13 セマフォア

—同期制御の一般的・実際的な手法—

先の第12節においては共有資源が1個の場合にしぼって排他制御の基本的な考え方について説明した。

ここでは共有資源が2個以上の場合にも適用でき、実際に排他制御が必要な時によく使われる、セマフォアと呼ばれる手法について説明する。

次の論文は

プロセス同期に関する古典的な文献で、その中でセマフォア, クリティカルセクションの問題, デッドロックの除去が議論されている。

E.W.Dijkstra(1968a), "Cooperating Sequential Processes," in *Programming Languages* (F.Genuys, ed.), pp.43-112, Academic Press.

セマフォア $\cdots$ 鉄道で進入可能か否かを示す腕木信号機; 一般に旗・ライトなどによる信号装置; 手旗信号

13–1 セマフォアとは何か

次の2つの操作 P(sem)，V(sem) によってのみ値が変更される、プロセス間に共通の整数型変数を一般にセマフォアと呼ぶ。ここで、semはセマフォアで、空いている共有資源の数を表す。

P(sem) 操作 : ....(資源使用の申し出に対するOS内の処置)

```
if (sem ≥ 1) {
   (1)sem ← sem−1;
   (2)return; /*P(sem)操作を呼んだ(i.e.今実行している)プロ*/
                /*セスに資源を割当て、プロセスの処理を継続する*/
} else {
   (1)このP(sem)操作を呼んだ(i.e.今実行している)プロセスを
      (1.1)(semが関与する)共有資源の空きを待つ待ち行列に入れる;
      (1.2)実行中から待ち状態に移す;            資源ごと
   (2)制御をディスパッチャに渡す;
}
```

V(sem)操作 : ....(資源返却の申し出に対する処置)

```
if (共有資源の空きを待つプロセスがある) {
    (1) その待ち行列からプロセスを1個取り出し実行可能状態にする;
            /*返却された資源を取り出したプロセスに割り当てる*/
    (2) V(sem)操作を呼んだ(i.e.今実行している)プロセスも
        実行中から実行可能状態に移す;
    (3) 制御をディスパッチャに渡す;    次の実行プロセスを決める
} else {
    (1)sem ← sem+1;   /*資源を解放*/
    (2)return;   /*V(sem)操作を呼んだ(i.e.今実行している)*/
                 /*プロセスの処理を継続する*/
}
```

補足：

- P(sem) 操作と V(sem) 操作をまとめて **PV 操作**と呼ぶこともある。

- P と V はDijkstraの母国語であるオランダ語の “Passeren”(パスを許す) と “Verhoog”(起こす) に由来する。

- 資源数が 1 で sem=1 と初期設定される場合は、P(sem) 操作と V(sem) 操作は各々12.3.2 節で議論した lock( ) と unlock( ) に相当するもので、バイナリ・セマフォアと呼ばれる。

- セマフォア (sem) 自身がプロセス間で排他制御が必要な共有資源であり、PV 操作を同時並行的に実行することは許されない。

  ⇨ lock( ) の場合と同様に、P(sem) 操作も V(sem) 操作もアトミックな (i.e. 分割不可能な) 操作として実装されなければならない。

さらに補足：

セマフォアの様に、着目しているシステムに関連したイベントの生起の履歴を1つの変数/データ構造の形に総合したものをイベント変数と呼ぶ。並行プロセスを考えている場合には、イベント変数は次の様なイベントの生起によって値が変わる。

- プロセスがクリティカルセクションに入る。
- プロセスがクリティカルセクションから出る。
- 新しいプロセスが始まる。
- 入出力操作が始まる。
- 入出力操作が完了する。
- メッセージを受け取る。
- プロセスが資源を確保する。
- プロセスが資源を解放する。

13–2 基本的なプロセス統合問題

第8.3節「多重タスキングの利点」(p.72) からの抜粋：

利点 (1) システム全体の処理効率が上がることがある。

例8.1 ... 多重プログラミング
例8.2(入出力とプログラム本体)
入出力と計算の部分を別々のプロセスとして実行すると、
- CPU と入出力資源を各々のプロセスで独立に使用でき、
- 装置が無駄に遊ぶのを防げる。

