オペレーティングシステム
第６章 プロセス間通信

https://github.com/tctsigemura/OSTextBook

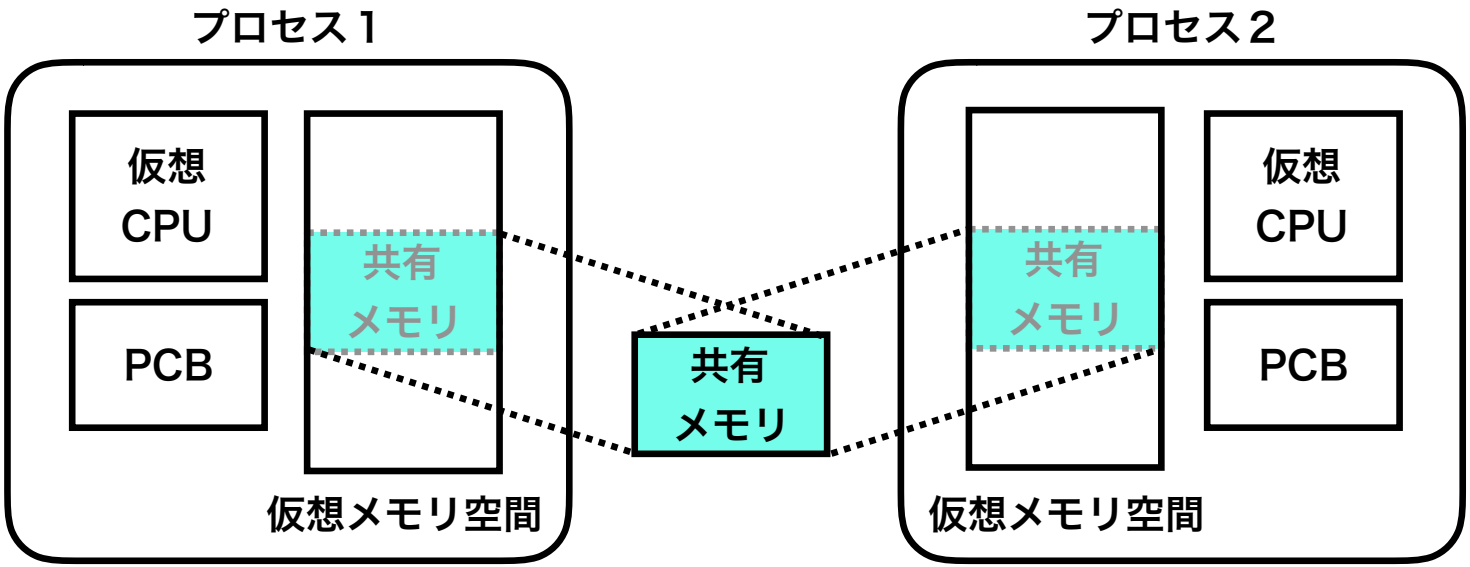
プロセス間通信の必要性

プロセス間通信（IPC：Inter-Process Communication）
複数のプロセスが情報を共有し協調して処理を進めることができる．
次のようなメリットが期待できる．

- 複数のプロセスが共通の情報へアクセスすることができる．
- 並列処理による処理の高速化ができる可能性がある．
- システムを見通しの良いモジュール化された構造で構築できる．

プロセス間で情報を共有する代表的な機構として，**共有メモリ**と**メッセージ通信**がある．

共有メモリ



- 同じ物理メモリを複数のプロセスの仮想メモリ空間に貼り付ける.
- MMU（Memory Management Unit）の働きで可能になる.
- 貼り付けが終わればシステムコールなしでデータ交換可能.
- プロセス間の同期機構は他に必要.

