■7 群（コンピュータ -ソフトウェア）－3 編（オペレーティングシステム）

7 章 プロセス間通信

（執筆者：吉澤康文）［2013 年 2 月 受領］

■概要■

4 章ではプロセス管理の基本的な説明を行ったが，本章では一つの処理を複数のプロセスで並列実行するマルチタスキングを中心に説明する．複数のプロセスが相互に協力し処理を誤りなく実行し，かつそれらの処理を高効率かつ高信頼に実現するには，プロセス間の通信機能が重要となる．ここでも UNIX をケーススタディにとりあげて，プロセス間通信と排他制御方式について説明する．

【本章の構成】

最初にマルチタスキングを実現するために通信機能が不可欠であることを解説する．ここにプロセス間で（情報などの）資源共有を行う必要性を示すと共に，問題点の提起を行う（7-1 節）．並列処理での資源共有に関する問題を解決するためにハードウェアならびに OS が考慮しなくてはならない基本的な機能を各種説明する（7-2 節）．これらの工夫として，排他制御が OS には用意されていることを示し，この機能を使うことでプログラマは高信頼な並列処理のプログラミングが可能になることを示す（7-3 節）．より具体的に理解を深めるために，セマフォの説明，そしてデッドロック問題回避を示す（7-4 節）．そこで UNIX での具体的なプロセス間通信と排他制御の機能を解説することにより，情報システムの設計に必要とされる並列処理の基本を身につけることができるようにする（7-5 節）．

■7群 － 3編 － 7章

7-1 基本的な考え方

（執筆者：吉澤康文）［2013年2月 受領］

7-1-1 プロセス間通信の必要性

複数のプロセスが共同して作業するにはプロセス間の通信が必須となる．これは我々が共同作業する場合に相互の連絡手段をもつのと全く同じである．例えば**図 7･1** に示すように，複数のプロセスで処理を進めるには，あるプロセスが仕事を完了したことを知らせる必要がある．図 7･1 の右のように複数のプロセスからの完了通知を受けた後に次の処理が始まる場合もある．

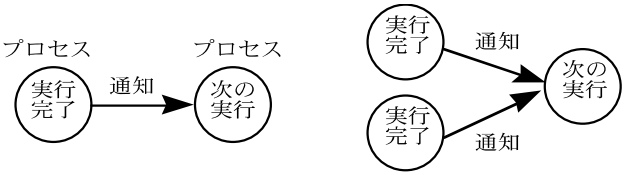

図 7・1　プロセス間通信による処理の進行

このようにプロセス間の通信は並列処理の最も基本的な機能であるため，OS がその手段を提供する必要がある．それをプロセス間通信（IPC：Inter Process Communication）と呼び，OS では同期制御（Synchronization）と呼ぶこともある．

7-1-2 共有資源の利用

複数のプロセスで仕事を進める際にもう一つ重要な機能を必要とする．複数のプロセスで処理を分担しているとしても，データを相互に共有して処理を進めることが多い．このとき，共有しているデータを同時に更新するという競合問題が発生する．**図 7･2** では二つの独立したプロセスが共有データを何の規則もなく更新する様子を示している．

図 7・2　プロセス間共用データの同時更新

図 7･2 において，プロセス A が一度ディスパッチされてプロセスの実行が完了するまでプロセス切換えが起こらなければ問題は生じない．しかし，現代の OS ではプロセスの切換えが任意の時点で発生すると考えた方がよい．そこで，プロセス切換えが発生したときに矛盾を引き起こす例を**図 7･3** に示す．

この例では，全く同一の処理を二つのプロセスが実行している．例えば，預金の引出し業務などを想定すると理解しやすい．プロセス A がまず共有データを読込む（①）．このとき

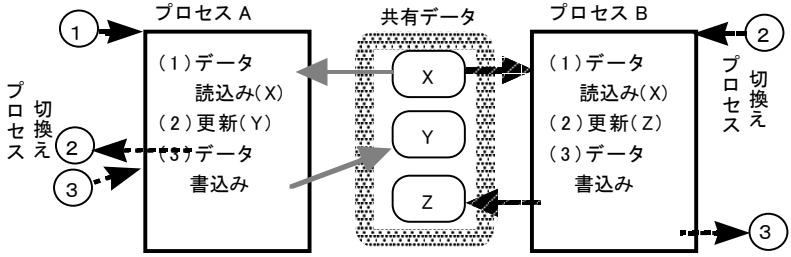

図 7・3　共用データの同時更新時に生じる不都合

の値を X とする．つまり，処理開始の最初の時点の預金残高である．そこで処理（例えば預金の引出し）を進めて結果 Y を書き込もうとしたときにプロセス切換えが起きたとする(②)．そこで，次にプロセス B が実行されデータを読込む．まだプロセス A による書込みが済んでないので値は最初の X のままである．ここでプロセス B は処理（例えば預金の積立て）を行い，計算結果が Z になり書込みを完了する．実行部分が残ったプロセス A が再開される(③)．つまり，プロセス A が求めた計算結果（預金の引き出した結果）である Y の値を書き込むことになる．

上記の結果として，明らかに矛盾が生じている．つまり，プロセス B の計算結果は全く無視されてしまい，積立てた金額は反映されなくなっている．

このようにプロセスが任意の時点で切り換えられたとしても矛盾のないように処理ができなければならない．図 7･3 で示した例における問題は，共有データに対するアクセスルールがないという点にある．人間社会では共有物を使うときは，使用することを宣言し，順番を守るというルールがある．そこで，OS にも，共有資源の使用を宣言し使用の独占権をもらう機能が必要であり，これを排他制御（Serialization）あるいは相互排除（Mutual Exclusion）という．

7-1-3 クリティカルセクション

図 7･3 に示した例は，データベースを複数のプロセスでアクセスする場合にも当てはまる．再び，典型的な例として銀行システムの預金システムを単純化した例を引用する．例えば，預金の残高照会の処理では，複数のプロセスが同時に同一のデータベースを読み出していても問題はない．しかし，預金の預け入れ，引き出しのように，データベースを更新するような処理で，書込みを行うプロセスが一つでもあるときは，その処理の間，データベースのアクセスは抑止されねばならない．

図 7･3 は更新処理を同時に行うために生じる矛盾の例であり，このような矛盾が生じるプログラムの処理区間をクリティカルセクション（Critical Section）あるいはクリティカルリージョン（Critical Region）と呼ぶ．その矛盾を避けるためのいくつかの方法が考案されている．以下，その方法について説明する．

■7群 － 3編 － 7章

7-2 排他制御

（執筆者：吉澤康文）［2013年2月 受領］

7-2-1 割込み禁止による排他制御

排他制御する必要性が生じるのは，クリティカルセクションの実行中にプロセススイッチが発生するためである．したがって，プロセススイッチを発生させないことが最も単純な解決法として考えられる．プロセススイッチは3章3-1節に述べたように，割込みやある種のシステムコールによってプロセスの再スケジュールが行われたときに発生する．このため，クリティカルセクションに入る前に，すべての割込みを禁止状態（disable）とし，クリティカルセクションの実行後に割込み可能状態（enable）とすれば，プロセススイッチが発生せず，排他的な処理が達成されることになる．**図7･4**にプログラムのひな形を示す．ここで，disable( ) が割込み禁止とする操作であり，enable( ) で割込みを可能にする操作となる．

```
While(TRUE){
        disable();  /* 割込み禁止状態にする */
        critical_section();  /* クリティカル
                                セクションの実行 */
        enable();  /* 割込み可能状態にする */
        non_critical_section(); /* 非クリティカル
                                セクションの処理 */
}
```

図7･4　割込み禁止によるクリティカルセクションの実行

このような方法では，タイマ割込み，入出力割込みなどすべての割込みが抑止される．もちろん，複数のプロセッサで実行している際の他系からの通信割込み信号も受付けないことになる．したがって，プロセススイッチは抑止されるので，クリティカルセクションを独占して実行することが可能となる．しかし，この方式の最大の欠点は，割込み禁止のような権限をユーザプログラムに与えることから生じる信頼性と性能の低下にある．

一般のユーザプログラムに割込み禁止モードを可能とする権限を与えた場合，そのプログラムが割込み可能状態に戻さないことも予想される．つまり，enable( ) を実行しないケースである．プログラムにはバグがあり得るので，このようなことは意図的，あるいは無意識になされると考えた方がよい．もしこのようなことが起きると，コンピュータシステムは割込みを受付けない状況に陥り，端末からの入力が行えなくなるばかりか，タイマ割込みで起動されるすべての処理が実行不可能となる．つまり，システムダウンと同じ状況になる可能性が生じ，信頼性が低下する．

もう一つ重要なことは，この方式はマルチプロセッシングに対応できないことである．すなわち，マルチプロセッシング環境では割込み禁止の設定が CPU ごとに行われるので，各CPU に一つのプロセスが disable( ) で排他的に動作したとしても，システム全体としてみると，接続されている CPU の台数だけこのクリティカルセクションに入り込むプロセスが存在する可能性がある．したがって，この方法は1台の CPU を前提にした方法であるため排他制御がシステム全体として正しく実行されないことになってしまう．

7-2-2 鍵を掛ける排他制御方式

我々がある部屋を独占して使用するとき，例えば，ホテルの部屋を借りるというようなとき，多くの場合，使用中であることを表示したり，部屋に鍵を掛けたりする．これと全く同様に，クリティカルセクションに入る前に，ロック（lock）を掛ける方法が考案されている．つまり，プログラマはクリティカルセクションに入る前にロックを掛けることを試みるのである．そして，ロックが掛かった後にクリティカルセクションの処理を行い，完了後ロックを空けるのである．この方法は，前記のように割込み禁止というハードウェアに依存した方法ではなく，ソフトウェアによる方式といえる．

