
/*== メッセージの受信 ================ */
np_receive(msg,s) char *msg; int *s; {
  USHORT rc, nr, nw;
  rc=DosRead(hp,msg,MSG_SIZE,&nr);  /* 受信  */
  *s=nr;
  if(rc) err("DosRead(msg) failed.",rc);
  rc=DosWrite(hp,"OK",2,&nw); /* OKを応答 */
  if(rc) err("DosWrite(OK) failed.",rc);
}
/*== 名前つきパイプの解放 ==============*/
np_free() { USHORT rc;
  rc=DosClose(hp);
  if(rc) err("DosClose failed.",rc);
}
```

```
/*--- npclient.c:クライアントプロセス ----------*/
/* [MS-Cによる作成] cl npclient.c npclient.def */
/* [モジュール定義ファイルの例は省略]          */
/*-------------------------------------------*/
#include "sample3.h"   /* 例題用ヘッダファイル */
#include "err.sub"     /* errルーチン          */
#define CtlZ    0x1a   /* ファイル終了文字     */
#define MSG_SIZE 81    /* メッセージの最大長+1 */
main() { USHORT nr; int i; char c;
  char msg[MSG_SIZE]; /*  送信メッセージ      */
  np_open();      /*  名前つきパイプのオープン */
  while(TRUE) {  /* 標準入力より1行取り出す   */
    for(i=0;i<MSG_SIZE-1;) {
      DosRead(0,&c,1,&nr);   /* 1文字入力      */
      if(nr!=1) if(i==0) c=CtlZ; else c='¥n';
      msg[i]=c; i++; if(c=='¥n'||nr!=1) break;
    }
    np_send(msg,i);      /*  メッセージの送信   */
    /*  ファイル終了文字によりプロセスを終了  */
    if(msg[0]==CtlZ) {
      np_close(); /*  名前つきパイプのクローズ  */
      DosExit(EXIT_PROCESS,0); /* プロセス終了 */
    }
  }
}
/*== 名前つきパイプのオープン ========== */
                  /* 名前つきパイプの名前  */
#define NP_NAME   "¥¥pipe¥¥npbuffer"
HFILE   hp;             /*  パイプハンドル     */
#define ATTR   0x0000/*FILE_NORMAL(r/w file)*/
#define FLAG   0x0001/*FILE_OPEN(open only) */
  /* MODE = OPEN_ACCESS_READWRITE |
          OPEN_SHARE_DENYNONE                  */
#define MODE  0x0042
np_open() { USHORT rc, r;
  rc=DosOpen(NP_NAME,&hp,&r,0L,
                  ATTR,FLAG,MODE,0L);
  if(rc) err("DosOpen failed.",rc);
}
/*== メッセージの送信 ================ */
np_send(msg,s) char *msg; int s; {
  USHORT rc, nw, nr; char R[MSG_SIZE];
  rc=DosWrite(hp,msg,s,&nw);  /*  送信  */
  if(rc) err("DosWrite(msg) failed.",rc);
  rc=DosRead(hp,R,MSG_SIZE,&nr); /* OK受信 */
  if(rc) err("DosRead(R) failed.",rc);
  if(nr!=2||strncmp(R,"OK",2)) { R[nr]=0;
    printf("np_send: nr=%d R=<%s>¥n",nr,R);
    err("np_send failed.",0);
  }
}
/*== 名前つきパイプの解放 ==============*/
np_close() { USHORT rc;
  rc=DosClose(hp);
  if(rc) err("DosClose failed.",rc);
}
```

図-5　名前つきパイプによるメッセージ転送の例