UNIX の共有メモリシステムコール等（前半）

共有メモリなどの識別に使用するキーを生成(ライブラリ)

```
key_t ftok(const char *path, int id);
```

返り値 : 引数から作成されるキー値
`path` : 実際に存在するファイルのパス
`id` : キーの作成に使用する追加の情報(同じ `path` から異なるキーを作る)

共有メモリセグメントの作成(システムコール)

```
int shmget(key_t key, size_t size, int flag);
```

返り値 : 共有メモリセグメント ID
`key` : キー
`size` : セグメントサイズ(バイト単位)
`flag` : 作成フラグとモード

- ftok() は，path と id から一意な key 値を生成する.
- shmget() は，key 値で識別されるメモリセグメント ID を返す.
- shmget() は，メモリセグメントを作ることもできる.
- flag は，rwxrwxrwx と IPC_CREAT 等のフラグ

UNIX の共有メモリシステムコール等（後半）

共有メモリセグメントをプロセスの仮想アドレス空間に貼り付ける(システムコール)

```
void *shmat(int shmId, void *addr, int flag);
```

返り値 : 共有メモリセグメントを配置したアドレス
shmId : 共有メモリセグメント ID
addr : 貼り付けるアドレス(NULL(0) は, カーネルに任せる)
flag : 貼り付け方法等

共有メモリセグメントをプロセスの仮想アドレス空間から取り除く(システムコール)

```
int shmdt(void *addr);
```

返り値 : 0=正常, -1=エラー
addr : 取り除く共有メモリセグメントのアドレス

共有メモリセグメントの制御(システムコール)

```
int shmctl(int shmId, int cmd, struct shmid_ds *buf);
```

返り値 : 0=正常, -1=エラー
shmId : 共有メモリセグメント ID
cmd : 削除 (IPC_RMID) 等のコマンド
buf : コマンドのパラメータ

UNIX の共有メモリサーバ例（前半）

```
 1  // 共有メモリサーバ(ipcUnixSharedMemoryServer.c) :共有メモリからデータを読みだし表示する
 2  #include <stdio.h>
 3  #include <stdlib.h>
 4  #include <string.h>
 5  #include <unistd.h>
 6  #include <sys/types.h>
 7  #include <sys/ipc.h>
 8  #include <sys/shm.h>
 9  #define SHMSZ     512                                  // 共有メモリのサイズ
10  int main() {
11    key_t key=ftok("shm.dat",'R');                       // キーを作る
12    if (key==-1) {                                       // エラーチェック
13      perror("shm.dat");
14      exit(1);
15    }
16    int shmid=shmget(key,SHMSZ,IPC_CREAT|0666);          // 共有メモリを作る
17    if (shmid<0) {                                       // エラーチェック
18      perror("shmget");
19      exit(1);
20    }
```

UNIX の共有メモリサーバ例（後半）

```
21    char *data=shmat(shmid,NULL,0);                  // 共有メモリを貼り付ける
22    if (data==(char *)-1) {                          // エラーチェック
23      perror("shmat");
24      exit(1);
25    }
26    strcpy(data, "initialization...\n");             // 共有メモリに書き込む
27    while(1) {                                       // 共有メモリの内容を
28      printf("sheared memory:%s",data);              //    5 秒に 1 度メモリを表示
29      if (strcmp(data, "end\n") == 0) break;         //    "end"なら終了
30      sleep(5);
31    }
32    if (shmdt(data) == -1){                          // 共有メモリをアドレス空間
33      perror("shmdt");                               //    と切り離す
34      exit(1);
35    }
36    if (shmctl(shmid, IPC_RMID, 0) == -1){           // 共有メモリを廃棄する
37      perror("shmctl");
38      exit(1);
39    }
40    return 0;
41  }
```

UNIX の共有メモリクライアント例（前半）

```
1  // 共有メモリクライアント(ipcUnixSharedMemoryClient.c):共有メモリにデータを書き込む
2  #include <stdio.h>
3  #include <stdlib.h>
4  #include <errno.h>
5  #include <sys/types.h>
6  #include <sys/ipc.h>
7  #include <sys/shm.h>
8  #define SHMSZ     512                          // メモリのサイズ
9  int main() {
10   int      shmid;
11   key_t    key;
12   char    *data, *s;
13   if ((key=ftok("shm.dat",'R')) == -1) {  // サーバ側と同じキーを作る
14     perror("shm.dat");
15     exit(1);
16   }
```