この方法のひな形を**図 7･5** に示す．lock( ) がロックを取得する操作であり，lock( ) が完了したならばロックが取得できたとプログラマは判断してよいことにする．そしてクリティカルセクションを実行し，完了した際には unlock( ) によりロックの解放を行うのである．したがって，lock( )，unlock( ) は共に重要な操作であるため，システムコールとして OS がその処理を行うこととするのが一般的である．

```
while(TRUE) {
     lock();   /* 鍵を取得する */
     critical_section(); /* クリティカル
                                  セクションの実行 */
     unlock();  /* 鍵を解放する */
     non_critical_sectoin(); /* 非クリティカル
                                    セクションの処理 */
}
```

図 7･5　クリティカルセクションの実行時に鍵を取得する

7-2-3 ビジーウエイトによるロック

上記の lock( )，unlock( ) などのロックの具体的な掛け方をここでは考察する．単純な方法は**図 7･6** に示すとおりである．ここで，クリティカルセクションにプロセスが入っていないとき，変数 LOCK はゼロとしておく．したがって，初期値はゼロである．

```
int LOCK = 0;  /* 変数：鍵の宣言．初期値をセット*/
lock()         /* 鍵取得処理開始              */
{
  while(LOCK != 0) ; /* 鍵を取得するまで待つ  */
     LOCK = 1;       /* 鍵の取得を示す        */
}
unlock()             /* 鍵の解放処理 */
{ LOCK = 0; }
```

図 7･6　ビジーウエイトによるロック確保方式

ロックを掛ける要求が来ると，lock( ) では変数 LOCK の値を確認し，もしもゼロでないならば，ほかのプロセスがクリティカルセクションに入っていると判断する．したがって，LOCK の値がゼロになるまで待ち続ける．図 7･6 では空文（; ）を実行した後で，再び LOCK の値がゼロか否かを判断し続けるのである．このために lock( ) は CPU を使い続けているので，この状態をビジーウエイトと呼んでいる．つまり，CPU の利用率は 100 %であるが，生

産的な処理を行っていないというわけである．

このような処理を行う OS では，タイマによる割込みを可能にしておき，LOCK を監視し時間の打切りを行うような機構が必要になる．あるいは，前章に述べたタイムスライススケジューリングなどの機能を兼ね備えておかねばならない．そうしないと，永遠に LOCK の監視を行う状況が続き，実質的なシステムダウンにつながってしまう．

この一連の処理は，1 台の CPU の場合には割込み禁止で実行すれば可能であり有効な場合もある．しかし，前記のとおり，割込み禁止で行う処理部は極力短い範囲とするべきである．また，仮に割込み禁止状態での実行が有効であってもマルチプロセッシング環境では正しく動作しないという致命的な欠点がある．

7-2-4 マシン語レベルでの操作

図 7･6 に示したように高級言語で記述したビジーウエイト処理を考察してきた．それらは最終的にはマシン語で実行されるので，ここでは lock( ) 処理の一部をマシン語でどのように実行されるかを考察する．図 7･6 の一部をマシン語として記述したのが**図 7･7** である．

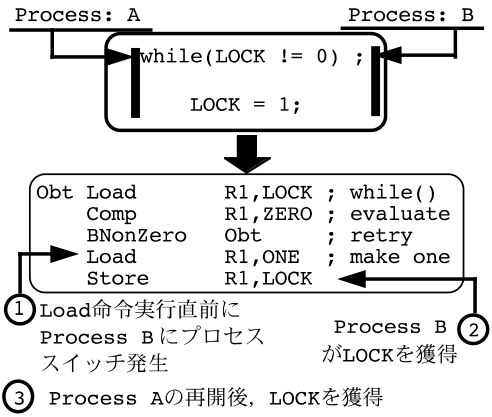

図 7･7　命令語でのビジーウエイト処理

クリティカルセクションに入るためには，この例に示したようにメモリ上の変数 LOCK を必要とする．LOCK 値を調べ，値を入れるという操作は一瞬にはできない．つまり，LOCK を調べ，判定した後に値を格納することになるので，コンピュータは複数の命令を実行して初めて LOCK 値が更新されることになる．

lock( ) の操作を図 7･7 のようなマシン語で考察する．最初は，LOCK の値がゼロかどうかを判定する．この例では，レジスタ R1 に LOCK の値を読み込み，定数の ZERO と比較する．その結果を判定して分岐する命令が実行される．つまり，LOCK の判定にマシン言語の三つのステップが必要となる．仮に，LOCK の値がゼロであったときには，LOCK に 1 をセットするために更に 2 ステップを必要とする．つまり，クリティカルセクションに入ることを決定するためには，最低でも 5 ステップの命令を実行しなくてはならない．

割込みが許可されている状況では，lock( ) 処理のわずかな 5 ステップの実行が完了するまでの間にも割込みが発生する可能性がある．運悪く，割込みが発生し，プロセススイッチとなり（①), ほかのプロセスが lock( ) を呼び出したとすると, まだ LOCK の値はゼロなので, 後から lock( ) を呼び出したプロセスがクリティカルセクションに入る権利を得てしまう（②)．そして，割り込まれたプロセスが再開されると，そのプロセスは LOCK 値がゼロである処理を続行するので，そのプロセスもクリティカルセクションに入ってしまうことになる（③)．

上記のごとく，排他制御を行う場合にはLOCK変数をあたかも 1 マシンサイクル[*1] で内容をテストし，値を更新する機能が必要となる．そして，マルチプロセッシング環境においても矛盾なく動作することが保障される必要がある．このため，LOCKのような変数に対する操作は多くの場合，ハードウェア的な排他制御の手段が講じられている．

7-2-5 アトミックオペレーション

排他制御を行うには，一つの変数を一瞬の内にチェックし値をセットすることを行う機能があれば都合がよい．しかし，時間ゼロでこの操作を行うのは原理的に不可能である．そこで，LOCK のような変数を排他制御の変数として指定したときには，この変数を参照している間は割込みが発生しないこと，またマルチプロセッサ環境では，ほかのプロセッサにより参照（読込み，書込み）が禁止される機構が望ましい．これらの理由から，この種の機能はハードウェア的に実装されている．

この操作をアトミックオペレーション（Atomic Operation）と呼び, 代表的な命令として IBM 社の System/370 に用意された TS（Test and Set）命令がある．TS 命令は指定したオペランドの先頭の 1 ビットの操作に対してアトミックオペレーションがなされる．また，System/370 以降に用意された CS（Compare and Swap）命令では，4 バイトのワード（Word）に対してアトミックオペレーションを行う．更に CDS（Compare Double and Swap）命令は 8 バイトのオペランドを対象にしているので，双方向ポインタ（Bidirectional Pointer）に対する操作の場合に向いている．それ以降のコンピュータには類似の命令が開発されている．

```
CS       R1,R3,D2(B2)
if (R1 が第 2 オペランドに等しい)
   {R3 を第 2 オペランドに格納}
else
   {R1 の内容を第 2 オペランドに格納}
```

図 7・8　アトミックオペレーションを実行する CS 命令の仕様

図 7･8 に，CS 命令の仕様を示した．ここで，R1 は第 1 オペランドであり汎用レジスタである. また, D2(B2) は第 2 オペランドであり, 排他制御に使用する主メモリ上の変数（LOCK）を指定する．そして第 3 オペランドの R3 はレジスタであり，第 2 オペランド（LOCK）に代入する値を入れておく．この命令は以下のように実行される．

*1 マシンサイクル : 1 マシンサイクルで命令が実行されているときは割込みが発生しないという前提である．

・第 1 オペランドが第 2 オペランドと比較される．
・その結果，同一ならば第 3 オペランドが第 2 オペランドに格納される．
・もし，同一でないならば，第 2 オペランドの内容が第 2 オペランドに格納される．

CS 命令のオペランドにはアトミックオペレーションがなされるので，たとえほかの CPU から同一のオペランドに読出しや書込みの参照があっても，その命令は抑止されてしまう．つまり，参照を試みた CPU は事実上 CS 命令が完了するまで止まっていることになる．このような操作を行うために，一般的に CS 命令は便利であるが多くのマシンサイクルを要するので多用すると性能低下になりかねない．

7-2-6 アトミックオペレーションの例

上記の CS 命令を用いたロックの取得例を**図 7･9** に示す．ここでは排他制御変数を LOCK とし，もしロックが取得されていないならば LOCK = 0 とする．そして，ロックを取得したならば，該当の CPU 番号である MyCPUid を LOCK に入れるものとする．①ではあらかじめ MyCPUid の値をレジスタ R3 に格納しておく．②は LOOP のラベルである．③～⑤はロックがほかのプロセスにより使用されていないことを確認しているが，ロックがほかのプロセスにより使用中ならば，解放されるまで③から⑤の処理を続ける．このような待ち方をスピンウエイト（Spin Wait）と呼ぶ．

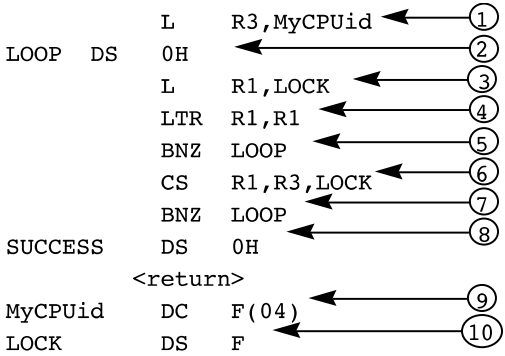

図 7･9　アトミックオペレーションの実施例

そして，ロックが使用されていないことを確認したならば，その時点でアトミックオペレーションを行うために，CS 命令を使用する．CS 命令では，内容がゼロであるレジスタ R1 を再度 LOCK と比較する．つまり，LOCK がゼロであったのは③の時点であり，その後，この CPU では④～⑤を実行しているので，その間にほかの CPU が LOCK を変更してしまっている可能性がある．もし，変更がなされていればロックの取得は失敗であり，LOCK を R1 に格納して CS 命令は完了する．そして，再度 LOOP に制御が渡り再試行される．

CS 命令で LOCK が変更になっていないならば，R3 の内容が LOCK に書き込まれ，ロックを取得することができるのである．そして，⑧に制御が渡ることになる．

この例で示したように，②～⑤の操作は CS 命令を極力実行しないで済むようにした工夫である．このようなアトミックオペレーションを行うハードウェア機構を用いれば，排他制

御の lock( ) は割込み許可モードで実行できるばかりでなく, マルチプロセッシング環境でも矛盾することなく動作できる.

