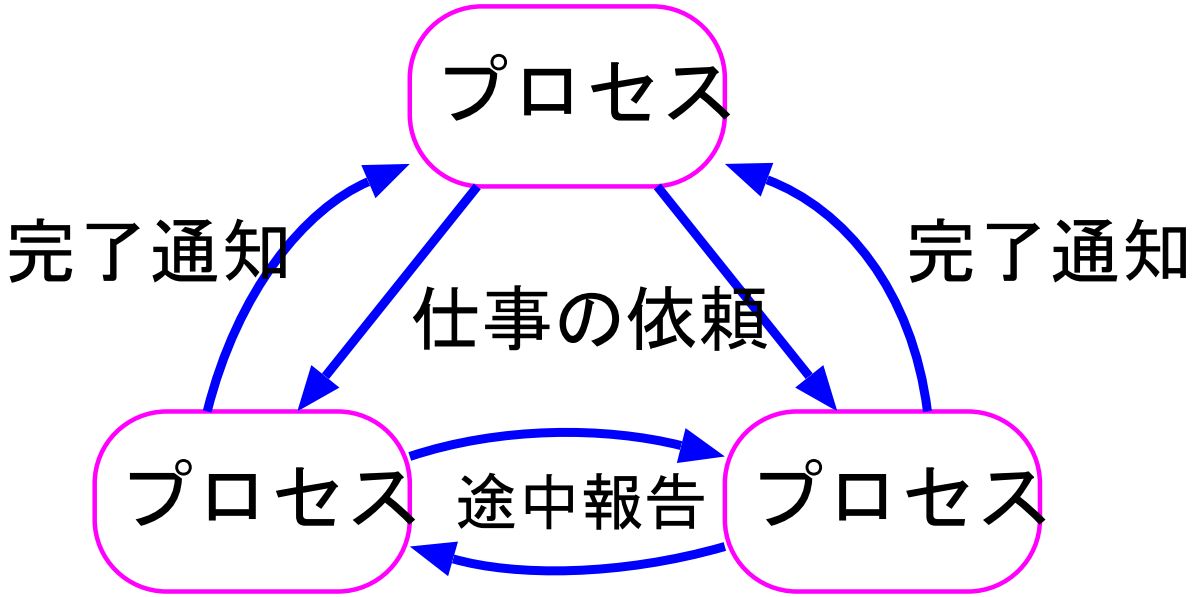
12 プロセス間の相互作用と排他制御

12–1 プロセス間通信の必要性

複数のプロセスが共同作業する場合はプロセス間の通信が必要になる。



並列処理を行う場合、プロセス間の通信は最も基本的な機能。

⇨ **OSがその手段を提供**する必要がある。

- プロセス間の情報のやり取りを**プロセス間通信 (IPC)** と呼ぶ。
- プロセス間通信等の手段を通じて複数のプロセスの実行順序を制御することを**同期制御**と言う。