の問題と同様の問題である．その解決方法としても，共有資源をアクセスする専用のプロセスを1つ用意し，ほかのプロセスはそのプロセス経由で共有資源を利用するという方法と，プロセス間で互いに排他的に制御して共有資源をアクセスするという方法がある．前者の方法は，結局，前節で述べた N 対1のクライアント／サーバ間のメッセージ転送で解決するということにほかならない．したがってここでは，後者の方法のみについて，具体的な問題とそのプログラム例を示すことにする．

(1) 食事をする哲学者の問題

食事をする哲学者の問題 (dining philosophers problem) は，古典的な並行プログラミング問題の中でも非常に有名なものである．これは，「5人の哲学者が，自分の両側のフォークを使ってテーブルのまん中にあるスパゲティを取り食事をすることと，フォークをもとに戻して哲学的な思考にふけるということを繰り返している．食事をするときは左右のフォークを同時に使うが，各人の間にフォークが1本しかない場合でも，互いに協力してだれもが食事ができるようにするにはどのようにしたらよいか」という問題である (図-6 参照)．5人の哲学者を並行プロセスとし，その間のフォークを両側のプロセスの共有資源とすることにより，この問題のプログラムを作成することができる．

この問題は，共有資源管理に関連して発生するデッドロック (deadlock) と飢餓状態 (starvation) という2つの重要な問題を提起している．デッドロックとは，プロセスが，永久に解放されることのない共有資源の解放を待つ状態に陥ることをいう*．また，飢餓状態とは，デッドロックにはなっていないが，ほかのプロセスがいつも先に共有資源を獲得してしまい，いつまでたってもその共有資源を獲得できない状態に陥ることをいう．食事をする哲学者問題において，フォークは両側のプロセスの共有資源であるが，単純に両側のプロセスの間の排他制御だけを行うという解では，5人が一斉に右側のフォークを取り次に左側のフォークを取るという行動を取ったときにデッドロックが発生する．また，もしデッドロックが発生しない解が得られても，両側の哲学者のどちらかがいつも食事をしていて，その間にいる哲学者がいつまでたっても食事できないという飢餓状態が起こるなら，それは満足な解ではない．

まず図-7 に，各哲学者プロセスのプログラムの例を示す．各哲学者は，最初に自分の座席を獲得した後，3秒間以上思考し，次に両側のフォークを取って5秒間食事をし，フォークを戻すという行動を繰り返すプログラムとなっている．ここで最初の座席の獲得は，自分がどの座席に座るかはそのときになってみないと分からないと仮定したために必要となった処理で，この問題では本質的ではない．後のフォークの獲得や解放の処理をどうするかが本質的に重要な問題である．

図-8 は，これらの処理を記述した DLL の例である．最初の座席を獲得する dp_getseat (&s) (s は獲得された座席番号) ルーチンでは，空いている座席 i を探して s に返すとともに，seat [i] に

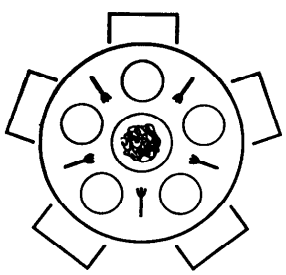


図-6 食事をする哲学者の問題

```
/*--- dpclient.c ----------------------------------------------*/
/* [MS-Cによる作成例]                                          */
/*  cl dpclient.c dpclient.def dptable.lib                     */
/* [モジュール定義ファイル dpclient.defの例]                   */
/*   NAME  DPCLIENT  WINDOWCOMPAT                             */
/*   PROTMODE                                                 */
/*   STACKSIZE 4096                                           */
/*-------------------------------------------------------------*/
#include "sample3.h"   /* 例題用ヘッダファイル  */
USHORT EXPENTRY dp_getseat(int far *);
USHORT EXPENTRY dp_getforks(int);
USHORT EXPENTRY dp_putforks(int);
#define idle(s) DosSleep(s*1000L)   /*  s秒idle  */
main() { int seat;
   if( dp_getseat(&seat) )  /*  座席の獲得      */
     printf("No more seat.¥n");
   else {
     for(;;) {
       printf("%d: Thinking...¥n",seat);  idle(3);
       dp_getforks(seat);   /*  フォークを獲得  */