■7群 － 3編 － 7章

7-3 OSによる排他制御

（執筆者：吉澤康文）［2013年2月 受領］

7-3-1 排他制御におけるプロセス状態の遷移

クリティカルセクションに入る前にはロックを掛ける方法が有効であり，図7･6に示したlock( ) を呼び出す必要がある．この lock( ) 関数では，排他制御変数である LOCK の操作は上記に説明したようにアトミックオペレーションを行う CS 命令などを使う必要があった．

ところで，排他制御を行うにあたり，図7･6に示したような論理で十分だろうか．つまり，あるプロセスがクリティカルセクションに入っているとき，ほかのプロセスが同一のクリティカルセクションに入ろうとすると，図7･6では既にクリティカルセクションに入っているプロセスがクリティカルセクションを抜けることを宣言する unlock( ) を呼び出すまでビジーウエイトしてしまうことになる．

ビジーウエイトは CPU のむだ遣いであり，プロセスの生産的な作業に貢献しない．したがって，ビジーウエイトするのではなく，当該プロセスをロック待ち状態にし，CPU を使用できるほかのプロセスを動かした方がコンピュータの生産性は向上することになる．つまり，ロックを取得失敗したプロセスはブロック状態（Blocked State）にしておき，ロックが解放された時点でブロック状態から実行可能な状態（Ready State）にする方が効果的である．

排他制御機能は並列処理する場合に必須であるため，OS の標準的な機能として提供されていなければならない．そこで，このような状況をプログラマが意識することなく，OS はクリティカルセクションに入るのを待っているプロセスと待たせているプロセス間の相互の通信を暗黙の内に行う必要がある．

図 7･10 にその機構を示す．ここでは，プロセス A が先にクリティカルセクションに入り lock( ) を実行する（①）．そこで，共有データを読み込み更新するが，運悪く割込みが発生し（②），プロセススイッチが生じ，プロセス B に制御が渡る（③）．プロセス B はクリティカルセクションに入ろうとして lock( ) を呼び出すが，既にプロセス A によってロックはとられているので，プロセス B は OS によりロック待ち状態になり，ブロックされる．

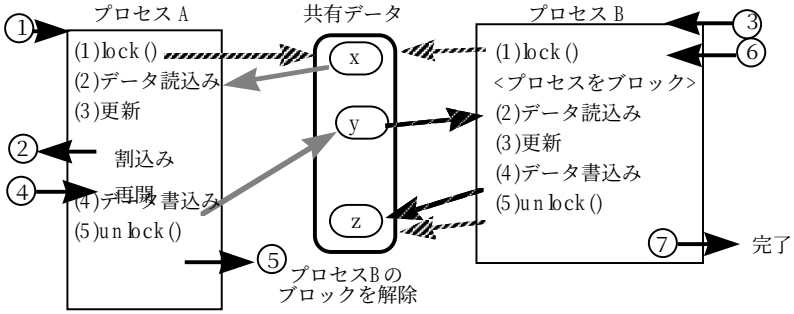

図 7・10　OS の提供する排他制御の実現方式

この結果，再びプロセス A が制御を受け取り（④），クリティカルセクション内の処理が

再開される．そこで共有データの内容を書き込み，処理を完了させると unlock( ) でロックを解放する．このとき，OS の unlock( ) 処理ではロックを待っているプロセスが存在することを判定し，もしあれば，そのプロセスのブロック状態を解き，実行可能状態にする（⑤）．この結果，プロセス B は実行可能となり lock( ) を再度試みる（⑥）．今度はロックを取得することができ，クリティカルセクションの実行が行われる（⑦）．

7-3-2 OS の排他制御方式

OS は共有資源の排他的な利用のために，複数のプロセス間の橋渡しを行う必要がある．図 7･6 はロック取得と解放の原形であるが，上記のとおり OS の機能としては不十分である．ここではロック取得時，解放時の各々において，OS の実行すべき手順を整理して説明する．**図 7･11** がそのおおまかな論理である．

```
int LOCK = 0;  /* 変数：鍵の宣言．初期値をセット*/
lock()         /* 鍵取得処理開始                */
{
  if (LOCK != 0) { /* プロセスをブロックする    */
      make_block();
  } else  {
     if (atomic(LOCK) == 0) return;
     make_block();  /* アトミックオペレーション
     }                 により鍵を取得する     */
}
unlock()             /* 鍵の解放処理 */
{ LOCK = 0;
  rel_process();     /* 排他制御待ちプロセスを
                        実行可能状態にする */
}
```

図 7・11　OS の排他処理方式

(1)　ロック取得時の処理

(a) ロックがほかのプロセスで既に取得されているか調べる．

(b) もし，ほかのプロセスに取得されている場合は，本プロセスをロック取得待ち状態にする（図 7･11 では make_block( ) を呼び出す）．

(c) ロックが解放されていれば，アトミックオペレーションによりロック変数の書換えを試み，ロックの取得を確認する（図 7･11 では atomic(LOCK) 呼び出す）．

(d) もしロックの取得が確認できたなら（図 7･11 では atomic(LOCK) を呼び，返り値がゼロの場合）処理は完了する．しかし，ロック取得に失敗したときは，本プロセスをブロック状態にする．

(2)　ロック解放処理

(a) 排他制御変数を解放状態にする（図 7･11 では LOCK = 0）．

(b) 排他制御により待ち状態にあるプロセスが存在することを調べ，もし存在するならば，それらのプロセスを実行可能状態にする（図 7･11 では rel_process( ) を呼び出す）．

■7群 － 3編 － 7章

7-4 セマフォの実際

（執筆者：吉澤康文）［2013年2月 受領］

7-4-1 基本原理

今まではロックを確保できたか否かという2値を取り扱ってきた．また，ある一つの資源に対する排他制御を考えていた．しかし，もっと一般的な問題としてこの制限を取り払って考える必要がある．その代表的な排他制御法がE. W. Dijkstra（ダイクストラ）によって提案されたセマフォ（Semaphore）である．セマフォとは，もともとは鉄道の単線区間などで進入許可を示す腕木信号のことをいう．

セマフォではプロセス間の通信として整数値をイベント（事象：Event）変数として使用する．イベント変数とは，クリティカルセクションにプロセスが入ったり，出たりすることや，資源を使用したり解放したり，などを記録する状態を示す整数である．このイベント変数に対する二つの操作命令が用意されている．これらは，7-3-2項に示した，lock( )，unlock( ) の機能を含んだもので，P(sem) と V(sem) 操作である．ここで，sem は整数値のイベント変数であり，初期値は資源数で，それらへの操作は以下のとおりである．

(1) **P(sem) の手順**

(a) sem が1以上なら，sem 値を－1して，処理を完了する．

(b) 上記以外は，P(sem) を実行したプロセスを sem 待ち行列に入れブロックし，スケジューラに制御を渡す．

(2) **V(sem) の手順**

(a) sem 値を+1する．

(b) sem 待ち行列にブロックされているプロセスがあれば，それらのブロック状態を解除し実行可能状態とし，スケジューラに制御を渡す．

(c) 待ち行列がなければ，処理を完了し，V(sem) を呼び出したプロセスを続行する．

P(sem), V(sem) 命令におけるイベント変数 sem の初期値は資源数である．したがって，7-3-2項で示した lock( )，unlock( ) は sem = 1 の例であり，バイナリセマフォ（Binary Semaphore）と呼ぶことがある．また，セマフォ操作のことを PV オペレーションと呼ぶこともある．PV はオランダ語の P（Passeren），V（Verhoog）の意味である．この処理で，イベント変数の sem の操作はアトミックオペレーションされねばならない．

7-4-2 生産者と消費者問題

ソフトウェア設計において，性能や信頼性は重要な要素であり並列処理はこの両者に有効である．したがって，並列処理できる部分はマルチタスキングの構造となるように設計しておくべきであるが，そのようなときに直面する問題のひとつに「生産者と消費者の問題（Producer And Consumer Problem）」がある．ここでは，セマフォを使う例として説明する．

図 7･12 に示すように，複数のプロセスが共通のデータ領域にレコードを書き込み，書き込まれたレコードを複数のプロセスが読み出すというモデルが「生産者と消費者問題」である．

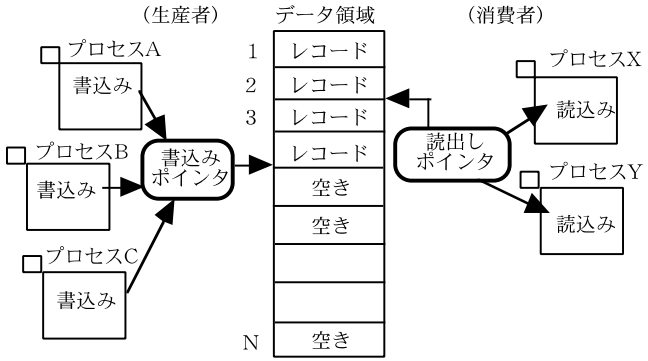

図 7・12　生産者と消費者問題

```
#difine   N   50

int  mutex = 1;       /* for serialize the critical section */
int  n_vacant = N;    /* number of available entries */
int  n_occupied = 0;  /* number of unused entries */

struct rec{           /* record definition */
           char name[16];
           int  age;
           float score;
}

struct rec com_area[N]; /* common entries */

void  producer(void)
{
     struct rec *person;
     while(TRUE) {     /* do forever  */
         make(person); /* make one record */
         P(n_vacant);  /* decrement n_vacant */
         P(mutex);     /* get permission */
         put_rec(person);  /* put new item */
         V(mutex);     /* leave critical section */
         V(n_occupied); /* increment occupied counter */
     }
}

void  consumer(void)
{
     struct rec *person;
     while(TRUE){
         P(n_occupied);   /* decrement n_occupied counter */
         P(mutex);        /* get permission */
         get_rec(person); /* get one record */
         V(mutex);        /* leave critical section */
         V(n_vacant);     /* increment vacant entry no. */
         use(person);     /* manipulate gotten record */
     }
}
```