同期 ··· (岩波情報科学辞典, 前川3.1節より)
ある事象が起きるまでプロセスを待たせることを言う。

同期は共用資源を操作している並行プロセスの間で取られ、排除と協同の2種類に大別できる。

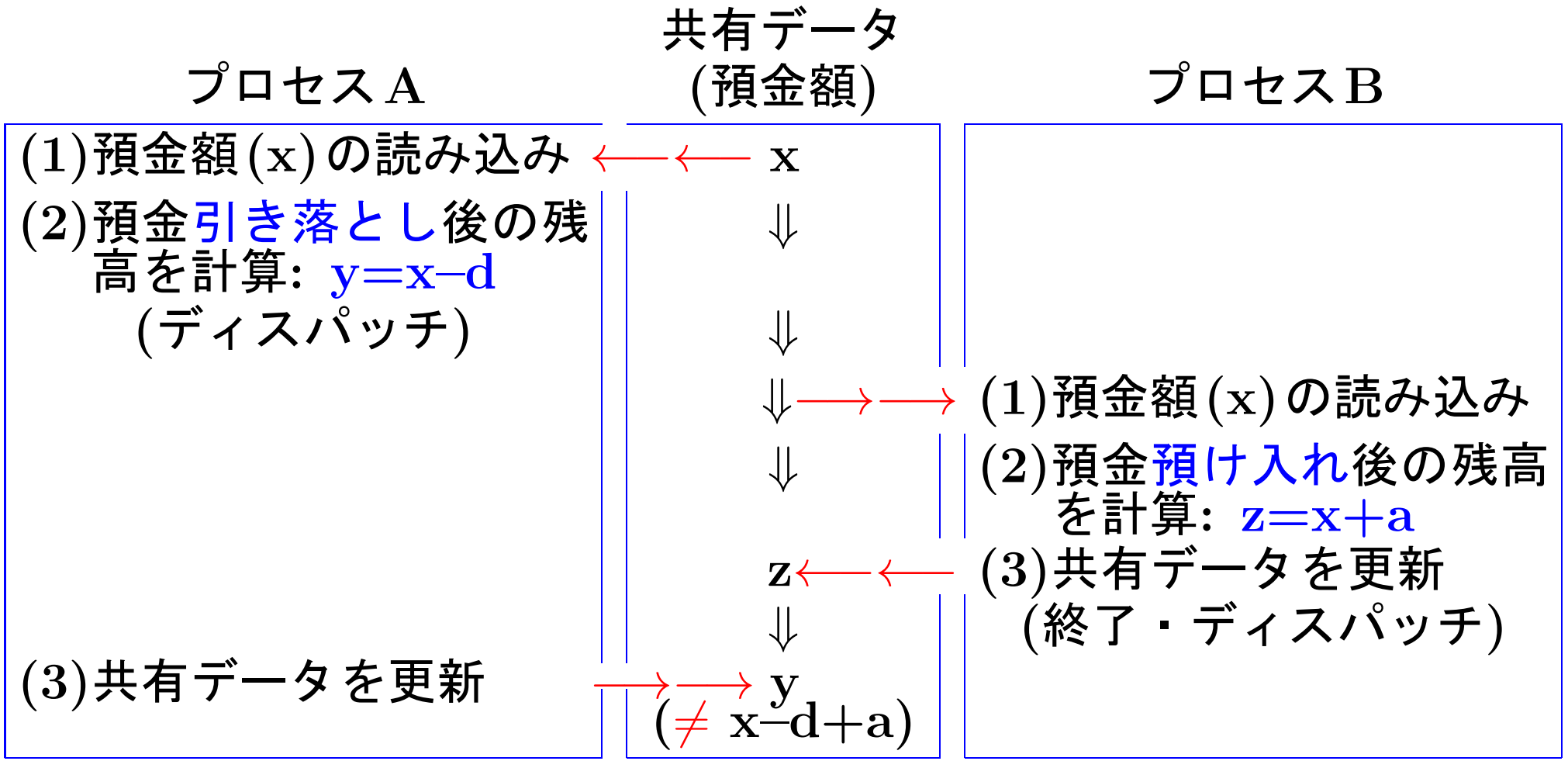
排除 ··· プロセス間に資源の競合がある時、「資源が空くまで」プロセスを待たせる。