UNIX の共有メモリクライアント例（後半）

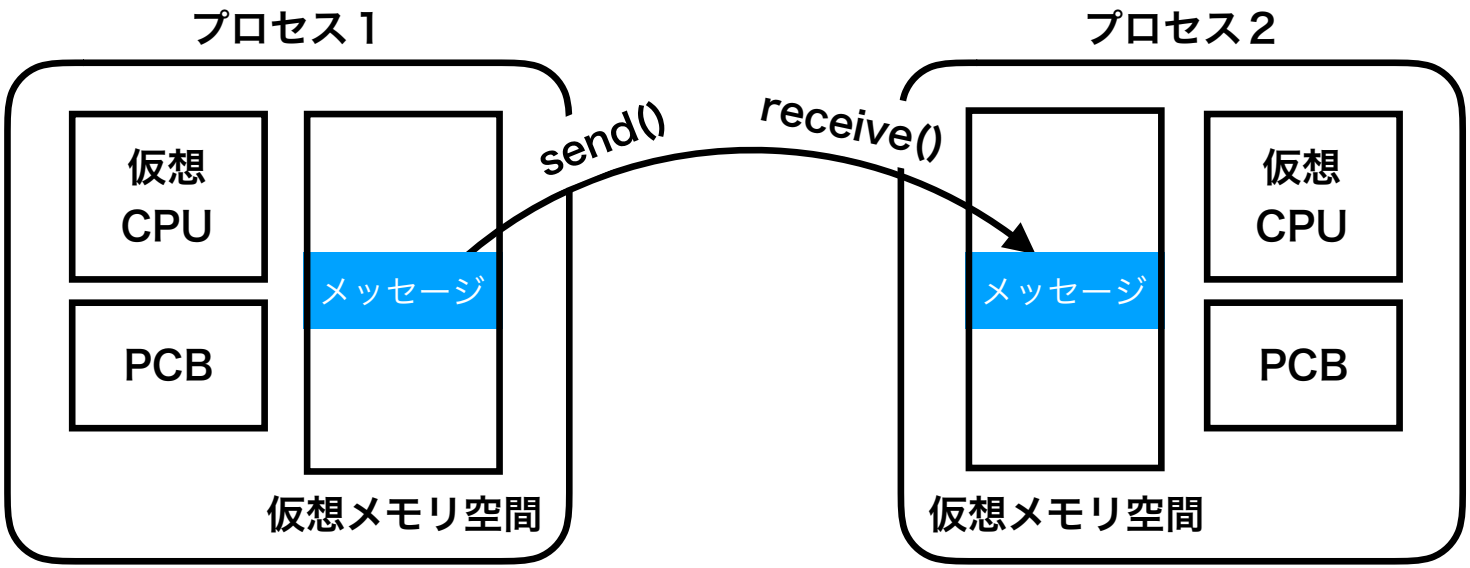
```
17    if ((shmid=shmget(key,SHMSZ,0666))<0) { // 共有メモリを取得する
18      perror("shmget");
19      exit(1);
20    }
21    data=shmat(shmid,NULL,0);               // 共有メモリを貼り付ける
22    if (data == (char *)-1) {               // エラーチェック
23      perror("shmat");
24      exit(1);
25    }
26    printf("Enter a string: ");
27    fgets(data,SHMSZ,stdin);                // 共有メモリに直接入力する
28    if (shmdt(data)==-1){                   // 共有メモリをメモリ空間と
29      perror("shmdt");                      //    切り離す
30      exit(1);
31    }
32    return 0;
33  }
```