利点 (2) 信頼性向上に繋がる。

- fail-soft なシステムを容易に構築できる。
  具体的には、異常が起こっても、異常の起こったプロセスだけを終了させ残りのサービスは続行できる。
- fault-tolerant なシステムを容易に構築できる。
  具体的には、同一の仕事をするプロセスを複数用意しておけば、1 つが異常終了しても残りの 1 つが動作してくれる。

利点 (3) 複数のCPUによる多重プロセシング環境においても、ソフトウェアを再設計することなく直ちにハードウェアに見合った性能向上が期待できる。

システム全体の性能向上のためにも信頼性向上のためにも多重タスキングは有効であるが、全てのプロセスが全く独立に動作する訳ではない。

共有資源の排他制御を始めとして、複数のプロセスをうまく協調／同期させなければならない。

⇨ この節では、OSの実装の際に遭遇する
幾つかの代表的なプロセス統合の問題を挙げ、
各々どの様にプロセスの協調を図れば良いかを説明する。

例示

13–2–1 排他制御問題

排他制御問題 (相互排除) :

2つ以上のプロセスが共用資源を同時に使用しない様にすることを言う。

セマフォアを用いた排他制御 (共用資源が1個の場合) :

12.3.2節で紹介した lock( ), unlock( ) の場合と同様である。

すなわち、セマフォア mutex を mutex=1 と初期設定した上で、全てのクリティカル・セクションを次の様に書き換えれば良い。

```
P(mutex);        /* 資源を確保 */
```

クリティカル・セクション

```
V(mutex);        /* 資源を解放 */
```

セマフォアを用いた排他制御(同等の共用資源が1個以上の場合):

共用資源が1個の場合の簡単な拡張で済む。

すなわち、セマフォアmutexを　mutex=共用資源の個数　と初期設定した上で、全てのクリティカル・セクションを次の様に書き換えれば良い。

```
P(mutex);       /* 資源を確保 */
```

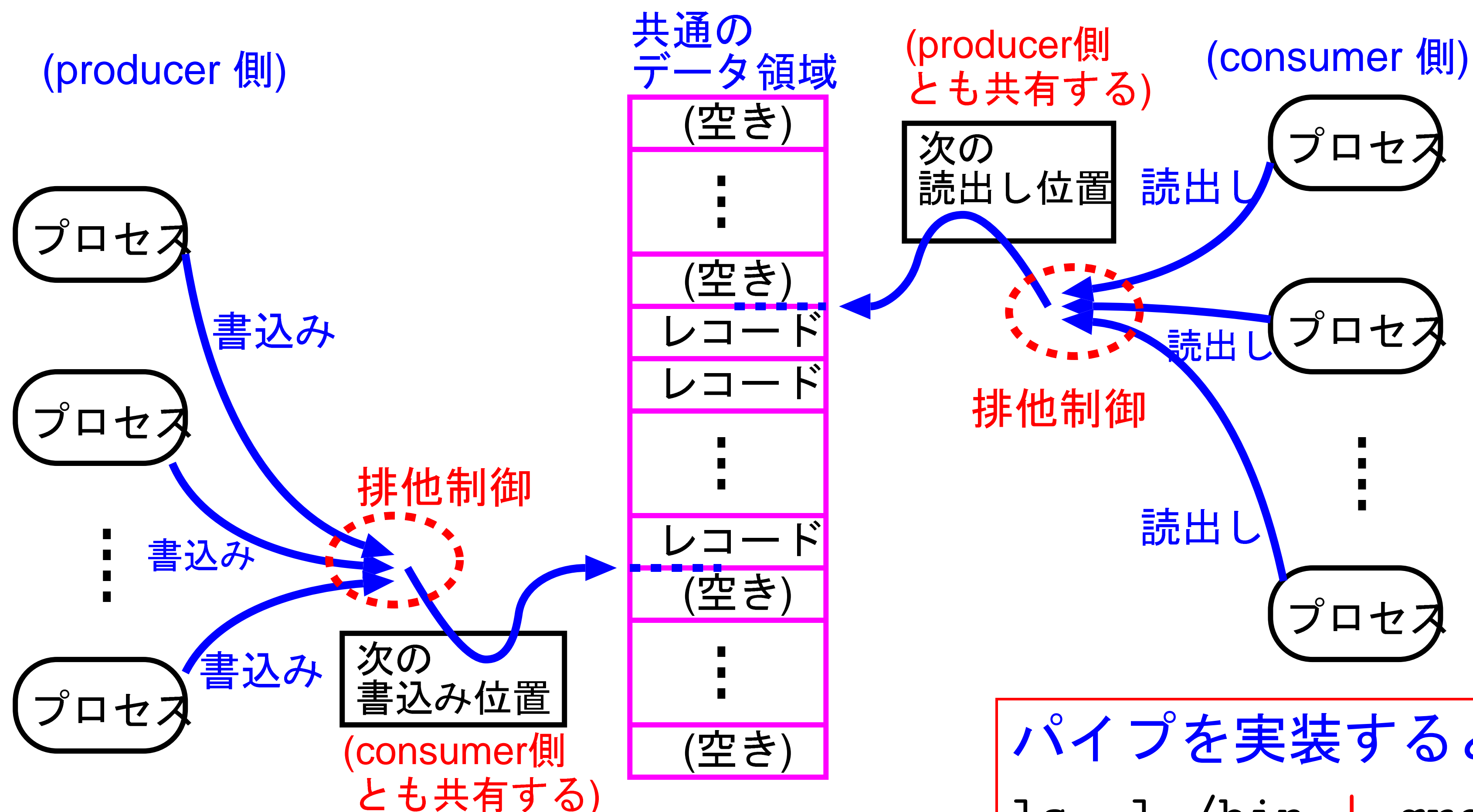
クリティカル・セクション

```
V(mutex);       /* 資源を解放 */
```