図 7・13　セマフォを利用した生産者と消費者問題

この場合，生産者と消費者は各々一つのプロセスのこともある．このモデルのポイントとなる点は三つある．

(1) データ領域は各プロセスが共通してレコードを取り出したり書き込んだりするため，操作は排他制御を行う必要がある．
(2) 読出しポインタ値は書込みポインタ値を追越してはならない．つまり，生産者がレコードをまだ書き込んでいないとき，消費者のレコード要求は待たされねばならない．
(3) 逆に読出しポインタを書込みポインタが追越してはならない．つまり，消費者の処理が遅い場合には生産者プロセスを待ち状態にしなければならない．
(4) 上記の待ち状態は生産者，消費者がレコードを生産した場合，ならびに消費した場合にそれぞれ解除されるようにしなくてはならない．

図 7･13 は生産者と消費者の問題のプログラム例である．この例では，mutex なるイベント変数で共通データ領域のレコード挿入や取出しの操作を逐次処理する．また，データ領域には *N* 個のレコード領域があるので，n_vacant の初期値はその値を入れておく．レコードは人名と年齢のような単純な個人レコードが想定されている．put_rec( )，get_rec( ) は共通領域にレコードを入れたり，取り出したりするプログラムである．書込みポインタ，読出しポインタの操作は mutex により排他制御される．

7-4-3 デッドロック問題

排他制御においてプログラマの注意すべき点がいくつかある．その代表的なものがデッドロック（Deadlock）である．**図 7･14** にデッドロックの一例を示した．この例では，二つの共有資源 R1，R2 があると仮定し，デッドロックに至る過程を説明する．

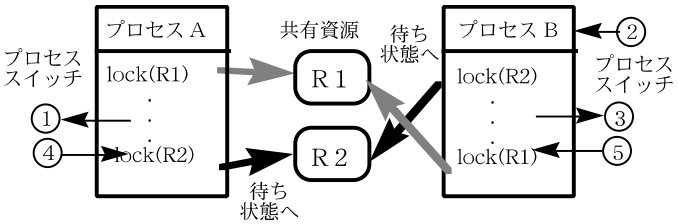

図 7・14　排他制御におけるデッドロックの問題

(1) プロセス A が最初に実行を開始し，lock(R1) によって資源 R1 を確保する．
(2) その後，何らかの割込みが発生し，プロセススイッチが生じる（図中の(i)）．
(3) その結果，プロセス B の実行となり，資源 R2 を lock(R2) で確保する（図中の(ii)）．
(4) 再び割込みが発生し，プロセススイッチにより制御がプロセス A に戻る（図中の(iii)～(iv)）．
(5) そこで，プロセス A は共通資源 R2 を確保すべく lock(R2) を実行する．
(6) しかし，R2 はプロセス B により既に確保されている．このためプロセス A は待ち状態になる．
(7) そこで，プロセス B が実行再開され，資源 R1 を確保するために lock(R1) を実行する（図中の(v)）．

(8) しかし，資源 R1 はプロセス A により既に確保されているのでプロセス B は待ち状態に陥る．

(9) この結果，プロセス A とプロセス B は，お互いがもっている資源を待ちあうことなるので，永久に待ち状態が解除されなくなる．この行詰り現象をデッドロック状態と呼び，現実にコンピュータシステムで生じる問題である．

7-4-4 デッドロックの回避

上記，図 7･14 がデッドロック状態に陥った理由は明白である．両プロセスは二つの資源を必要としているが，資源確保の順序が逆になっている．プロセス A は R1, R2 の順に確保し，プロセス B は R2, R1 の順に確保している．つまり，この問題の解決は以下のとおりである．

(1) 複数の資源を必要とするときは，資源を確保する順序を同一にすべきである．

(2) または，複数の資源を必要とする処理では，一度にまとめて資源の確保を行うべきである．

上記はデッドロックを回避する基本的な事項であるが，以下の点も設計上重要である．

(3) 更に，使用済みの資源は直ちに解放すること．

(4) 資源を排他的に使用するときは，ほかのプロセスが使用していないことを確認してから確保の要求を行うこと．

そして，

(5) 排他的（Exclusive）に利用するのか共用（Share）するのかをはっきりと区別して要求を行うこと．

などである．

■7群 － 3編 － 7章

7-5 UNIXにおけるプロセス間通信と排他制御

（執筆者：吉澤康文）［2013年2月 受領］

7-5-1 コマンドインタープリタにおける並列処理

UNIX は多くの開発者により発展を遂げたカーネルである．そのためか，プロセス間通信に関する機能は実に豊富にある．ここでは，その中から代表的であり基本的な機能を説明することにする．

UNIX が元祖で代表的なプロセス間通信機能がパイプ（Pipe）である．UNIX はプログラマのための OS として生まれているので，プログラムを作成する人のために有益なコマンドが数多く用意されており，それがまた新しい価値を生んできた．これらのコマンドを実行するのがコマンドインタープリタあるいはシェル（Shell）と呼ばれるプログラマの道具（環境）である．

シェルにはたくさんのコマンドが開発されているため，それらを組み合わせることで一つの意味のある結果を得る場合がある．例えば，「smith というログイン名の人が現在 UNIX を利用しているか知りたい」というときは，who というコマンドと，その実行結果を入力にして文字を検索する grep (global regular express printer）なるコマンドの組合せで可能となる．UNIX では，端末から以下のように入力する．

```
who  |  grep "smith"
```

二つのコマンドは縦棒 ｜でつなげればよく，who の出力はディスプレイには出力されない．who の出力をどこかに貯めておき，grep がそれを読み出す仕組みになっているのである．このようにコマンドの実行結果を一時的に蓄えておくのがパイプである．もし，パイプの中に who の出力する文字列があるならば，grep コマンドは smith という文字列を次々と探していればよいわけであり，二つのコマンドは並列に動作できることになる．

上記の縦棒 ｜は「who の出力をパイプに置き，grep の入力をパイプから取り出せ」という意味である．そして，「who と grep の処理を並列に実行せよ」ということを示している．以下，本節では UNIX での代表的な並列プロセスの実行を，シェルの作り方を例に説明する．

7-5-2 パイプの概要

二つのコマンドをパイプ ｜ でつなぐ処理を詳しくみると，**図 7･15** に示すようになっている．つまり，パイプとは概念的には二つのプロセス間の情報路であり，あたかも水道のパイプを引いたような類推（Analogy）である．UNIX にはこのような現実社会の道具を類推としたシステムコールがある．3 章 3-2 節で説明した fork( ) なども食事に使うフォークの類推であり，手で握っている部分は 1 本であるが食べ物を刺す部分が複数本になっていることから子プロセスを複数作るイメージに合致させたネーミングである．

図 7･15 はシェルによってコマンド who の実行結果をコマンド grep の入力に使用する例である．つまり，コマンド who を実行するプロセス X の標準出力への文字列をプロセス Y が標準入力からの入力とし，コマンド grep を実行するのである（標準入力，標準出力についてはファイル管理の章で説明する）．

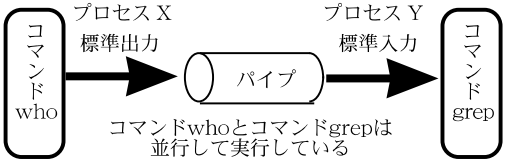

図 7・15　シェルコマンド実行におけるパイプによるプロセス間通信

パイプは，前節 7-4-2 で説明した生産者と消費者問題のように，二つのプロセス間の情報の入出力をメモリ上で行い，両プロセスの並列動作の高性能化を図っているという利点がある．このように，OS により入出力の動作をメモリ上でシミュレーション（Simulation）するテクニックの一例をここに見ることができる．**図 7･16** にその概念的なモデルを示す．パイプの実体はメモリであり，それを二つのプロセスが共有して使用している．write システムコールを実行するプロセス A が生産者であり，read システムコールを行うプロセス B が消費者ということになる．したがって，UNIX カーネルはこの両者の同期をとる役割をもつ．

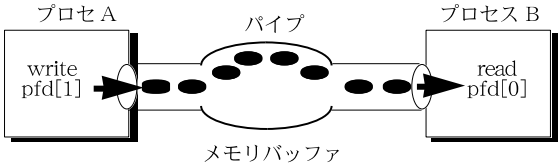

図 7・16　入出力をメモリでシミュレーションするパイプ

7-5-3 パイプを作る：pipe

UNIX はパイプを作成するシステムコールが用意されている．パイプは親子間及び子供間のプロセス通信に使用されるのが一般的である（特殊なものとして，名前付きパイプがある）．つまり，プロセスファミリ内部（ユーザ ID が同じプロセス同志）の通信手段といえる．このため，親プロセスはシステムコール pipe( ) を実行してあらかじめパイプを作成しておき，その後，fork( ) により子プロセスを生成する．この結果，同一のパイプを親子で共有することになるので交信が可能になる．別の方法として，親が複数回 fork( ) を実行すれば，兄弟プロセス同志の通信が可能隣，高度な並列処理が可能となる．

ここで，パイプシステムコールの仕様を**図 7･17** に示す．パイプには一方向にしか情報が流

```
int pipe(pfd)    /* パイプを作る    */
int pfd[2];      /* ファイル記述子 */
/* 返り値 = 0  if succeed          */
/*          -1 if failed           */
/* pfd[0]: 読み込み用記述子        */
/* pfd[1]: 書き込み用記述子        */
```