UNIX の共有メモリプログラム実行例

```
[Terminal No.1]                          | [Terminal No.2]
$ ./ipcUnixShearedMemoryServer           |
sheared memory:initialization...         | $ ./ipcUnixShearedMemoryClient
sheared memory:initialization...         | Enter a string: abcdefg
sheared memory:abcdefg                   |
sheared memory:abcdefg                   | $ ./ipcUnixShearedMemoryClient
sheared memory:abcdefg                   | Enter a string: 1234567
sheared memory:1234567                   | $ ./ipcUnixShearedMemoryClient
sheared memory:1234567                   | Enter a string: end
sheared memory:end                       |
$                                        | $
```

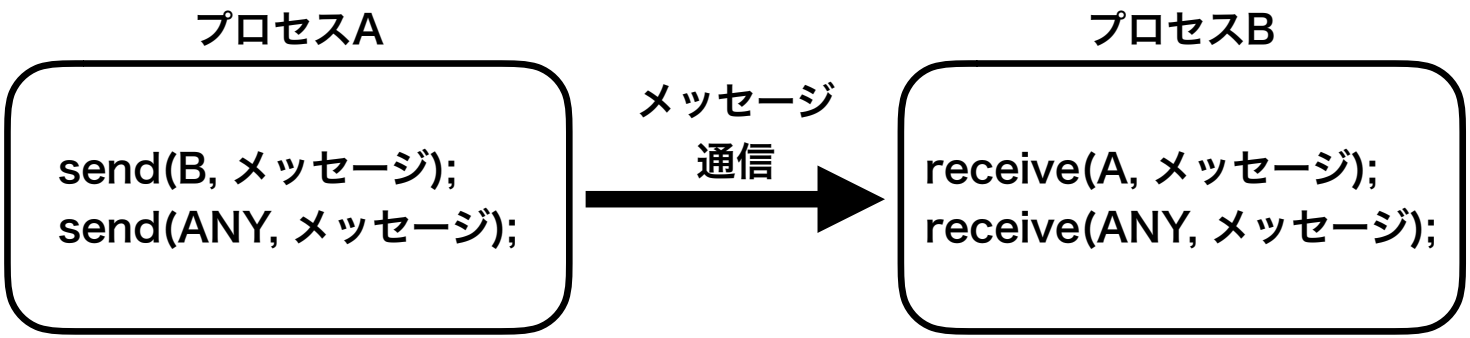
- このプログラムは相互排除をやっていない.
- このプログラムは**使用してはならない**.

メッセージ通信

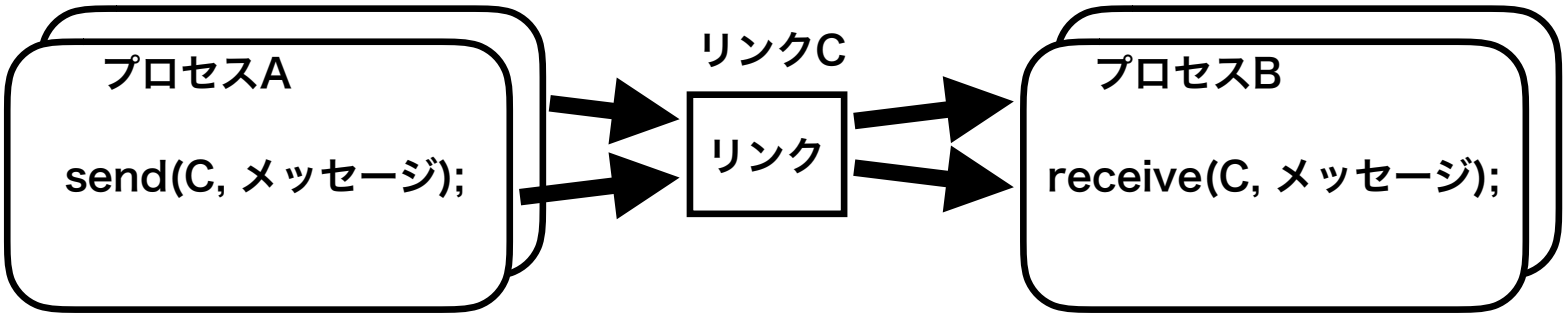


- `send()` システムコールでメッセージを送信する.
- `receive()` システムコールでメッセージを受信する.
- メッセージ通信は同期機構も含んでいる.