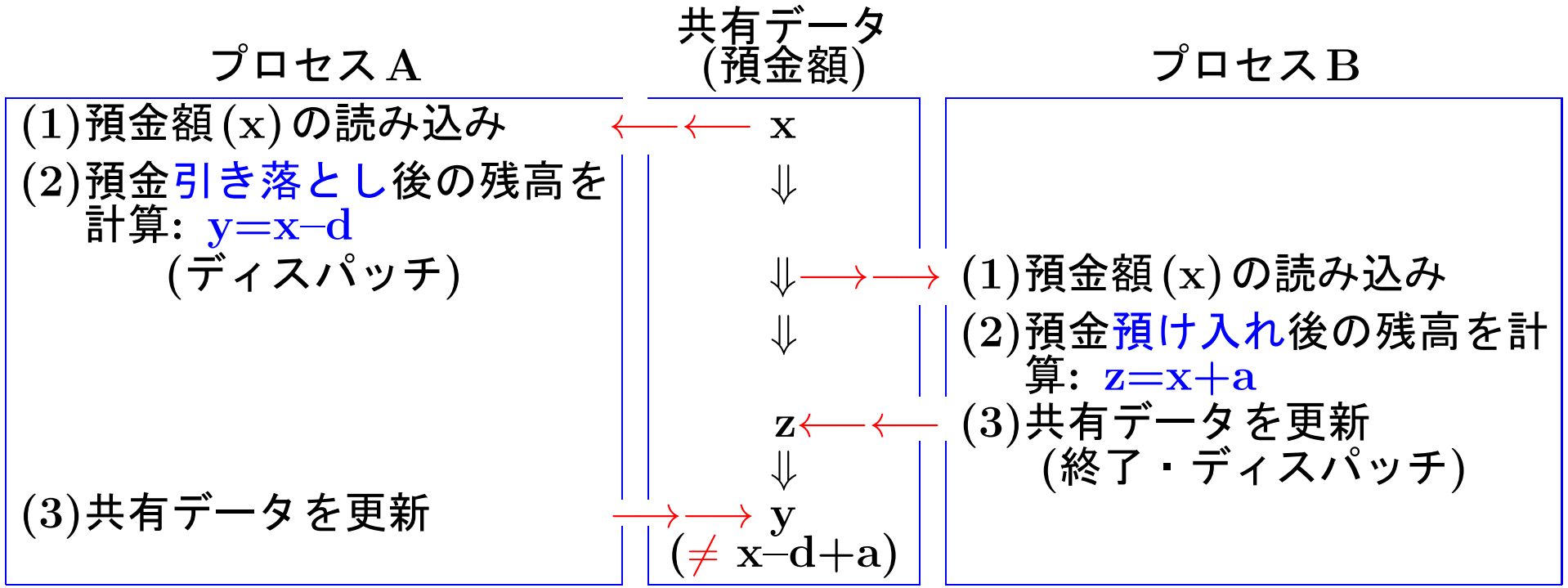
協同 ··· 複数のプロセスが情報交換を行いながら互いに協調して共通の目的のために処理を進める。(情報交換の際に相手のメッセージを待つこともある。)

12–2 排他制御の必要性

複数のプロセスがデータを共有する場合、プロセスの切替え時期が悪いと共有データに矛盾が生ずることがある。

例12. 1 (預金の引出し/預け入れ)