13–2–2 生産者と消費者の問題

生産者と消費者の問題：

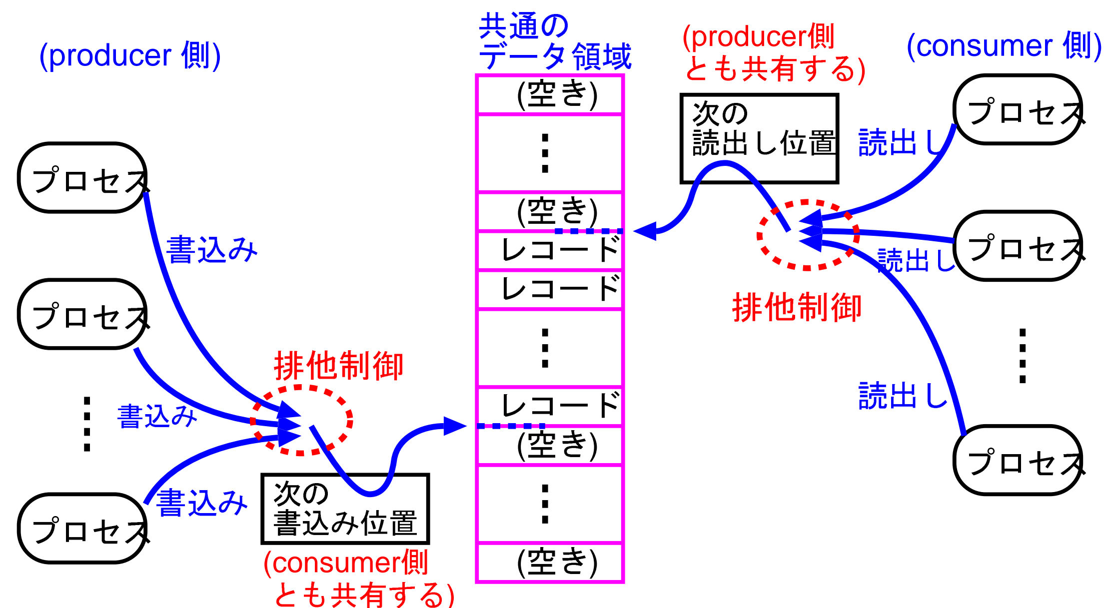
producer と呼ばれる一連のプロセスが共通のデータ領域にレコードを次々と書き込み、一方では書き込まれたレコードは別の **consumer** と呼ばれる一連のプロセスが書き込まれた順に読み出して処理してゆく。