通信相手の指定方式（Naming）



直接指定方式



間接指定方式

- **直接指定方式**でも ANY を用いることで多対多通信が可能.
- **間接指定方式**は自然に多対多通信が可能

通信方式

一般に

- バッファリング（あり／なし）
- メッセージ長（固定／可変）
- メッセージ形式（タグあり／なし）
- 同期方式
  - 非同期方式（ノンブロッキング）
  - 同期方式（ブロッキング）
  - ランデブー方式（クライアント・サーバモデルに特化）

UNIX の場合は

- 間接指定方式
- バッファリング＝あり
- メッセージ長＝可変長
- メッセージ形式＝タグあり
- 同期方式／非同期方式どちらも可能

UNIX のメッセージ通信システムコールなど（前半）

メッセージ構造体(以下の構造体を自分で宣言して使用する)

```
struct msgbuf {
    long mtype;       // メッセージの型
    char mtext[N];    // メッセージの本体(N バイト)
};
```

メッセージキューの ID を返す.

`int msgget(key_t key, int msgflg);`(システムコール)

- 返り値 : メッセージキュー ID
- key : キー(ftok() で作成したもの)
- msgflg : IPC_CREAT 等のフラグとアクセス許可ビット

- mtype がタグの役割を持つ.
- key は共有メモリで紹介したものと同じ（ftok() 関数で作る).
- 「メッセージキュー」＝「リンク」
- msgsnd(), msgrcv() で msgflg に IPC_NOWAIT を指定すると**非同期**.

UNIX のメッセージ通信システムコールなど（後半）

```
int msgsnd(int msqid, const void *msgp, size_t msgsz, int msgflg);
  返り値 : 0=正常,-1=エラー
  msqid  : メッセージキュー ID
  msgp   : メッセージ構造体のポインタ
  msgsz  : メッセージ本体のバイト数
  msgflg : IPC_NOWAIT 等のフラグ

メッセージキューからメッセージを受信する(システムコール)
int msgrcv(int msqid, const void *msgp, size_t msgsz, long msgtyp, int msgflg);
  返り値 : -1=エラー、受信したメッセージの本体バイト数
  msqid  : メッセージキュー ID
  msgp   : メッセージ構造体のポインタ
  msgsz  : メッセージ本体の最大バイト数
  msgtyp : 受信するメッセージの型
  msgflg : IPC_NOWAIT 等のフラグ

メッセージキューの制御(システムコール)
int msgctl(int msqid, int cmd, struct msgid_ds *buf);
  返り値 : -1=エラー、0<=コマンドにより異なる
  msqid  : メッセージキュー ID
  cmd    : 削除(IPC_RMID)等のコマンド
  buf    : コマンドのパラメータ
```

UNIX のメッセージ通信プログラム例 1

```
1  // ipcUnixMessage.h : メッセージ構造体の宣言
2  #define MAXMSG 100                          // メッセージ本体の長さ
3  struct msgBuf {                             // メッセージ格納用構造体
4    long mtype;                               //   メッセージの型
5    char mtext[MAXMSG];                       //   メッセージの本体
6  };
```

```
1  // メッセージ送信プログラム(ipcUnixMessageWriter.c):メッセージキューを作成し送信する
2  #include <stdio.h>
3  #include <stdlib.h>
4  #include <sys/types.h>
5  #include <sys/ipc.h>
6  #include <sys/msg.h>
7  #include "ipcUnixMessage.h"                 // msgBuf 構造体の宣言
8  int main() {
9    struct msgBuf buf;                        // メッセージ領域
10   int msqid;                                // メッセージキュー ID
11   key_t key;                                // メッセージキューの名前
12   if ((key=ftok("msgq.dat",'b'))==-1) {     // ftok はファイル名から
13     perror("ftok");                         //  重複のない名前 (キー) を
14     exit(1);                                //   生成する
15   }
```