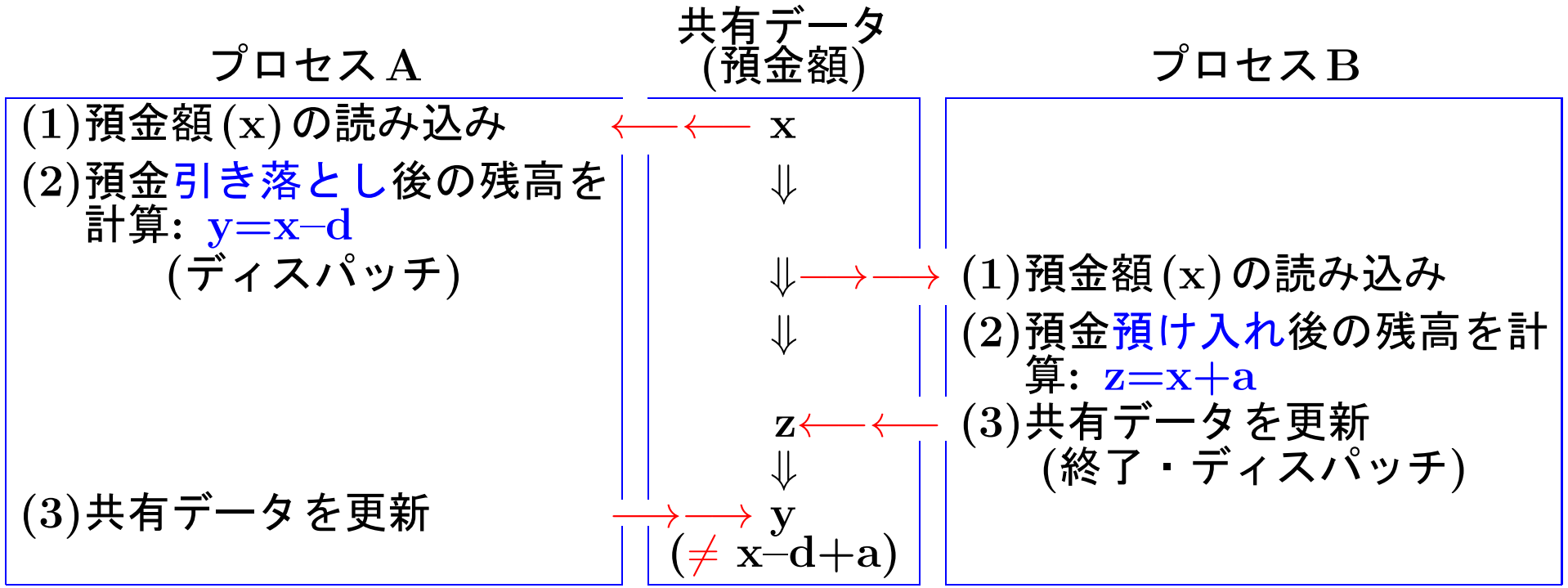


問題点：共有データに対するアクセスにルールが無い。

⇨ 共有資源については、１つのプロセスが使い始めた時点で、(その使用が終るまで)他のプロセスからのアクセスを禁止する必要がある。

預金の引出し/預け入れに関しては、

- 読み出しだけなら、矛盾は起きない。
- 共有データの更新を行うプロセスが１つでもあれば、その処理の間、その共有データへのアクセスは抑止されなければならない。



⇨ 共有資源については、1つのプロセスが使い始めた時点で、(その使用が終るまで)他のプロセスからのアクセスを禁止する必要がある。

クリティカル・セクション:

… プログラムの中で、途中に別のプロセスの実行が割り込んではいけない(i.e.共有データに矛盾が生じる可能性がある)区間

クリティカル・リージョン, 危険部分とも言う。

12–3 排他制御のために何をすれば良いか—基本

クリティカル・セクションの最中にプロセス切替えを起こさない。

12–3–1 割込み禁止による排他制御

考え方：
クリティカル・セクションの間は割込み禁止にする。そのために、割込み禁止の操作disable()と割込みを可能にする操作enable()を用いて全てのクリティカル・セクションを次のように書き換える。

```
disable();    /* 割込み禁止にする */
┌──────────────────────┐
│  クリティカル・セクション  │
└──────────────────────┘
enable();     /* 割込み可能にする */
```

⇨ タイマ割込み、入出力割込みを含む全ての割込みが抑止される。
⇨ クリティカル・セクションの間中、資源を独占できそう。

—

欠点：

- 割込み禁止の権限を一般ユーザに与えるのは危険。
  (システム全体の信頼性と性能の低下に繋がる。)

例えば、
(意図的に、もしくは誤って) クリティカル・セクション修了後に enable() が実行されないと、コンピュータは割込みを受け付けないモードのままで、

◇ 端末からの入力が行えなくなる。

◇ タイマ割り込みで起動される全ての処理が実行不可能になる。

- マルチプロセッシングに対応できない。

なぜなら、
マルチプロセッシング環境では割込み禁止の設定はCPU毎に行われる。

⇨ クリティカル・セクションの間に他のプロセッサ上の別のプロセスが共有資源にアクセスする可能性が残る。

12–3–2 鍵を掛けることによる排他制御

考え方：

クリティカル・セクションの間は資源に鍵を掛けて、他のプロセスが資源を使えなくする。そのために、資源が空くまで待って空いたらその鍵(i.e. 使用権)を確保して鍵を掛ける操作lock()と、鍵を解放する操作unlock()を用いて全てのクリティカル・セクションを次のように書き換える。

```
lock();       /*資源が空くまで待って空いたら*/
              /*その鍵を確保して鍵を掛ける   */
```

クリティカル・セクション

```
unlock();     /* 鍵を解放する */
```

補足：

lock()とunlock()は重要な操作であるので、システムコールとしてOSがその処理を行うのが一般的である。

ビジーウェイト法による lock( ) と unlock( ) の実装 (第一次案) :

