オペレーティングシステム
第５章 プロセス同期

https://github.com/tctsigemura/OSTextBook

共有資源

- スレッド間の共有変数
- プロセス間の共有メモリ
- カーネル内のデータ構造
- ファイル
- 入出力装置
- その他

競合（Race Condition，Competition）

```
                                  // スレッド間の共有変数
                                  receipt DS      1     // 入金（３万円）
                                  payment DS      1     // 引き落とし（２万円）
                                  account DS      1     // 残高（１０万円）

// 入金管理スレッド                                    // 引き落とし管理スレッド

// 会社から給料（３万円）を受領し                       // カード会社から引き落としを
// receipt に金額を格納した.                          // 受信し payment に金額を格納した.

// 口座 account に足し込む                            // 口座 account から差し引く
(1)   LD      G0,account                             (a)   LD      G0,account
(2)   ADD     G0,receipt                             (b)   SUB     G0,payment
(3)   ST      G0,account                             (c)   ST      G0,account

// 次の処理に進む                                     // 次の処理に進む
```


クリティカルセクション（Critical Section）

複数のプロセス（スレッド）が同時に実行すると競合が発生する！！
例えば共有変数を変更する処理はクリティカルセクションである.
(前の例で「(1) から (3)」と「(a) から (c)」)
(クリティカルリージョン（Critical Region）とも呼ぶ)
クリティカルセクションには複数のスレッドが入ってはならない.

クリティカルセクションの競合問題を効率よく解決するには，

(1) 二つ以上のスレッドが同時に入らない.
(2) 入っているスレッドがない時は，すぐに入れる.
(3) 入るためにスレッドが永遠に待たされることがない.

相互排除（mutual exclusion）

複数のスレッドが同時にクリティカルセクションに入らない制御！！
(排他制御，相互排他も呼ぶ)

相互排除を行うプログラムの部分のことを次のように呼ぶ.

- エントリーセクション（Entry Section）
  クリティカルセクションに入る手続き
- エグジットセクション（Exit Section）
  クリティカルセクションを出る手続き

割込み禁止

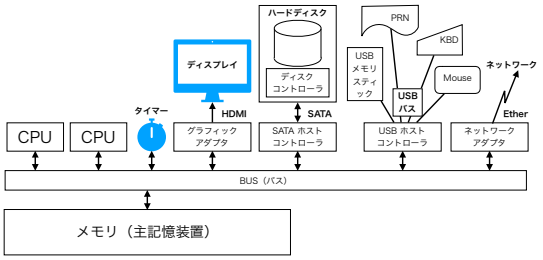
シングルプロセッサシステムで相互排除に使用できる.

```
// 口座 account に足し込む                          // 口座 account から差し引く
     DI        // Entry Section                          DI        // Entry Section
(1)  LD        G0,account                       (a)   LD        G0,account
(2)  ADD       G0,receipt                       (b)   SUB       G0,payment
(3)  ST        G0,account                       (c)   ST        G0,account
     EI        // Exit Section                           EI        // Exit Section
```

- エントリーセクション（Entry Section）
  割り込み禁止の場合は DI 命令
- エグジットセクション（Exit Section）
  割り込み禁止の場合は EI 命令

DI 命令，EI 命令は特権命令なのでカーネル内だけで可能.
長時間の割り込み禁止は NG なので注意が必要.

SMP のハードウェア構成を思い出す



- SMP では割り込み禁止だけでは相互排除できない．
- CPU はメモリ（バス）を共有する（バスの利用は順番に行う）
- 割り込みは機械語命令の途中では発生しない．

専用命令（TS 命令）

TS（Test and Set）命令は SMP システムでの相互排除に使用できる．
「TS R, M」は以下を**アトミック（atomic）**に実行する．

1. バスをロックする
2. $R \leftarrow [M]$
3. `if （R==0）` $Zero \leftarrow 1$ `else` $Zero \leftarrow 0$
4. $[M] \leftarrow 1$
5. バスのロックを解除する

専用命令（TS 命令の使用例）

```
// エントリーセクション
L1      DI              // クリティカルセクションでプリエンプションしないように
        TS      G0, FLG // ゼロを取得できるプロセス（スレッド）は一時には一つだけ
        JZ      L2      // ゼロを取得できた場合だけクリティカルセクションに入れる
        EI              // ビジーウェイティングの間はプリエンプションのチャンスを作る
        JMP     L1      // クリティカルセクションに入れない場合はビジーウェイティング

// クリティカルセクション
L2      ...

// エグジットセクション
        LD      G0, #0
        ST      G0, FLG // フラグのクリアは普通のST命令でOK
        EI              // クリティカルセクション終了，プリエンプションしても良い

// 非クリティカルセクション
        ...

// メモリ上に置いたフラグ（CPUのフラグと混同しないこと）
FLG     DC      0       // 初期値ゼロ（TS命令により 1 に書き換えられる）
```