UNIX のメッセージ通信プログラム例２

```
16    if ((msqid=msgget(key,0644|IPC_CREAT))==-1) { // メッセージキューを作る
17      perror("msgget");
18      exit(1);
19    }
20    printf("Enter lines of text, ^D to quit:\n");
21    buf.mtype = 1;                                // メッセージの型
22    while (fgets(buf.mtext,MAXMSG,stdin)!=NULL) { // キーボードから 1 行入力
23      if (msgsnd(msqid,&buf,MAXMSG,0)==-1) {      // メッセージを送信
24        perror("msgsnd");
25        break;
26      }
27    }
28    if (msgctl(msqid,IPC_RMID,NULL) == -1) {      // メッセージキューを削除
29      perror("msgctl");
30      exit(1);
31    }
32    exit(0);
33  }
```

UNIX のメッセージ通信プログラム例３

```
 1  // メッセージ受信プログラム(ipcUnixMessageReader):メッセージキューから受信する
 2  #include <stdio.h>
 3  #include <stdlib.h>
 4  #include <sys/types.h>
 5  #include <sys/ipc.h>
 6  #include <sys/msg.h>
 7  #include "ipcUnixMessage.h"
 8  int main() {
 9    struct msgBuf buf;
10    int msqid;
11    key_t key;
12    if ((key=ftok("msgq.dat",'b'))==-1) {        // 送信側と同じキーを作る
13      perror("ftok");
14      exit(1);
15    }
16    if ((msqid=msgget(key,0644))==-1) {          // ipcUnixMessageReader が作った
17      perror("msgget");                          //  メッセージキューを取得
18      exit(1);
19    }
```

UNIX のメッセージ通信プログラム例 4

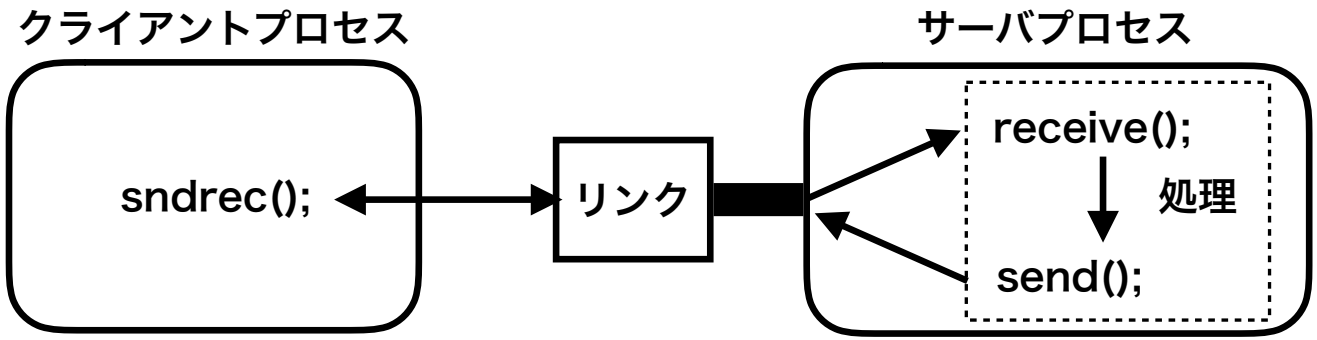
```
20    printf("ready to receive messages.\n");
21    for(;;) {
22      if (msgrcv(msqid,&buf,MAXMSG,0,0)==-1) {  // 先頭のメッセージを読み出す
23        perror("msgrcv");                       // メッセージキューが削除され
24        exit(1);                                //   エラーが発生したら終了
25      }
26      printf("%ld:%s",buf.mtype,buf.mtext);     // 受信したメッセージを表示
27    }
28    exit(0);
29  }
```

```
[Terminal No.1]                          | [Terminal No.2]
$ ./ipcUnixMessageWriter                 | $ ./ipcUnixMessageReader
Enter lines of text, ^D to quit:         | ready to receive messages.
abcdefg                                  | 1:abcdefg
1234567                                  | 1:1234567
^D                                       | msgrcv: Identifier removed
$                                        | $
```

第２１章
TacOSのメッセージ通信

TacOS のメッセージ通信機構

クライアント・サーバモデルに特化した**ランデブー方式**のプロセス間通信機構を提供する.