資源が使用中かどうかを表す変数 LOCK を用いれば、関数 lock( ) と unlock( ) は次の様に実装出来るのではないかと考えられる。

```
/* 大域変数 LOCK の初期設定 */
int LOCK = 0;       /* LOCK==1 <==> 「使用中」 */

/* 関数 lock( ) の定義 */
void lock( )
{
  while (LOCK != 0)     /* 資源が空くまで待つ */
    ;
  LOCK = 1;             /* 空いたらすぐに鍵 (i.e. 使用権) を確保 */
}

/* 関数 unlock( ) の定義 */
void unlock( )
{
  LOCK = 0;
}
```

問題点：

- ビジーウェイトの間はCPUの浪費となる。
  ⇨ ビジーウェイトのプロセスを待ち状態にする機構が必要。

- マルチプロセシング環境においては、複数のプロセスが大域変数LOCKに同時にアクセスして同じ資源の鍵を各々が手にするということが起こり得る。
  ⇨ 大域変数LOCKへのアクセスも排他制御しないといけない。
  ⇨ ハードウェアの整備。(p.167 Test-and-set命令等)

- 最終的にはlock()も機械語で実行される。 LOCKは大域共有変数なので、関数lock()の機械語コードの中にクリティカルセクションが出来るのでは？ ⟹次の段落
  ⇨ CPUが1台の場合もこれではうまく排他制御できない。
  ⇨ ハードウェアの整備。(p.167 Test-and-set命令等)

機械語レベルでのlock()の処理：

LOCKは大域共有変数なので、関数lock()を機械語風に書き下ろしてみると、例えば次の様になる。

```
Obt     Load        R1,LOCK     # while (LOCK != 0)
        Comp        R1,ZERO     #    ;
        BNonZero    Obt         #
        Load        R1,ONE        # LOCK = 1;
        Store       R1,LOCK       #
```

しかし、LOCKは大域共有変数なので、これだとこの本体全体がクリティカルセクションになってしまう。

⇨ 関数lock()をクリティカルセクション無しの機械語コードで表すことが可能になる様な、機械語命令がほしい。例えば、...

- 共有変数の値のチェックと(その結果に応じた)更新を単一ステップ内で実行。
- 実行中は他プロセッサからの共有変数へのアクセスを抑止する。

Test-and-set 命令：

lock() 手続きをクリティカルセクション無しの機械語コードで表せる様にするために、次の様な機械語命令がハードウェアから提供されていることが多い。

- 共有変数 (.e.g.LOCK) の変数値のチェックと (その結果に応じた) 更新を単一ステップ内で実行。

- マルチプロセシング環境にも対処するため、その機械語命令を実行する間は、他のプロセッサから共有変数へのアクセスを抑止する。

例えば、IBM System/370においては 次の様な命令が用意されていた。

- TS(Test and Set)命令 ···
  指定したオペランドの先頭の1バイトを共有変数と見て、共有変数に対する先頭ビットのチェックと値 X"FF"の代入をatomic operationとして(i.e. 途中で割り込まれることなく)実行する。
- CS(Compare and Swap)命令 ···
  指定した4バイトの語領域を共有変数と見て、共有変数のチェックと(その結果に応じた)更新をatomic operationとして実行する。
- CDS(Compare Double and Swap)命令 ···
  指定した8バイトの倍長語領域を共有変数と見て、共有変数のチェックと(その結果に応じた)更新をatomic operationとして実行する。

このうち、CS命令は 次の様に動作する。

(命令形式)　CS R1,R3,D2(B2)

R1 ··· 第1オペランドとなる、汎用レジスタの番号。
R3 ··· 第3オペランドとなる、汎用レジスタの番号。
D2(B2) ··· 第2オペランドとなる変数領域の番地を指定した部分である。B2は変数領域の近くを指す汎用レジスタ(ベースレジスタと言う)の番号、D2はB2から変数領域までの変位を表す。

(処理内容)　共有変数

```
if (レジスタR1の内容 == D2(B2)の指す変数領域の内容) {
  D2(B2)の指す変数領域 ⟵ レジスタR3の内容
}else {
  レジスタR1 ⟵ D2(B2)の指す変数領域の内容
}
```