パイプを実装すると...

```
ls -l /bin | grep sh
```

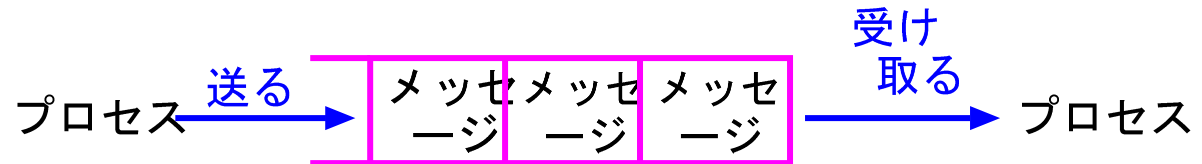
⇨ producer間の協調,
consumer間の協調,
producer-consumer
間の協調
が必要になる。



- データ領域にレコードを書き出す際、producer間で排他制御。
- データ領域からレコードを読み出す際、consumer間で排他制御。
- データ領域が一杯の時は、producerプロセスはレコード書き込みの処理を一時停止し、空きが出来た時点ですぐに書き込みを再開。
- データ領域が空の時は、consumerプロセスはレコード読み出しの処理を一時停止し、新しいレコードが出来た時点ですぐに読み出しを再開。

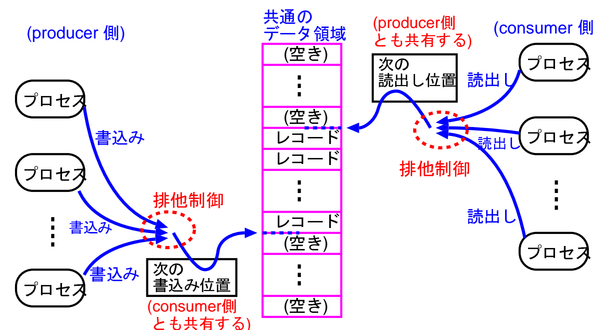
プロセス間のメッセージ通信を実現する方法として、
次のようなものがある。

- メッセージを入れるために空のバッファを1個以上用意する。
- メッセージを送る側と受ける側の間には、
  メッセージの入ったバッファの待ち行列を想定する。



- メッセージを送る側は、メッセージを送りたい時は空のバッファにメッセージを入れ待ち行列の最後尾に付け加える。
- メッセージを受ける側は、待ち行列の先頭からバッファを取り出すことによって、次のメッセージを読む。

これは、**producer**(送り手)と**consumer**(受け手)が各1個の場合の「生産者と消費者の問題」に他ならない。



セマフォアを用いたproducerとconsumerの実装(各1個の場合):

(producer間, consumer間の排他制御) ⇨ 不要

(データ領域が一杯の時の協調)

- 空き領域がproducerにとっての共有資源であるので、「空き領域の大きさ」を記憶する共有変数`n_vacant` をセマフォアとして用意する。

  ⇨ `n_vacant=0`が「利用可能な資源なし」に相当

- 領域を解放するのはconsumerであり、`n_vacant` はconsumerがレコードを処理する度に増える。

⇨ `P(n_vacant)`操作を実行するのはproducer、
`V(n_vacant)`操作を実行するのはconsumer

(データ領域が空の時の協調)

- 書き込み済のデータ領域がconsumerにとっての共有資源であるので、「書き込み済領域の大きさ」を記憶する共有変数n_occupied をセマフォアとして用意する。

⇨ n_occupied=0が「可能な資源なし」に相当

- このセマフォアに関しては、書き込み済領域の生成が「共有資源の解放」に相当し、これを行うのはproducerになる。n_occupiedはproducerがレコードを生成する度に増えていく。