1. サーバプロセスが**リンク**を所有し通信を待ち受ける.
2. クライアントプロセスは `sndrec()` 関数でメッセージを送信する.
3. サーバプロセスは `receive()` 関数を用いてメッセージを受信する.
4. サーバプロセスはメッセージの内容に合った処理を行う.
5. サーバプロセスは処理結果を `send()` 関数を用いて返信する.
6. `sndrec()` 関数が完了しクライアントは処理結果を受取る.

TacOS のリンク構造体

```
1  #define LINK_MAX 5          // リンクは最大 5 個
2
3  struct Link {               // リンクを表す構造体
4    PCB server;               // リンクを所持するサーバ
5    PCB client;               // リンクを使用中のクライアント
6    int s1;                   // サーバがメッセージ受信待ちに使用するセマフォ
7    int s2;                   // クライアント同士が相互排除に使用するセマフォ
8    int s3;                   // クライアントがメッセージ返信待ちに使用するセマフォ
9    int op;                   // メッセージの種類
10   int prm1;                 // メッセージのパラメータ 1
11   int prm2;                 // メッセージのパラメータ 2
12   int prm3;                 // メッセージのパラメータ 3
13 };
```

- リンクはサーバが所有する.
- セマフォを用いて相互排除と同期を行う.
- リンクに書き込めるメッセージの形式は固定.

TacOS のリンク作成ルーチン

```
1  Link[] linkTbl = array(LINK_MAX);                    // リンクの一覧表
2  int linkID = -1;                                     // リンクの通し番号
3
4  // リンクを生成する(サーバが実行する)
5  public int newLink() {
6    int r = setPri(DI|KERN);                           // 割り込み禁止、カーネル
7    linkID = linkID + 1;                               // 通し番号を進める
8  #ifdef DEBUG
9    printF("newLink:ID=%d,SERVER=%d\n",linkID,curProc.pid);
10 #endif
11   if (linkID >= LINK_MAX)                            // リンクが多すぎる
12     panic("newLink");
13
14   Link l = linkTbl[linkID];                          // 新しく割り当てるリンク
15   l.server = curProc;                                // リンクの所有者を記憶
16   l.s1 = newSem(0);                                  // server が受信待ちに使用
17   l.s2 = newSem(1);                                  // client が相互排他に使用
18   l.s3 = newSem(0);                                  // client が返信待ちに使用
19   setPri(r);                                         // 割り込み復元
20   return linkID;                                     // 割当てたリンクの番号
21 }
```

- 割込み禁止による相互排除を行っている.

TacOS のメッセージ通信ルーチン（サーバ用）

```
1  // サーバ用の待ち受けルーチン
2  public Link receive(int num) {
3    Link l = linkTbl[num];
4    if (l.server != curProc) panic("receive");        // 登録されたサーバではない
5    semP(l.s1);                                       // サーバをブロック
6    return l;
7  }
8
9  // サーバ用の送信ルーチン
10 public void send(int num, int res) {
11   Link l = linkTbl[num];
12   if (l.server != curProc) panic("send");           // 登録されたサーバではない
13   l.op = res;                                       // 処理結果を書込む
14   semV(l.s3);                                       // クライアントを起こす
15 }
```

- receive() はリンクにメッセージが届くのを待つ.
- receive() はメッセージが書き込まれたリンクを返す.
- send() はリンクに処理結果（16bit）を返信する.