       printf("%d: Eating.....¥n",seat);  idle(5);
       dp_putforks(seat);   /*  フォークを戻す  */
     }
   }
}
```

図-7 食事をする哲学者問題における哲学者プロセスのプログラム例

* デッドロックの問題については，Peterson と Silberschatz の "Operating System Concepts 2nd Edition", Addison-Wesley, 1985 (宇津宮・福田訳：オペレーティングシステムの概念，培風館，1987) に詳しく解説されている．

```
/*--- dptable.c ----------------------------------------------------*/
/* [MS-Cによるdptable.dllの作成例]                                   */
/*  cl -Asnu -Gs dptable.c dptable.def                               */
/*       -Fedptable.dll doscalls.lib[又はos2.lib]                    */
/* [implibコマンドによるdptable.libの作成例]                         */
/*  implib dptable.lib dptable.def                                   */
/* [モジュール定義ファイル dptabel.defの例]                          */
/*   LIBRARY   DPTABLE                                               */
/*   EXPORTS                                                         */
/*     DP_GETSEAT                                                    */
/*     DP_GETFORKS                                                   */
/*     DP_PUTFORKS                                                   */
/*------------------------------------------------------------------*/
#include "sample3.h"   /* 例題用ヘッダファイル   */
#define SEAT_FULL 2001 /*  エラーコード          */
ULONG mutex=0L;            /* 排他制御用セマフォ*/
ULONG waiting[5];          /* 再開待セマフォ     */
int  fork[5]={0,0,0,0,0};  /* 5本のフォーク      */
int  eating[5]={0,0,0,0,0}; /* 食事している人    */
int  seat[5];  /* 座席（使用中のＰＩＤをセット）  */
int _acrtused = 0; /* crt0ルーチンを取り込まない  */
                /*  プロセス終了ルーチンの宣言    */
void EXPENTRY end_process(USHORT);
/*-- rc=dp_getseat(&s):座席の獲得 ---------------------*/
USHORT EXPENTRY dp_getseat(int far *s) {
   int i; PIDINFO p;
   DosSemRequest(&mutex,-1L); /* 排他制御開始 */
            /*  空いている座席を調べる        */
   for(i=0;i<5;i++) if(seat[i]==0) { *s = i;
    DosGetPID(&p); seat[i]=p.pid; waiting[i]=0L;
            /*  プロセス終了ルーチンを登録    */
    DosExitList(EXLST_ADD,end_process); break;
   }
   DosSemClear(&mutex);        /* 排他制御終了 */
   if(i<5) return(0); else return(SEAT_FULL);
}
```

```
/*--- rc=dp_getforks(s):フォークを獲得 ----------------*/
USHORT EXPENTRY dp_getforks(int s) {
   for(;;) {
    DosSemRequest(&mutex,-1L);/* 排他制御開始 */
    if( !fork[s] && !fork[(s+1)%5] ) {
     eating[s]=fork[s]=fork[(s+1)%5]=TRUE;