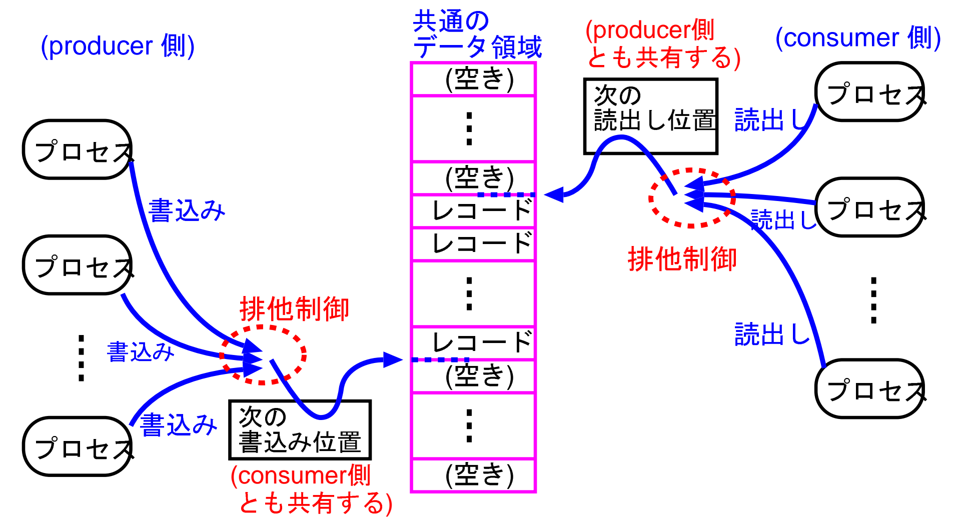
⇨ P(n_occupied)操作を実行するのはconsumer、
V(n_occupied)操作を実行するのはproducer

以上の方針に従えば、producerとconsumerは次の様に実装することが出来る。

```
#define  TRUE  1

/* データ領域 */
#define  N  50
struct record {
  ......
};
struct record Buffer[N];

/* セマフォア */
typedef  int  Semaphore;

Semaphore n_vacant  =N;    /*データ領域が一杯の時の協調のため*/

Semaphore n_occupied=0;      /*データ領域が空の時の協調のため*/
```

```
/*------------------------------------------------------
/* producerのプロセス */
void producer(void)
{
  int next_vacant=0;    /* 次の空き領域の番号 */

  while (TRUE) {
    P(n_vacant);
    Buffer[next_vacant]の上に新しいレコードを構成する;
    V(n_occupied);
    next_vacant = (next_vacant+1)%N;
  }
}
```

```
/*-----------------------------------------------------------
/* consumerのプロセス */
void consumer(void)
{
  int next_occupied=0;              /* 次の書き込み済領域の番号 */

  while (TRUE) {
    P(n_occupied);
    Buffer[next_occupied]内のレコードを読み出して
                                                必要な処理を行う;
    V(n_vacant);
    next_occupied = (next_occupied+1)%N;
  }
}
```



セマフォアを用いた producer と consumer の実装 (一般の場合) :

(producer 間, consumer 間の排他制御) ⇨ 必要

⇨ 13.2.1 節の場合と同様に出来る。

(データ領域が一杯の時の協調) ⇨ 「(各 1 個の場合)」と同様に出来る。

(データ領域がの空の時の協調) ⇨ 「(各 1 個の場合)」と同様に出来る。

一般の場合の producer と consumer の実装例を次に示す。

(共有資源を確保する時間を出来るだけ短くする様に工夫した。)

```
#define  TRUE  1
```

複数のプロセス間で変数等を共有するための仕掛けが必要

```
/* データ領域 */
#define  N  50
struct record {
   ......
};
struct record Buffer[N];
int next_vacant   =0;    /*次の空き領域の番号*/
int next_occupied=0;    /*次の書き込み済領域の番号*/
```

複数のプロセス間で共有するので大域変数とした

```
/* セマフォア */
typedef  int  Semaphore;
Semaphore mutex_producer=1;     /*producer間の排他制御のため*/
Semaphore mutex_consumer=1;     /*consumer間の排他制御のため*/
Semaphore n_vacant   =N;     /*データ領域が一杯の時の協調のため*/
Semaphore n_occupied=0;       /*データ領域が空の時の協調のため*/
```

```
/*-----------------------------------------------------------------
/* producerのプロセス */
void producer(void)
{
  struct record new_record;

  while (TRUE) {
    変数 new_record 上に新しいレコードを構成;
            共有資源を確保する時間を出来るだけ短くするために、
            直接Buffer上に書き込むのは避けた
    P(mutex_producer);
    P(n_vacant);
    Buffer[next_vacant]=new_record;
    V(n_occupied);
    next_vacant = (next_vacant+1)%N;
    V(mutex_producer);
  }
}
```

```
/*-----------------------------------------------------------
/* consumerのプロセス */
void consumer(void)
{
  struct record new_record;

  while (TRUE) {
    P(mutex_consumer);
    P(n_occupied);
    new_record=Buffer[next_occupied];
    V(n_vacant);
    next_occupied = (next_occupied+1)%N;
    V(mutex_consumer);
    読み出したレコード new_record に関する処理を行う;
  }
}
```