図 7・17　pipe システムコールの仕様

れないという制限があるので，pipe(pfd) はファイル記述子（File Descriptor）の対が作成される．つまり，パイプからの読込み用に pfd[0]と書込み用の pfd[1] が作られるのである（ファイル記述子については，次章の UNIX ファイルで説明する）．

パイプは一方通行の通信チャネルであるので，親から子に情報を流すときは，親プロセスは pfd[1] に書込みを行い，パイプにデータを置き，子プロセスは pfd[0] を指定して，そのデータを読み込む．したがって，パイプから読込みを行わない親の pfd[0] と書込みを行わない子の pfd[1] は無用であるので，fork( ) の後で，親子のプロセスはそれらをクローズ（Close）しておくことが望ましい．

パイプを作成してから fork( ) を実行し，その後，パイプを一方向のデータチャネルとするために親の pfd[0] と子の pfd[1]をクローズした様子を**図 7･18** に示す．

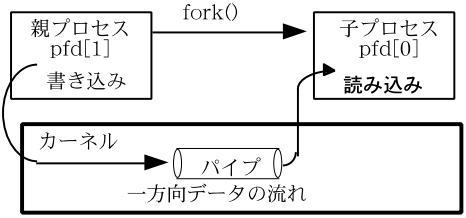

図 7・18　パイプは一方向のデータチャネル

7-5-4 双方向パイプ

パイプは親子のプロセス間で容易にかつ高速に行える情報通信手段であるため，UNIX のプログラミングでは頻繁に利用される．しかし，一つのパイプは一方向の通信チャネルであるため，相互に通信するためにはもう一つのパイプを必要とする．

したがって，子プロセスを生成する前に fork( ) で 2 本のパイプを作っておき，親子で送信，受信のパイプをそれぞれに約束しておく必要がある．そして，不要なファイル記述子をクローズして破棄することが望ましい．このようなパイプの利用方法を双方向パイプと呼ぶ．**図 7･19** に双方向パイプを実現したときのイメージを示した．

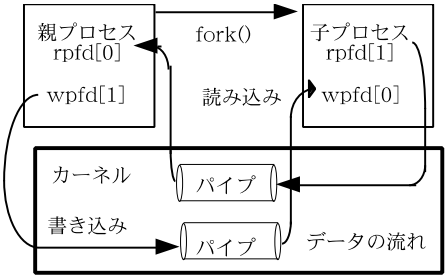

図 7・19　双方向パイプの実現方法

7-5-5 ファイル記述子のコピーを行う：dup

UNIX のシェル実行環境では，三つのファイルが使用可能状態になっている．それらのファイル記述子（ファイル記述子は 5 章 5-5 節で説明）は，0 が標準入力（Standard Input : 多くはキーボード），1 が標準出力（Standard Output : ディスプレイ）であり，2 が標準エラー出力（Standard Error Output : システムコンソール）である．そこで，7-5-1 項で説明した，who | grep smith" のコマンドの処理を考える．

この処理は，図 7･15 に示したように二つのプロセスにより並列処理されるが，who を処理するプロセスの標準出力はパイプへの出力であり，grep の標準入力はパイプになっている．このようなトリックを可能にするためには，who のファイル記述子 1 をパイプへの出力とし，grep の標準入力 0 をパイプからの入力にしなくてはならない．この処理形態を示したのが**図 7･20** である．

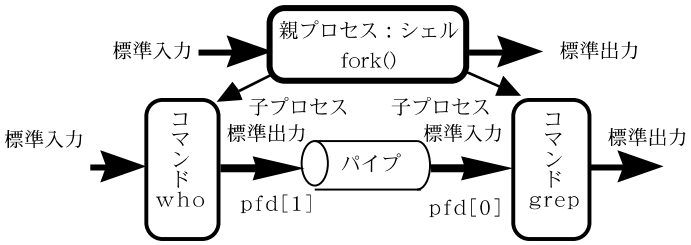

図 7・20　シェルコマンド実行におけるパイプと標準入出力の関係

この例では，親プロセスはパイプを生成した後に，二つの子プロセスを生成し，who と grep を各々に実行させている．このとき，親プロセスが生成したパイプは子プロセスどうしの通信に利用し，そのパイプは who の標準出力と grep の標準入力に使われる．つまり，パイプのファイル記述子の pfd[1] が who の標準出力となり，pfd[0] が grep の標準入力となるようにしなければならない．このトリックを行うシステムコールが dup である．

dup を使って図 7･20 のような環境を整えるならば，who や grep のプログラムは標準入出力を前提にしたプログラムを作成しておくだけでよく，パイプ利用のために一切の変更をする必要はない．シェルのようなプログラムがその環境を dup で整えるだけでよい．そこで，そのトリックのメカニズムを以下に説明する．

(1)　ファイル記述子とファイルポインタ [*2]

図 7･21 にファイルがオープンされたときのプロセステーブルとファイル記述子テーブル，ファイルテーブルそして i-node テーブルなどの関係を示す．

ファイルがオープンされるとファイル記述子が返り値として与えられるが，その値はファ

[*2] ファイルポインタは 2 種類の意味で出てくる．一つは，図 7･21 のファイル記述子テーブルが示しているファイルテーブルへのポインタのことである．もう一つは，図 7･21 に示したファイルテーブル内のデータとしてのファイルポインタである．この値は，ファイル内の読出し，書込みの先頭からのバイトアドレスである．つまり，アクセスポインタである．lseek( ) システムコールではこの値を求めたり，新たな値を設定できる．

イル記述子テーブルのエントリ番号（index）である．シェル実行環境では，エントリ番号 0, 1, 2 が作られ，ファイルテーブルへのポインタが stdin，stdout，stderr の値になるのである．先に説明した pipe( ) システムコールはパイプというファイルを作成（creat 相当）するので，i-node が作られ，メモリ上にも高速アクセスのために i-node テーブルの内容を保持する．

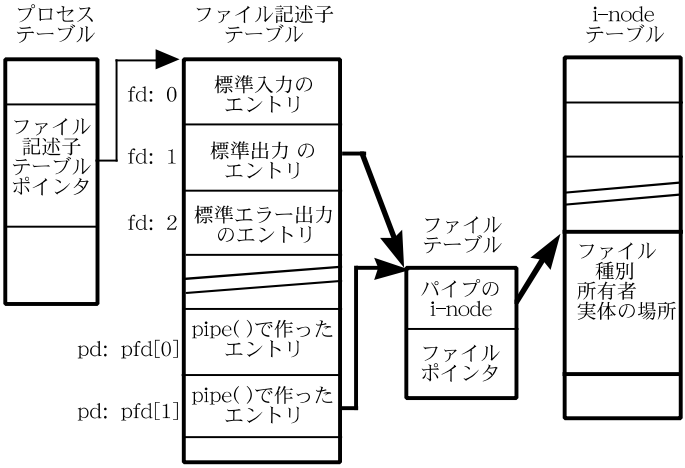

図 7・21　プロセステーブルとファイル記述子の関係

その結果，pipe( ) の返り値としてファイル記述子の番号である pfd[0]，pfd[1] が返され，図 7･21 に示したようにファイル記述子テーブルの pfd[0] と pfd[1] 番目が，各々のエントリとして作成される．

who | grep "smith" の処理では，who を実行するプロセスの標準出力を pfd[1] としなくてはならないが，そのためには図 7･21 に示したようにファイル記述子 1 のエントリがパイプのファイルテーブルを示している pfd[1] のエントリと同じファイルテーブルのポインタを示す必要がある．次にその手順を示す．

(2)　i-node へのポインタをコピーする dup システムコール

図 7･21 のように標準出力のエントリに pfd[1] の内容をコピーするために，UNIX には dup( ) システムコールが用意されている．具体的な手順として，まず dup( ) を実行する前に標準出力のエントリを空き状態にしておく．このために，close( ) システムコールによりファイルを閉じればよい．close( ) のパラメータはファイル記述子なので，標準出力ならば，close(1) である．標準入力はオープン状態であるので，最も値の小さなファイル記述子が空き状態になる．dup( ) はファイル記述子テーブルの先頭から空き状態のエントリを探し，指定されたファイル記述子に対応するファイルポインタをコピーするというわけである．

dup( ) システムコールの仕様は**図 7･22** のとおりである．dup( ) では引数に指定したファイル記述子に対応するファイルポインタを空いている最も数字の小さなファイル記述子の該当ファイルポインタにコピーするのである．図 7･20 の who を実行するプロセスを例にとると，

標準出力を close( ) することでファイル記述子 1 を空きとしておき，その後に dup(pfd[1]) を実行すると，ファイル記述子 1 の部分に pfd[1] のファイルポインタがコピーされることになり，図 7･21 の状態になる．

```
int dup(fd)    /* fdのコピーを行う    */
int fd;        /* original fd値     */
/* 返り値 = 新fdの値      */
/*         -1 失敗のとき */
/*  空き状態の最小値のfd を割当てる  */
```

図 7・22　既存のファイル記述子をコピーする dup

(3)　標準出力を pfd[1] にする手順

図 7･23 には who を実行する子プロセスの処理の概略を示した．親プロセス（シェル）は pipe( ) を実行し，あらかじめパイプを作成する．その後，who を実行するために子プロセスを fork( ) し生成する．

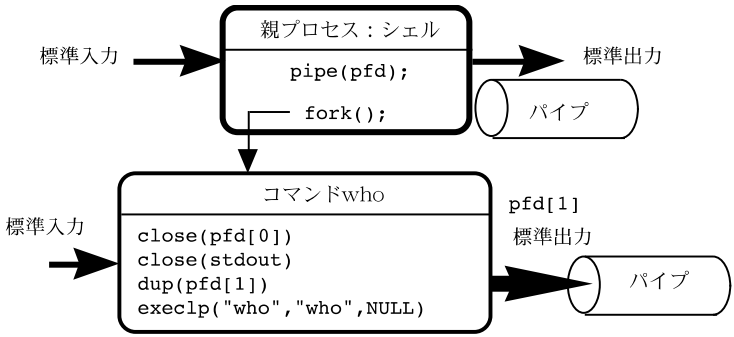

図 7・23　who **を処理するプロセスの生成とパイプ**

次に，子プロセスは使用しないファイル記述子である pfd[0] を close( ) する．who の標準出力をパイプにするために，第 1 ステップとして close(stdout) を行い，ファイル記述子テーブルを空き状態にする．その段階で stdout のエントリ番号 1 が，ファイル記述子テーブルの中で最も小さいことがわかっている. したがって, そのエントリ 1 にパイプの記述子 pfd[1] をコピーする必要が生じ，dup(pfd[1]) を実行するのである．このとき，プログラムとしてはその返り値が 1 であることを確認すべきである．このような環境を設定しておくと，who のプログラムを一切変更することなく stdout に出力した情報がパイプに出力される．