TacOS のメッセージ通信使用例（サーバ側）

```
1  // プロセスマネージャサーバのメインルーチン
2  public void pmMain() {
3    pmLink = newLink();                                    // リンクを生成する
4    while (true) {                                         // システムコールを待つ
5      Link l = receive(pmLink);                            // システムコールを受信
6      int r=pmSysCall(l.op,l.prm1,l.prm2,l.prm3,l.client);// システムコール実行
7      send(pmLink, r);                                     // 結果を返す
8    }
9  }
```

- プロセスマネージャ（サーバプロセス）の例.
- サーバはリンクを作成した後，受信，処理，返信を繰り返す.
- receive() を用いてリンクからメッセージを受信.
- pmSysCall() がプロセスマネージャの処理ルーチン.
- send() を用いてクライアントに処理結果を返す.

TacOS のメッセージ通信ルーチン（クライアント用）

```
1  // クライアント用メッセージ送受信ルーチン
2  public int sndrec(int num, int op, int prm1, int prm2, int prm3) {
3    Link l = linkTbl[num];                          // 他のクライアントと相互
4    semP(l.s2);                                     //   排除しリンクを確保
5    l.client = curProc;                             // リンク使用中プロセス記録
6    l.op = op;                                      // メッセージを書込む
7    l.prm1 = prm1;
8    l.prm2 = prm2;
9    l.prm3 = prm3;
10   int r = setPri(DI|KERN);                        // 割り込み禁止､カーネル
11   iSemV(l.s1);                                    // サーバを起こす
12   semP(l.s3);                                     // 返信があるまでブロック
13   setPri(r);                                      // 割り込み復元
14   int res = l.op;                                 // 返信を取り出す
15   semV(l.s2);                                     // リンクを解放
16   return res;
17 }
```

- iSemv() を使用するので割込み禁止による相互排除が必要.
- クライアント間での相互排除にセマフォ s2 を使用.

TacOS のメッセージ通信使用例（クライアント側）

```
1 public int exec(char[] path, char[][] argv) {
2   int r=sndrec(pmLink,EXEC,_AtoI(path),_AtoI(argv),0);
3   return r;                                          // 新しい子の PID を返す
4 }
```

- exec システムコールを例にする.
- exec は path のプログラムを新しいプロセスで実行する.
- 上のプログラムは SVC 割込みハンドラから呼出される.
- SVC 割込みハンドラはシステムコールの種類を判断し，exec システムコールの場合に上のルーチンを呼出す.
- 割込みハンドラはプロセスのコンテキストで実行される.
- exec システムコールはプロセスマネージャが処理する.
- プロセスマネージャへのメッセージ通信により処理を依頼する.
- _AtoI() 関数は参照（アドレス）を int 型に変換する.

練習問題

練習問題（1）

- 次の言葉の意味を説明しなさい.
  - 共有メモリ
  - メッセージ通信
  - 直接指定方式
  - 間接指定方式
  - バッファリング
  - 同期方式
  - 非同期方式
  - ランデブー方式
  - メッセージのタグ

練習問題（2）

- プロセス間の共有メモリとスレッド間の共有変数の違いは何か？
- UNIX の共有メモリ使用例を実際に実行し動作確認しなさい．
- 動作確認したプログラムでは，サーバプログラムは共有メモリが変更されたことを確認しないで，一定の時間間隔で共有メモリの内容を表示している．
  - どのような不都合が予想されるか？
  - クライアントとサーバで同期をする方法はあるか？
- メッセージ通信でバッファを大きくすることのメリットは何か？
- UNIX のメッセージ通信プログラム例を実際に実行し動作確認しなさい．
- UNIX のメッセージ通信プログラム例は生産者と消費者の問題の解になっている．複数生産者と複数消費者の問題の解にもなっているか？
- UNIX のメッセージ通信プログラム例が複数生産者と複数消費者の問題の解にもなっているか，動作確認する手順を説明しなさい．