共有資源を確保する時間を出来るだけ短くするために、
Buffer上のレコードを直接処理するのは避けた

13–3 デッドロック

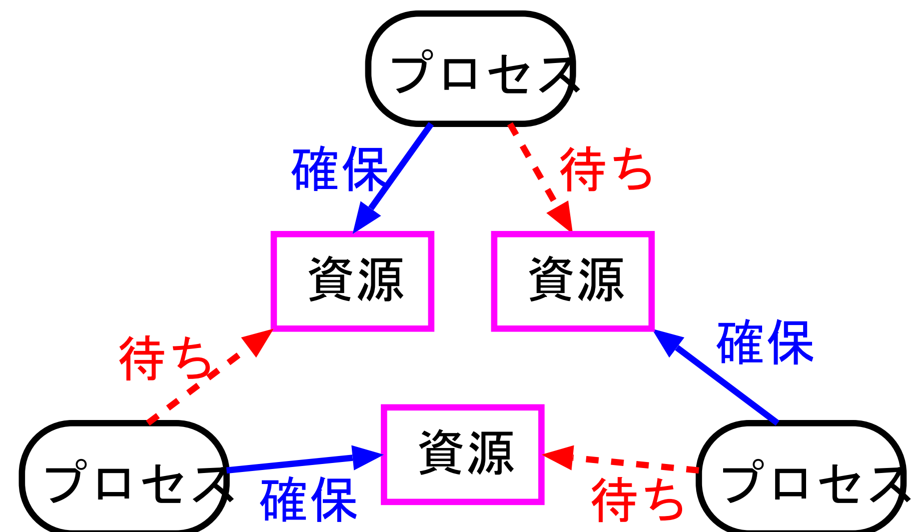
デッドロック：

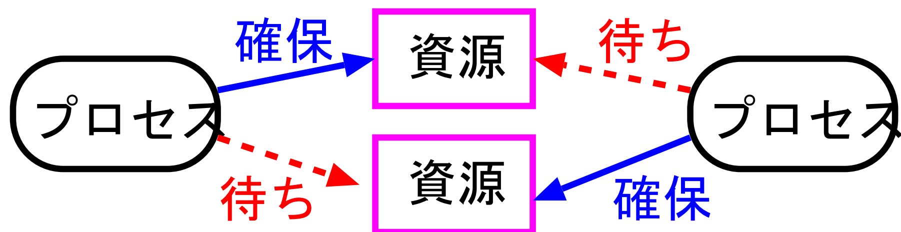
lock( )，unlock( ) や P(sem)，V(sem) といった命令／操作を無雑作に使用したのでは、