(4)　標準入力を pfd[0] にする手順

who を処理するプロセスと同様に，grep を処理するプロセスは，標準入力のファイル記述子テーブルに pfd[0] のファイルテーブルポインタを dup( ) を使用してコピーする. その手順は，**図 7･24** に示すとおりである．

親プロセスは who を実行する子プロセスを生成した後に，grep を実行する子プロセスを fork( ) する．親プロセスは使用しないパイプを close( ) により解放し，両子プロセスの完了

を wait( ) システムコールを発行して待つことになる．

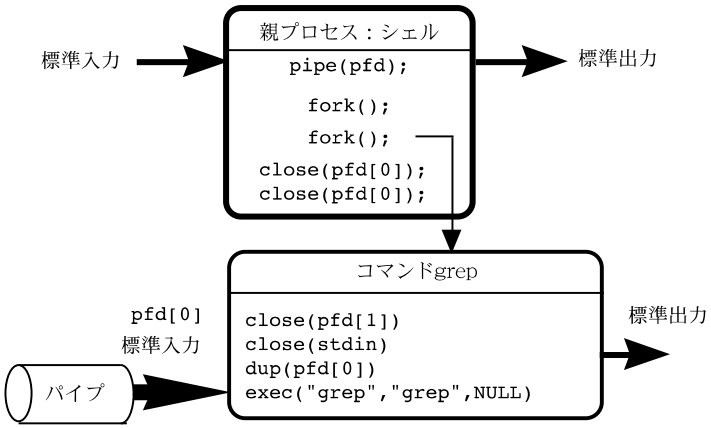

図 7・24　grep を処理するプロセス生成とパイプ操作

grep を実行する子プロセスは，who の子プロセスと同様に，不要となっているファイル記述子を close(pfd[1]) で解放する．そして，標準入力をパイプにするために，close(0) を行い，そこに dup(pfd[0]) で pfd[0] のファイルテーブルポインタをコピーする．その返り値が 0 であることを確認すると準備が完了したことになり，grep のプログラムをローディング（Loading）し処理を実行する．

これら，一連の処理によりパイプを使って二つの子プロセス間の情報通信を効率的に行うことが可能になる．つまり，機械的な動作を伴う入出力装置を使って情報通信するのではなく，パイプというメモリ上に作られた仮想的な装置に対して入出力を実行し，電気的な操作にシミュレートする技法である．この技法は「入出力の仮想化」であり，メモリを用いた高性能化技術の一例である．

(5)　dup( ) の拡張機能

上記のように，dup( ) は指定したファイル記述子のファイルテーブルポインタをコピーすることができるが，欠点は，プログラマが空きのファイル記述子の中で 1 番小さな番号をいつも意識していなければならないことである．標準入出力だけを取り扱っているときはそれで十分かも知れないが，一般的ではない．

そこで，コピーしたいファイル記述子を明示的に指定する機能として dup2( ) システムコールが用意されている．その仕様は**図 7・25** に示すとおりである．dup2( ) では，指定した newfd

```
int dup2(oldfd, newfd) /* duplicate fd   */
int oldfd;             /* original fd    */
int newfd;             /* destination fd */
/* 返り値 = (new fd value)         */
/*              -1  if failed          */
```

図 7・25　ファイル記述子をコピーする拡張 dup

のファイル記述子がオープンされた状態であると，カーネル内で close(newfd) を実行するので注意が必要である．返り値については dup( ) と同一である．

7-5-6 名前付パイプ

プロセス間通信としてパイプを詳しく説明してきた．パイプは親子，兄弟プロセス間の通信手段として有効である．つまり，同一のユーザ ID をもつプロセス間でのみ有効であり，ほかのユーザ ID をもつプロセスとの通信ができないという欠点がある．そこで，パイプの利点を生かして，異なるユーザ ID をもつプロセス間通信手段への拡張が望まれる．ここに説明する FIFO（First In First Out）は名前付パイプ（Named Pipe）とも呼ばれている．

パイプにはファイル記述子があり，図 7･21 に示したように内部的な管理テーブルだけをみるとファイルと全く同一に見える．しかし，パイプにはファイル名がないため，プロセス間通信を困難にしている．そこで名前を付けたパイプを生成する機能をもつのが，システムコールが mknod( )（Make Node）である．

mknod( ) 本来の目的はディレクトリ，スペシャルファイル，FIFO を作成することである．FIFO はパイプファイルであり，FIFO を指定したときのみスーパーユーザの権限がなくても実行できる．**図 7･26** には FIFO を作成する mknod( ) の仕様を示した．ファイルとしての性格をもたせるために，FIFO には名前が付けられ，更に，アクセス制御がなされる．また，入出力をメモリ上にシミュレートして性能を上げる機能ももたされている．ただ制約として，パイプであるがゆえに情報の流れは一方向であり，ランダムアクセスを行う lseek( ) が使用できないこと，既に読み込んだ情報を再度読み込むことはできない，などという点があげられる．つまり，先入れ先出し（FIFO）のみの使用に限定される．

```
#include <sys/types.h>
#include <sys/stat.h>

int mknod(fname, S_IFIFO, 0) /* make FIFO  */
char *fname;           /* ファイル名を指定 */
/* 返り値 = 0  if succeed            */
/*         -1  if failed             */
```

図 7・26　FIFO を生成するシステムコール

7-5-7 System V 系の IPC

AT&T により開発された System V ではプロセス間通信（IPC : Inter-process Communication）に次の 3 種類の機能拡張がなされ，それぞれ特徴を備え，目的に応じて使い分けるようになっている．

・メッセージ
・セマフォ
・共有メモリ

三つの IPC は仕様が共通している．各々のシステムコールと，使用時のインクルードファイル名を**表 7･1** に示す．使用の宣言は，msgget( )，semget( )，shmget( ) により行う．ここでは，通信を行う種別としての値を key としてパラメータに指定するが，これはプロセス間で

約束した値でなければならない．ちょうど，ファイルにおけるアクセストークン（Access Token）としての記述子を creat( ) で得ることに相当しており，get により識別子（Identification）を返り値として得て，その後のシステムコールに使用する．creat( ) と同様に，get においても flag を指定し，交信可能なプロセスの相手を限定することが可能である．

表 7・1　System V の IPC システムコール

機　能	メッセージ	セマフォ	共有メモリ
インクルード	<sys/msg.h>	<sys/sem.h>	<sys/shm.h>
使用の宣言	msgget	semget	shmget
IPC操作	msgsnd msgrcv	semop	shmdt shmat
制御操作	msgctl	semctl	shmctl

get では，記述子を作成する機能と，既に get で作成された記述子へのアクセスを可能にする機能とがある．後者はファイルにおける open( ) に相当し，これらは，get の flag で指定する．このような初期の手続きを行い，プロセス間において識別子を用いて IPC 操作や制御操作を行うことになる．以下，各々の概略を説明する．

(1)　メッセージ

プロセス間の通信手段としてパイプを説明した．パイプの特徴は，型のないバイト単位の情報を連続的に送る点にある．このようなデータを通信の分野ではストリームデータ（Stream Data）と呼ぶ．一方，データ長やデータ構造体などの型を指定して情報を通信する方法がある．これらはデータグラム（Datagram）と呼ばれ区別される．我々の社会での相互の通信と照らし合わせてみると，ストリームデータの通信が電話に相当し，データグラムの通信が手紙に相当する．

メッセージはデータグラムの通信に相当する．一つのメッセージは論理的なデータ構造をもち，プロセス相互にデータ構造について約束をした通信方法である．UNIX では，FIFO を使うと任意のプロセス間で通信路が設定できるので，パイプを通してメッセージ交換が可能になる．しかし，一般プログラマが FIFO を意識的に設定するのも煩わしい．そこで，メッセージを送信する手段などは意識させられることなくメッセージのやり取りを直接に可能とする方式が System V 系の UNIX には提供されている．

メッセージはプロセス間交信に一般性をもたせ，数種類のメッセージやメッセージの送受信に優先順位付けも許す．また，メッセージをプロセス間で送受することで同期処理も可能となる．System V におけるメッセージシステムコールの仕様を**図 7･27** に示す．

msgget( ) によってメッセージキューを生成すると，**図 7･28** に示す msqid_ds のようなメッセージキュー管理構造体が作成される．この構造体は msqid により間接的にポイントされている．このような環境を作成しておけば，メッセージキューに複数のメッセージのタイプをつなぐことができる．そして msgsnd( ) は任意のメッセージを作成し，そこにゼロ以外のタイプを指定してメッセージにキューイングすることができる．

一方，メッセージを受け取るプロセスは，既存のメッセージキューを msgget( ) で利用可能にし，msgrcv( ) で指定したタイプのメッセージを先頭から受け取ることができる．メッセ