    }
    DosSemClear(&mutex);        /* 排他制御終了 */
             /*  フォークが獲得できたら戻る   */
    if(eating[s]) return(0);
             /*  フォークが戻されるまで待つ   */
    DosSemSetWait(&waiting[s],-1L);
   }
}
/*--- rc=dp_putforks(s):フォークを戻す ----------------*/
USHORT EXPENTRY dp_putforks(int s) {
   DosSemRequest(&mutex,-1L); /* 排他制御開始 */
    eating[s]=fork[s]=fork[(s+1)%5]=FALSE;
   DosSemClear(&mutex);        /* 排他制御終了 */
             /*  両隣のプロセスを再開          */
   DosSemClear(&waiting[(s+1)%5]);
   DosSemClear(&waiting[(s+4)%5]);
}
/*-- end_process(tc):プロセス終了ルーチン ----------*/
void EXPENTRY end_process(USHORT tc) {
   int i; PIDINFO p;
   DosGetPID(&p);
   for(i=0;i<5;i++) if(seat[i]==p.pid) {
             /* 食事中であったらフォークを戻す */
    if(eating[i]) dp_putforks(i);
    seat[i]=0; /* 座席を解放 */
    break;
   }
   DosExitList(EXLST_EXIT,0);
}
```

図-8 食事をする哲学者問題を解く DLL のプログラム例

自分の PID を保存し，プロセス終了ルーチン end_process を登録する．プロセスがコントロールCなどで終了すると，この end_process ルーチンが呼ばれて食事中かどうかを調べ，食事中ならフォークを戻し，使用中の座席を解放するという処理を行う．dp_getseat で自分の PID を seat に記憶しているのはこの終了処理のためである．フォークを獲得する dp_getforks (s) (s は自分の座席番号) ルーチンでは，自分の両側のフォークが空くまで waiting [s] セマフォで待つ．一方，フォークを戻す dp_putforks(s) ルーチンでは，フォークを戻した後，セマフォ waiting [s−1] と waiting [s+1] をクリアし，両隣のプロセスを再開する．

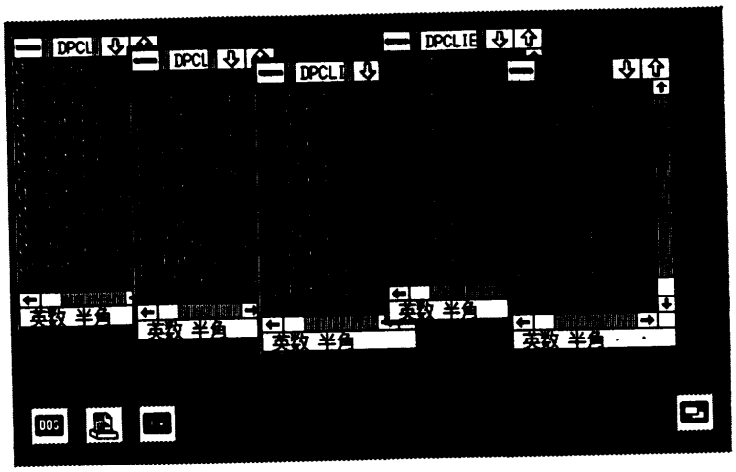


図-9 図-7 の哲学者プロセスを５つの別々のウィンドウ・セッション上で実行したときの例

図-9 は，５つのウィンドウ・セッションを開いて，各セッションで哲学者プロセスを実行させたときの例である．これにより，デッドロックや飢餓状態が起こっていないことを，一目で見ることができる．通常の，共有資源管理の場合は，食事をする哲学者問題のように難しい条件は少なく，単純な排他制御や，せいぜい前回述べた読み取り／書き込み問題の解のようなもので済むであ

```
/*-- sample3.h:例題で使用したヘッダファイル -------*/
#include "sample2.h"  /*  前回のヘッダファイル */
          /*-- 使用するデータタイプの定義 --------*/
typedef void   far * PVOID;
typedef USHORT PID;          /* プロセスＩＤ      */
typedef PID    far * PPID;   /* そのポインタ      */
typedef char   far * PSZ;    /* 文字列            */
#define HSYSSEM HSEM /* システムセマフォハンドル */
typedef HSYSSEM far * PHSYSSEM;
typedef unsigned short SEL;  /* セレクタ          */
typedef SEL    far * PSEL;   /* そのポインタ      */
typedef struct _RESULTCODES {
   USHORT codeTerminate;     /* 終了コード/PID    */
   USHORT codeResult;        /* 終了コード        */
} RESULTCODES;
typedef RESULTCODES far * PRESULTCODES;
typedef struct _PIDINFO {
   PID pid; TID tid; PID pidParent; } PIDINFO;
typedef PIDINFO far * PPIDINFO;
typedef void (pascal far * PFNEXITLIST)(USHORT);