全てのプロセスが他プロセスの確保した資源を待ち合って
結局どのプロセスも実行できない、

という厄介な状態に陥ることがある。

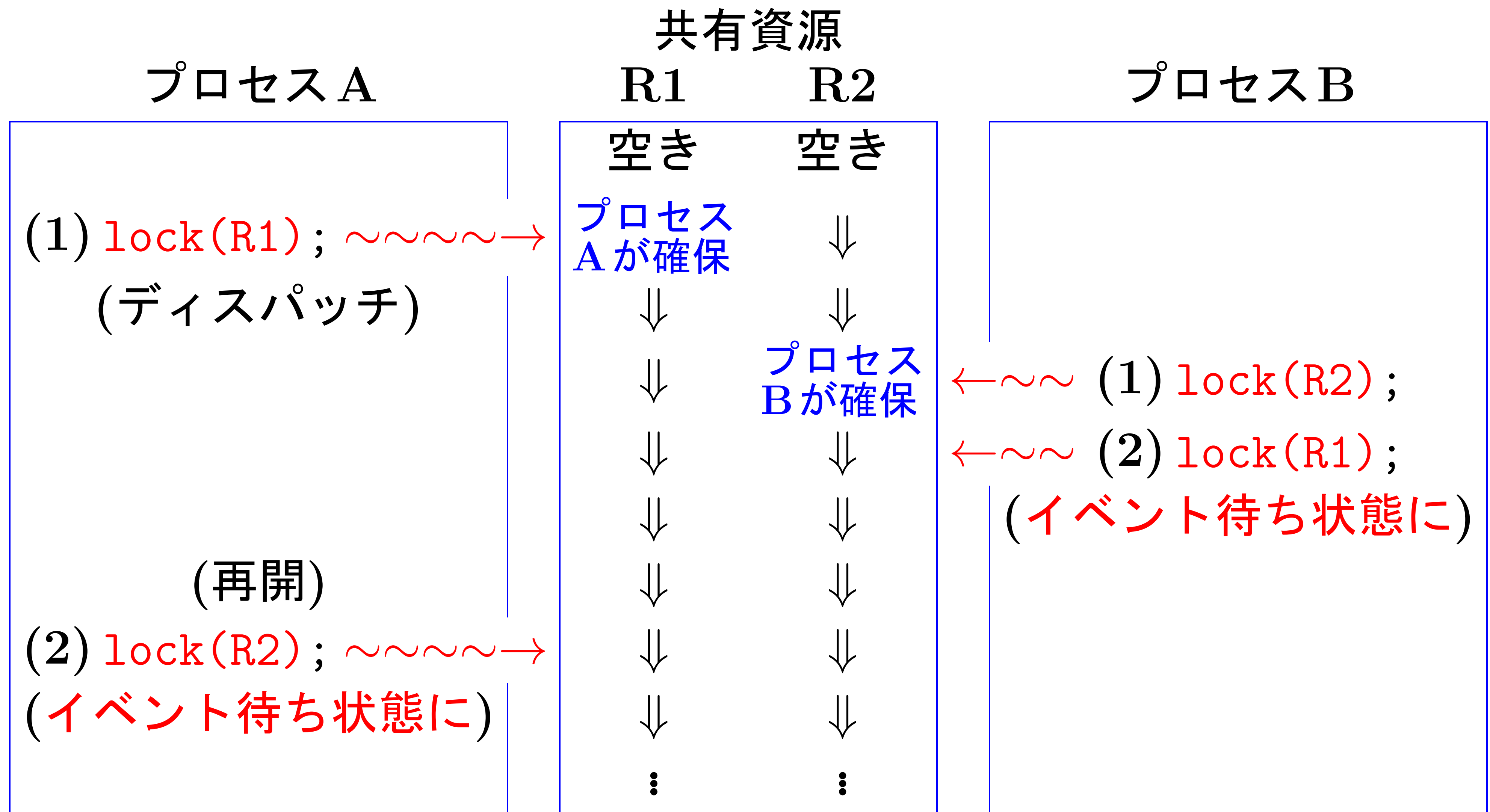
この様な状況をデッドロックと呼ぶ。





例13. 1 (デッドロックの発生)

デッドロックの状況は、例えば次の様にして発生する。



デッドロックの回避：

デッドロックを回避するには、単に次のような方法で資源確保を行うだけで良い。

(方法1) どの資源を先に確保するかの絶対的な優先順位を、全てのプロセス間で共通に定める。そして、複数の資源を必要とする場合は、その共通の順序に従った順序で必要な資源を確保していく。

(方法2) 複数の資源を必要とする場合は、
一度にまとめて資源の確保を試みる。

そのためには、
一度にまとめて資源の確保を試み、
全てが揃わなければ何れも確保しない、
というタイプの操作、言わば
AND条件付P( )操作
をOSが用意する必要がある。

資源確保の際の作法：

プロセスをうまく協調させた上で、システム全体を効率良く運営するためには、さらに次の点に留意すべきである。

- 使用済の資源は直ちに解放する。
- 資源を排他的に使用する時は、資源が空いていることを確かめてから確保の要求を行う。

> **補足：**
> この工夫はp.168の「CS命令を用いたlock()の実装」例の中で見受けられる。
>
> 要するに、lock() にしろ P(sem) にしろ、資源を排他的に確保しようとする命令／操作は内部に Test-and-Set命令 や CS命令の様な、多くのマシンサイクルを要する命令を含んでいるので、頻繁に実行するのは避けた方が良いということ。

- 共用(share)なのか排他的(exclusive)な利用なのかをはっきりと見極めた上で、資源要求を行う。