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/msg.h>

int msgget(key_t key, int msgflg)
/* open同様にメッセージ使用の宣言をする*/
/* key: プロセス間で約束された値 */
/* msgflg: openのアクセスフラグに相当する */
/*         IPC_CREAT を指定しキューの生成を指定する*/
/* return = -1   失敗 */
/*        queue ID ファイル記述子に相当 */

int msgsnd(int msqid, struct msgbuf *msop,
                      int msqsz, int msgflg)
/* msqid: msgget() の返値 */
/* msop: 下記の構造体(msgbuf)へのポインタ*/
/* msqsz: msgbufのmtextの長さを指定 */
/* msgflg:通常０  */

int msgrcv( int msqid, struct msgbuf *msop,
              int msgsz, long msqtyp, msgflg)
/* msqtyp:取出したいメッセージのタイプを指定  */
/*  */
struct msgbuf {
     long mtype;    /* message type, must be > 0 */
    char  mtext[1]; /* message data */
}
/* mtype: メッセージタイプの指定(0 以外を指定する）*/
/* mtext: メッセージ内容：任意の文字列 */
```

図 7・27　メッセージのシステムコール

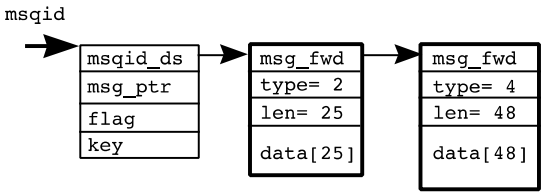

図 7・28　メッセージキューのデータ構造

ージの送受信についてはいろいろな選択が可能であるが，基本的な機能はここに説明したとおりである．

一方，メッセージを受け取るプロセスは，既存のメッセージキューを msgget( ) で利用可能にし，msgrcv( ) で指定したタイプのメッセージを先頭から受け取ることができる．メッセージの送受信についてはいろいろな選択が可能であるが，基本的な機能はここに説明したとおりである．

(2)　セマフォ

System V のセマフォの仕様を**図 7･29** に示す．ここでのセマフォは非負の整数変数値の集

合である．ここで「集合」という意味は，一般的なセマフォを実現するために，複数の排他的な資源の制御を同時に達成することを可能とし，複数の変数のことである．この機能により，デッドロックの回避方法において説明したように（7-4-3 項），複数の資源の同時要求が可能になる．

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/sem.h>

int semget(key_t key, int nsems, int semflg)

int semop(int semid, struct sembuf *sops, unsigned nops)

int semctl(int semid, int semnum, int cmd, char *arg)

struct sembuf{
     short  sem_num;    /* semaphor number */
     short  sem_op;      /* operation */
     short  sem_flg;     /* options    */
}
```

図 7・29　排他制御するセマフォの仕様

図 7･29 における semget( ) ではプロセス間に約束した key の値を指定し，また上記に説明した「集合」となるセマフォの数を nsems で指定する．新規のセマフォを作るときはゼロを設定してはならない．ここでゼロを指定すると，既にできあがっているセマフォの識別子（Semid）を受け取ることになる．

セマフォの操作は semop( ) で行う．ここでは，semget( ) で受け取った識別子を第 1 引数として指定する．複数のセマフォを操作するのが一般的なので，その数を nops で指定する．この数は，第 2 引数である sembuf の構造体配列の数を指定する．

そこで，sembuf では，各々のセマフォ（Sem_Num）に対して，sem_op によって操作を与える．sem_op の値が正なら sem_op の値がセマフォの現在値に加算される．つまり，解放（V-operation）である．逆に，負の値ならば現在のセマフォの値にこの負の値を加算し，それが負になるならプロセスを停止する．つまり，要求し待ち状態化する（P-operation）ことになる．sem_op がゼロの場合は，現在のセマフォ値がゼロになるまでプロセスを停止することを意味する．

上記のセマフォに関する操作には，先に説明したアトミックオペレーションが約束されている．

(3)　共有メモリ

プロセス生成時には各プロセスごとに独立したメモリ領域が作られる．プロセスに与えられるメモリ領域をアドレス空間（Address Space）ということもあるし，プロセスの固有領域であるため，プライベートメモリ領域（Private Memory Ares）と呼ぶこともある．これらのメモリ領域の構成は同じ UNIX でもインプリメントごとに異なっている．

図 7･30 には二つのプロセス間での情報転送の例を示す. この図では各プロセスには独立し

たメモリの領域が与えられているので，情報の転送は両プロセス空間へのアクセス権限があるカーネルだけが実行できる．このような方法では，カーネルが介在するためにシステムコール割込みを伴うので，余分な CPU を消費すること，ならびに同一の情報をシステム内に二重にもつ，という容量的な損失ともなる．

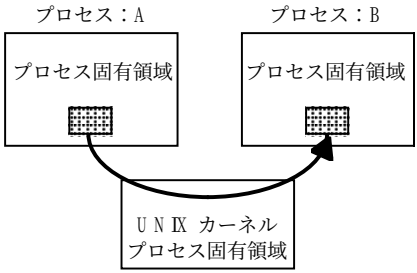

図 7・30　プロセス間情報転送

このような問題を解決するために，共有メモリが考案された．この場合の方法は**図 7･31** に示すとおりである．つまり，メモリ空間のある範囲をシステム共通領域として確保し，複数のプロセス間の情報通信を行う領域として利用する方法である．本方式は，仮想記憶の技術を利用することにより巧妙に実現される．

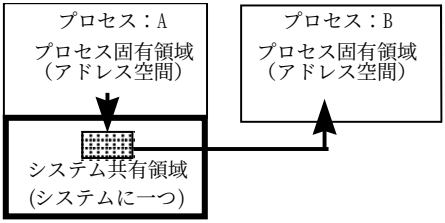

図 7・31　共有メモリによる情報転送の概念

もし，共有メモリを直接プロセス間でアクセスが可能ならば，図 7･30 のようなカーネルを介したプロセス間の情報転送のオーバヘッドが削減され，また，一つの情報だけをシステムに作るだけで済むため記憶容量のむだもなくなる．欠点としては，システム共通領域の分だけプロセス固有領域が小さくなってしまうことである．しかし，現代のコンピュータではアドレス空間は 32 ビット，あるいはそれ以上であることが多く，現実的には問題にならない．

共有メモリを利用するシステムコールの仕様は**図 7･32** に示すとおりである．shmget( ) で共有メモリの使用を宣言し，プロセス間で約束した key を指定し，またアクセス権を設定する．カーネルは共有メモリに領域が確保できれば共有セグメント（Shared Segment）を割り当て，共有メモリ識別子として shmid を割り付ける．一般的に，コンピュータでは論理的なデータの集合をセグメントと呼んでいる．ここでもその意味でセグメントとネーミングしている．

各プロセスセから共有セグメントへアクセスするために，プロセス固有領域内に窓

（Window）を作成する．この窓のサイズは共有セグメントと同じで，窓の参照が共有セグメントの参照となる機構ならば操作が容易となる．

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/shm.h>

int shmget(key_t key, int size, int shmflg)

char *shmat(int shmid, char *shmaddr, int shmflg)

int shmdt(char *shmaddr)

int shmctl(int shmid, int cmd, struct shmid_ds *buf)
```

図 7・32　共有メモリを使用するシステムコールの仕様

このために，共有セグメントと窓との対応をとる機能が shmat( ) である．shmat( ) の第 2 パラメータをゼロで指定するのが基本的な使い方であり，このとき，shmat( ) の処理は，共有セグメントと同じ大きさのメモリ領域をプロセス固有領域に見つけ，その先頭アドレスを呼び出し，元に返す．つまり，仮想的な窓を取り付ける（Attach）のである．

この操作をプロセス交信するすべてのプロセスが行うことで，**図 7･33** に示すようにプロセス固有領域内に共通セグメントをアクセスする窓が取り付けられることになり，自プロセスの領域のアクセスにより共通セグメントへのアクセスが実質的に行えることになる．このトリックは仮想記憶管理のマッピングテーブル操作によって行われる．

図 7・33　共有メモリへのアクセス法

shmdt( ) は固有領域内に shmat( ) でアタッチした領域を解放するデタッチ（Detach）操作である．shmdt( ) は単にアクセス窓を解放するだけであり，共有セグメントの解放は行わない．この解放は，shmctl( ) で行われるので共有セグメント領域を有効利用するためにも利用が完了したときは実行しておかねばならない．

共有メモリをプロセス間通信に利用する典型的な例は，データベース管理システム（DBMS：Database Management System）である．DBMS を利用している OLTP（On Line Transaction Processing）の代表としては，銀行オンラインシステムや列車，飛行機などの座席予約システムなどがある．つまり，共有メモリの利用法は，多くのシステムプログラムの設計場面に必須な機能となっている．

7-5-8 シグナル

(1) 目 的

パイプ，メッセージなどはプロセス間で相互に約束を明示的（Explicit）に行った通信である．しかし，マルチタスキング環境で並列処理を行っている際に，必ずしも明示的な通信を行う必要がない場面もある．一つの例として，あるプロセスが無限ループ（Dynamic Loop）に陥ってしまったようなバグが露呈したときは，明示的な通信手段が取れない．このような場合は，該当プロセスに対してカーネルがシグナル（Signal）を送り，プロセスを強制的に中断させる手段が望まれる. シグナルは, プロセス間の割込みであり,「ソフトウェア割込み」とも呼ばれる.

(2) 基本的な考え

各プロセスにシグナルの割込み機構を備えることができる．ハードウェアの割込みと同様に，割込み処理ルーチン（関数）をプログラマは作成し，その関数のアドレスをカーネルに通知しておくことができるのである．この機能がシステムコール signal( ) であり，シグナルを捕獲（Catch）する手段となる．

一方，ソフトウェア割込みを発生させる機能はシステムコール kill( ) である．kill( ) にはシグナルを送るターゲットのプロセス識別子（PID）とシグナルの種別を指定することになる．ほかのプロセスから kill( ) が実行された際に，該当する割込み処理ルーチンをあらかじめカーネルに通知しておかないと，未定義の割込みが生じたことになる．ハードウェアの未定義割込みと全く同様のことである．

図 7･34 に signal( ) と kill( ) システムコールの関係を示す．この図では，捕獲するシグナルは SIGaaa，SIGbbb，SIGccc である．そして，各シグナルの処理をする関数が a( )，b( )，c( ) であることをカーネルに伝えている．シグナル SIGddd は処理をカーネルのデフォルト（Default）に任せること，SIGiii はシグナルが発生しても無視することをカーネルに宣言している．また SIGddd は処理をカーネルの標準的処理であるデフォルト（Default）に任せること，SIGiii はシグナルが発生しても無視することをカーネルに伝えている．

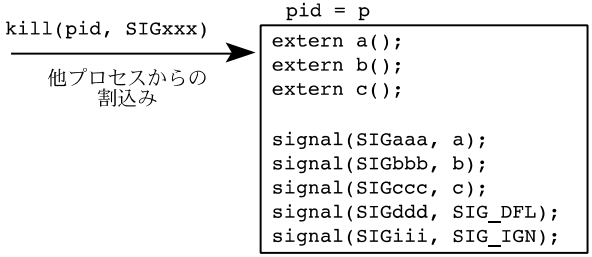

図 7・34 シグナルの送信と捕獲，無視，デフォルトなどの指定方法

(3) シグナル受信と送信の分類

プロセスはシグナルに対して以下の 3 種類の処理を指定できる．この指定は，signal( ) の第 2 パラメータで指定する．

・捕獲する：関数へのポインタを指定する．

・無視する：SIG_IGN を指定する．
・デフォルトの処理にまかせる：SIG_IDFL を指定する．

上記の具体的な指定は，図 7･34 に示したとおりである．

「無視する」と指定をしておけば，該当するシグナルが送られてきたときにプロセスが終了してしまうことを防げる．また，「デフォルト」という処置は UNIX カーネルの標準的な処置であり，一般的にプロセスの終了となるが，なかには無視と同じ処置を行い，プロセスをブロック（待ち状態にする）する場合もある．ただし，シグナル番号 9 の SIGKILL は捕獲して無視することはできない．SIGKILL が送られてくるとプロセスの終了となる．

一方，シグナルを UNIX で送るケースをまとめてみると以下のようになる．

・ハードウェア的に過ちが検出されたときカーネルより kill( ) が実行される．
　＜浮動小数点演算誤りの SIGFPE，メモリ例外の SIGSEGV など＞
・kill コマンド処理プログラムの中で実行される．
・端末からの特定の文字入力により実行される（kill( )）．
　＜割込み文字，中断文字，などの入力＞
・カーネルがソフトウェア的な処置として実行する場合．
・kill( ) をマルチタスキング環境でユーザプログラムから意図的に実行する．
　＜システムの終了などもこの一つ＞

(4) シグナルの仕様

signal( ) とkill( ) のシステムコールの仕様を**図 7･35** に示す．signal( ) では，第 1 パラメータsignumはシグナル番号 [*3] であり，これは，/usr/include/signal.hに定義されている．また，第 2 パラメータは先に述べた 3 種類であり，無視する処理のSIG_IGN，カーネルのデフォルト処理のSIG_IDFL，シグナル処理関数へのポインタである．