(副作用)　式　(レジスタR1の内容 − D2(B2)の指す変数領域の内容)の計算をした時と同等のコンディションコードがセットされる。

補足：
CS命令,TS命令,...等は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。

CS命令を用いたlock()の実装：

IBM System/370においては、CS命令を用いることによって関数lock()を次の様な機械語コードで実装することが出来る。

```
         L     R3,MyCPUid    #Load レジスタR3<--MyCPUid
LOOP     L     R1,LOCK       ビジーウェイト
         LTR   R1,R1         |#Load and Test Register
         BNZ   LOOP          |#Branch on Non-Zero
         CS    R1,R3,LOCK    #Compare and Swap
         BNZ   LOOP
         Return              #使用権獲得==>lock()を出る
MyCPUid  DC    F(04)         #Define Constant(領域確保)
LOCK     DS    F             #Define Storage(領域確保)
```

- **2~ 4行目**のループはビジーウェイトで待っている部分である。

  システム全体の性能低下を避けるため、
  CS命令を極力実行しないで済むように工夫している。

- CS命令を用いることにより、lock()の機械語コード上からうまくクリティカルセクションを無くすことが出来ている。

- CS命令を実行する間は共有変数へのアクセスが抑止されるので、このlock()の実装はマルチプロセシング環境においてもうまく働く。

LTR命令についての補足：

(命令形式) LTR R1,R2

(処理内容) 第2オペランドがR1で指定された汎用レジスタにロードされる。この時、オペランドの符号および大きさによりコンディションコードがセットされる。第2オペランドは変わらない。

(副作用) コンディションコードが次の様にセットされる。

0 ··· **R1**にロードされた値 $=$ **0** の時.
1 ··· **R1**にロードされた値 $<$ **0** の時.
2 ··· **R1**にロードされた値 $>$ **0** の時.
3 ··· (不使用)