typedef unsigned short HFILE;
typedef HFILE  far * PHFILE;
          /*-- 使用するマクロの定義 ---------------- */
#define MAKEULONG(l,h) ¥
 ((ULONG)(((USHORT)(l))|((ULONG)((USHORT)(h)))<<16))
#define MAKEP(sel,off) ((PVOID)MAKEULONG(off,sel))
          /*-- 使用するＡＰＩの定義 ---------------- */
#define EXPENTRY APIENTRY
#define CSEM_PUBLIC          1
USHORT APIENTRY DosCreateSem(USHORT,
                             PHSYSSEM,PSZ);
USHORT APIENTRY DosOpenSem(PHSYSSEM,PSZ);
USHORT APIENTRY DosCloseSem(HSYSSEM);
#define SEG_GIVEABLE         0x0001
#define SEG_GETTABLE         0x0002
USHORT APIENTRY DosAllocSeg(USHORT,PSEL,USHORT);
USHORT APIENTRY DosFreeSeg(SEL);
USHORT APIENTRY DosGiveSeg(SEL,PID,PSEL);
USHORT APIENTRY DosGetSeg(SEL);
USHORT APIENTRY DosAllocShrSeg(USHORT,PSZ,PSEL);
USHORT APIENTRY DosGetShrSeg(PSZ,PSEL);
#define EXEC_ASYNC           1
USHORT APIENTRY DosExecPgm(PBYTE,USHORT,
        USHORT,PSZ,PSZ,PRESULTCODES,PSZ);
#define EXLST_ADD            1
#define EXLST_REMOVE         2
#define EXLST_EXIT           3
USHORT APIENTRY DosExitList(USHORT,PFNEXITLIST);
USHORT APIENTRY DosGetPID(PPIDINFO);
USHORT APIENTRY DosMakeNmPipe(PSZ,PHFILE,
        USHORT,USHORT,USHORT,USHORT,ULONG);
USHORT APIENTRY DosConnectNmPipe(HFILE);
USHORT APIENTRY DosOpen(PSZ,PHFILE,PUSHORT,
        ULONG,USHORT,USHORT,USHORT,ULONG);
USHORT APIENTRY DosClose(HFILE);
```

図-10 今回の例題で使用したヘッダファイル

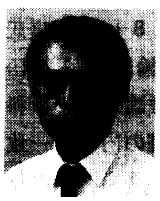
ろう．しかし，食事をする哲学者問題に対していろいろな解を作成してみることにより，共有資源管理の問題やデッドロック，飢餓状態の問題の理解が深まるものと思われる．読者は，実際にデッドロックを起こさせて，さらにそれを検出するというような演習問題に挑戦してみると良いだろう．

なお，共有資源に対するアクセスを管理する場合は，ここでの例のように，そのアクセス・ルーチンを DLL で用意し，プロセスは常にそのアクセス・ルーチンを介してのみ共有資源をアクセスするようにプログラミングするのが良い．こうすることにより，共有資源に対するアクセスの管理が確実となり，信頼性の高い並行プログラムを作成することができる．また，共有資源管理に対する具体的な管理方法が DLL の中にのみ記述されるので，管理方法の改善や変更も容易となる．

3.4 ま と め

今回は，マルチプロセス・プログラムの作成に関連して，OS/2 におけるプロセスの生成の方法について解説した後，プロセス間での共有メモリとセマフォを使ったデータ転送の実現方法，DLL によりモジュール化してプロセス間通信ツールを実現する方法，共有資源管理の方法とデッドロックや飢餓状態の問題などについて解説した．本講座の第1回で紹介したプロセス間通信機構のうち，キューとシグナルについては，今回その具体的な使用法を示すことができなかったが，これらは今回紹介したものに比べると使われるケースもそれほど多くないと思われるので省略する．なお，前回にも述べたが，OS/2 用に提供されているヘッダファイル os2.h はバージョンによって若干異なるため，正確を期すため今回の例題でも os2.h の代わりに**図-10** の sample3.h を用いた．また，32ビットメモリをサポートしている OS/2 バージョン 2.0 以上では，共有メモリやセマフォの仕様が変更されているので，ここで示した例題は書き換える必要があろう．

（平成2年12月27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．