```
#include  <signal.h>

void (*signal(int signum, void (*handler)(int)))(int);

int kill(int pid, int signum);
```

図 7･35　signal と kill の仕様

システムコール kill( ) では，正の PID 値を指定したときは実効ユーザ ID が同一のプロセスにしか kill( ) を送ることはできない．例外はス―パーユーザだけである．つまり，ス―パーユーザはすべてのプロセスに対して任意のシグナルを送ることが可能なのである．

kill( ) の実行は一つのプロセスファミリ内が原則である．しかし，この原則では kill( ) の使用範囲が限定されてしまう．そこで，kill( ) には，プロセス識別子（PID）の指定により特定の機能が用意されている．**表 7･2** にその一覧を示した．プロセス識別子がゼロ，－1，負の値のときに特別の意味をもたせている．ここで意図していることは，マルチタスキングの環境でのプロセス間シグナル通信であるため，プロセスファミリ内に対してシグナルを送る機能や，ス―パーユーザの絶対的な権限の発揮などにある．

[*3] signum に関する詳しい説明は UNIX のマニュアルやオンラインマニュアルを参照されたい．

表 7・2　kill( ) における pid の指定と意味

pid の値			kill(pid, signum) の機能
正			指定したpid のプロセスにシグナルを送信するが実効ユーザIDが同一でなければならない．例外はスーパユーザだけ．
0			全プロセスグループに対してシグナルを送る．したがって，バックグラウンドプロセスを殺すことが可能．
負	－1	SU	pid=0,1 (スワッパー，init)以外の全プロセスにシグナルを送れるので，SIGTERMを全プロセスに出しshutdown を行える．
		非SU	同一のユーザIDを持つすべてのプロセスにシグナルを送る．ユーザ自身のプロセスファミリーすべてを殺せる．
			絶対値に等しいプロセスグループのすべてのプロセスにシグナルを送る．サブシステムの終了に使う．

(5)　関連システムコール

シグナルに関係したシステムコールは pause( )，alarm( ) である．**図 7･36** にその仕様を示す. ファイルの読出しのシステムコールである read( ) をパイプに適用したときやメッセージの受取りの msgrcv( ) などは，相手プロセスから情報が供給されないとカーネルによりブロックされる．また，wait( ) も子プロセスの完了までブロックされる．しかし，pause( ) システムコールはプロセスを自分で待ち状態にする機能であり，それ以外の目的はない.

```
#include <unistd.h>
int  pause(void)  /* waiting a signal       */
                  /* return value always -1 */

unsigned int alarm(unsigned int nsec)
                  /* set alarm clock        */
```

図 7・36　シグナルに関連するシステムコール

したがって，pause( ) は何らかのイベントをシグナルで受け取ることを前提に使うことになる. この場合, イベントが発生してシグナル捕獲関数が完了し，リターンすると pause( ) は解除され次のステートメントに制御が移る．もし，イベントを繰り返して待ち，シグナル捕獲関数を再度実行したいならば，再度 pause( ) を実行するようにすればよい.

システムコール alarm( ) はアラームクロックのことであり，引数 nsec で指定された秒数の時計をセットする機能である．指定した実時間（Real Time）が経過すれば，signal( ) で指定した SIGALRM の捕獲関数に制御が渡ってくる．この時間は，実時間であって当該プロセスが CPU 消費した時間ではない. 本システムコールは自分自身にシグナルを送る代表例である.

時計の割込みイベントはプロセスに一つしか許されないので，alarm( ) を最後に実行した nsec 値が有効となる．したがって，nsed=0 を指定すると本プロセスのアラームクロック設定はなくなる.

複数の alarm( ) を実行すると，前回 alarm( ) で指定した nsec の残り時間を返り値として得ることができる．この値を alarm( ) に用いれば，前回設定した nsec の残り時間をあらためてセット可能である．このよい例がライブラリ sleep( ) の処理である．sleep(nsec) は指定した

nsec 秒間プロセスをブロック状態にする．sleep( ) の処理は，alarm( ) を使用することで処理可能である．以下，具体的な例で説明する．

図 7･37 に示すように，はじめに alarm(4) で 4 秒後にアラームクロックをセットし，その直後に sleep(1) で 1 秒間プロセスをブロックするとする．このような場合，sleep( ) 内では，remt = alarm(1) を実行するが，その返り値としては remt = 4 が得られる．したがって，sleep( ) 内では pause( ) して 1 秒後に割込みを得たら，残りの時間として remt = remt－1（つまり，3 秒後）を求め，alarm(remt) を実行し，処理を完了させる．この結果，図 7･37 のように，相対時刻 4 秒の時点で SIGALM のシグナルを受信することになる．

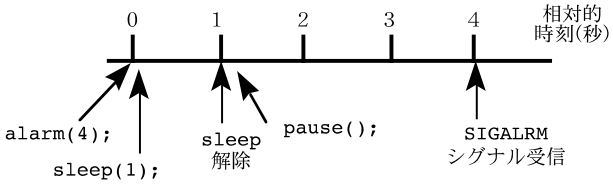

図 7・37　alarm(), sleep() **のアラームクロック関係**

■7群 － 3編 － 7章

7-6 演習問題

（執筆者：吉澤康文）［2013年2月 受領］

(1) UNIXのシェルはなぜカーネル外のプロセスとして実行されるのか.

(2) パイプの利用効果を述べよ.

(3) 双方向パイプの利用が有効な応用を考えてみよ.

(4) プロセス制御の排他制御を行うべき応用例をあげよ．また，その理由を述べよ.

(5) プロセス間通信の手法を複数述べ，それらの各々の方法を説明せよ.

(6) 単一プロセッサとマルチプロセッサとでは排他制御の実現にどのような違いがあるのか考察せよ.

(7) デッドロックが発生する状況はいかなるときか例を示して説明し，その解決方法も述べよ.

(8) デッドロックを回避する方法を述べよ.

(9) UNIXのシグナルはどうしてソフトウェア割込みと呼ばれるのか.

(10) 親子のプロセスで通信を行う以下のプログラムを作成せよ.
- パイプを作成し，パイプによる相互通信のプログラムを作成せよ.
- 上記において親プロセスが端末に出力（stdout），子プロセスが受信（stdin）を行い，子プロセスが端末に出力（stdout）するプログラムを作成せよ.
- 更に工夫して，双方向パイプを作成せよ．そこでは，子プロセスは端末出力をパイプをとおして親プロセスにデータを渡す方法を実現する．つまり，子プロセスは入力（stdin）も出力（stdout）も共にパイプを利用する．親プロセスは入力（stdin）をパイプにする.

(11) 上記のプログラムができたら，親のプログラムで端末から入力し，その情報を子に転送し，子はその情報を端末出力し，それに対する応答を入力してメッセージを親に伝えるような対話プログラムを作成せよ（親子の入出力が分かるように工夫せよ）.
- 基本形となるプログラムを作成する.
- 修了の形式を考える.

  ファイルを子のプロセスに送り，内容を処理する（例えば，特定の文字を数えるとか，数字を加算して答えるなど）プログラムを作成する（並列処理的は発送のプログラム）.