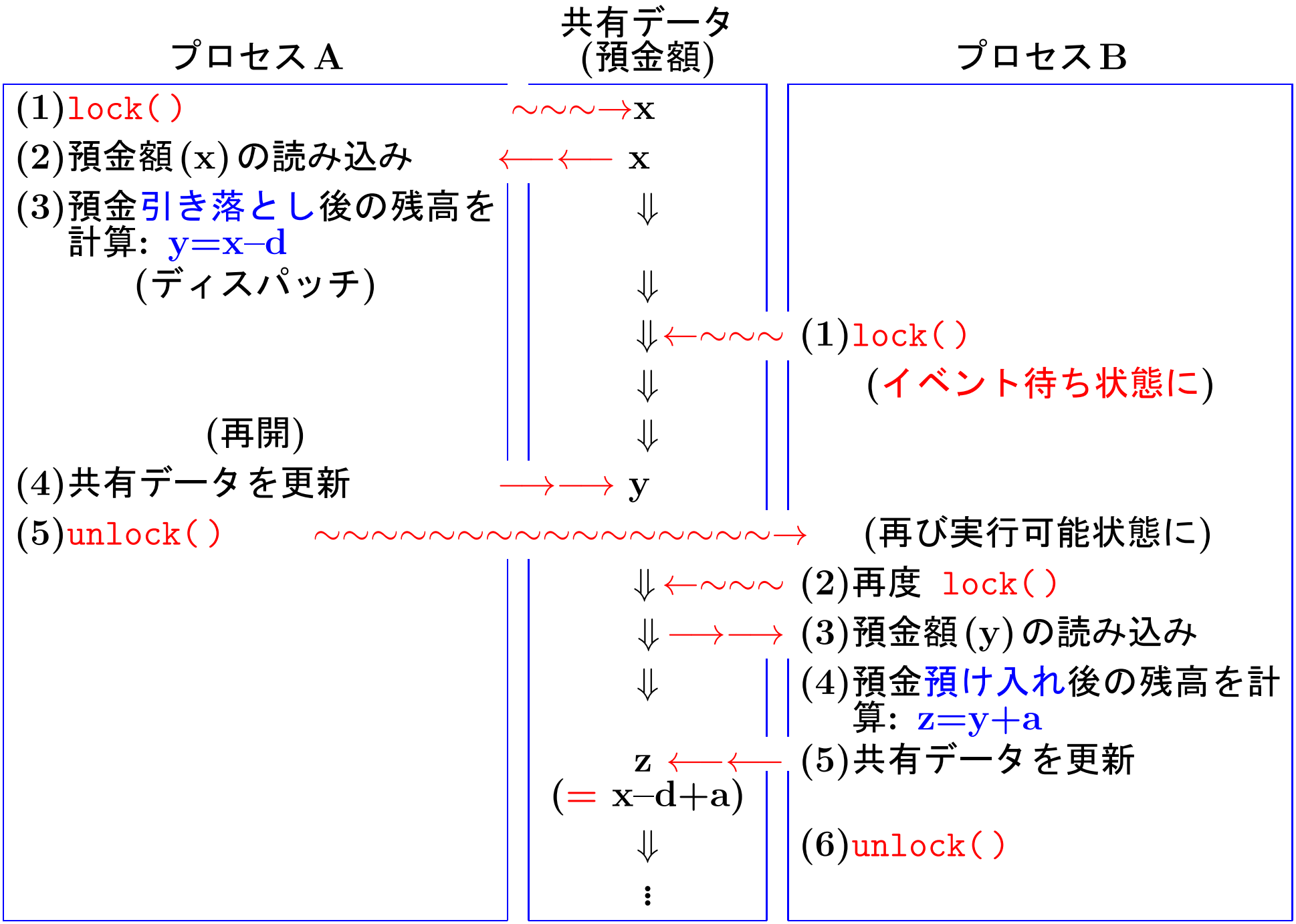
12–4 排他制御下でのシステムの効率的な運用

ビジーウェイトはCPUの浪費である。

⇨ lock( ) の中で確保したい資源が「使用中」なら、 一般的には、ビジーウェイトをするのではなくイベント待ち状態に入った (i.e.CPU を他のプロセスのために明け渡した) 方がシステム全体の生産性が上がる。

(但し、 待ちが短時間で終わることが予め分かっている場合は、 ディスパッチを伴う方法よりビジーウェイトの方が効率的で用いられることも ある。)

例12. 2 (排他制御の効率化) 例12.1において、確保したい資源が「使用中」の時イベント待ち状態に移る様にすると、次の様にデータ処理が進む。


プロセスA
共有データ
(預金額)
プロセスB
(1)lock()
x
(2)預金額(x)の読み込み
x
(3)預金引き落とし後の残高を
計算: y=x−d
(ディスパッチ)
(1)lock()
(イベント待ち状態に)
(再開)
(4)共有データを更新
y
(5)unlock()
(再び実行可能状態に)
(2)再度 lock()
(3)預金額(y)の読み込み
(4)預金預け入れ後の残高を計
算: z=y+a
z
(= x−d+a)
(5)共有データを更新
(6)unlock()

処理効率も考慮に入れたlock()とunlock()の実装：

共有資源が使用中であることが判明したら即座にイベント待ち状態に入り共有資源が空くのを待つ様にlock()とunlock()を書き換えると次の様になる。

```
/* 大域変数 LOCK の初期設定 */
int LOCK = 0;       /* LOCK==1 <==> 「使用中」 */

/* 関数 lock( ) の定義 */
void lock( )
{
  if (LOCK != 0) {      /* 資源が使用中なら即 */
    make_block( );      /* イベント待ち状態へ */
  }else if (compare_and_swap(LOCK) != 0) {
                        /* atomic operationで資源確保に失敗すると */
    make_block( );      /* イベント待ち状態へ */
  }else {
    return;             /* 資源確保に成功 */
  }
}
```

```
/* 関数 unlock( ) の定義 */
void unlock( )
{
  LOCK = 0;
  make_blocked_processes_ready( );
                /* 資源が空くのを待っているプロセスを     */
}               /* 実行可能状態に戻す*/
```

補足：

高級言語風に処理の中身を書いたが、lock( ), unlock( ) 等はOSがシステムコールとして提供するものである。

⇨ • 実際にはlock( )やunlock( )はCS命令,TS命令,...等を用いて機械語で実装される。

• 排他制御のためにOSがどういうことを行っているかは、lock( )やunlock( )の処理の中身を通じて充分理解してほしい。