- ビジーウェイティング（*Busy Waiting*）の一種．

専用命令（SW 命令）

SW（Swap）命令も SMP システムでの相互排除に使用できる．
「SW R, M」は以下を**アトミック（atomic）**に実行する．

1. バスをロックする
2. $T \leftarrow [M]$
3. $[M] \leftarrow R$
4. $R \leftarrow T$
5. バスのロックを解除する

ここで T は CPU 内部の一時的なレジスタ
（T レジスタの存在はプログラムから見えない）

- 次の例もビジーウェイティング（*Busy Waiting*）の一種．

専用命令（SW 命令の使用例）

```
// エントリーセクション
L1      DI              // クリティカルセクションでプリエンプションしないように
        LD      G0, #1  // フラグに書き込む値
        SW      G0, FLG // ゼロを取得できるプロセス（スレッド）は一時には一つだけ
        CMP     G0, #0  // ゼロを取得できたかテスト
        JZ      L2      // ゼロを取得できた場合だけクリティカルセクションに入れる
        EI              // ビジーウェイティングの間はプリエンプションのチャンスを作る
        JMP     L1      // クリティカルセクションに入れない場合はビジーウェイティング

// クリティカルセクション
L2      ...

// エグジットセクション
        LD      G0, #0
        ST      G0, FLG // フラグのクリアは普通のST命令でOK
        EI              // クリティカルセクション終了，プリエンプションしても良い

// 非クリティカルセクション
        ...

// メモリ上に置いたフラグ（CPUのフラグと混同しないこと）
FLG     DC      0       // 初期値ゼロ（SW命令により 1 に書き換えられる）
```


専用命令（CAS 命令）

CAS（Compare And Swap）命令も SMP システムでの相互排除に使用できる．「CAS R0, R1, M」は，以下を**アトミック（atomic）**に実行する．

1. バスをロックする
2. $T \leftarrow [M]$
3. `if` （$T == R0$）{ $[M] \leftarrow R1$; $Zero \leftarrow 1$; }
   `else` { $R0 \leftarrow T$; $Zero \leftarrow 0$; }
4. バスのロックを解除する

```
// 口座 account に足し込む                  // 口座 account から差し引く
L1      LD      G0,account              L2      LD      G0,account
        LD      G1,G0                           LD      G1,G0
        ADD     G1,receipt                      SUB     G1,payment
        CAS     G0,G1,account                   CAS     G0,G1,account
        JNZ     L1                              JNZ     L2
```

ロックフリー（Lock-free）なアルゴリズム

フラグを用いる方法（Peterson のアルゴリズム）

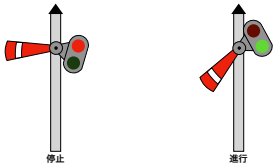
```
                // スレッド間の共有変数
                boolean flag[] = {false, false};  // クリティカルセクションに入りたい
                int turn = 0;                     // 後でやってきたのはどちら

// スレッド 0                                     // スレッド 1
...                                               ...

// エントリーセクション                           // エントリーセクション
flag[0] = true;                                     flag[1] = true;
turn = 0;                                           turn = 1;
while (turn==0 && flag[1]==true)                    while(turn==1 && flag[0]==true)
   ; // ビジーウェイティング                         ; // ビジーウェイティング

// クリティカルセクション                         // クリティカルセクション
...                                                 ...

// エグジットセクション                           // エグジットセクション
flag[0] = false;                                    flag[1] = false;

// 非クリティカルセクション                       // 非クリティカルセクション
...                                                 ...
```


セマフォ（Semaphore）

1965 年に E. W. Dijkstra が提案したデータ型である．

- **ビジーウェイティング（Busy Waiting）**を用いない
- オペレーティングシステムが提供する洗練された同期機構
- システムコール等でユーザプロセスに提供
- サブルーチンとしてサービスモジュール等に提供

セマフォ（*Semaphore*：腕木式信号機）の元々の意味はこれ！



セマフォ（Semaphore）

- セマフォはデータ構造（セマフォ型，セマフォ構造体）
  例えば C 言語で：「`typedef struct { ... } Semaphore;`」
- カウンタは 0 以上の整数値（0 は赤信号の意味）
- プロセスの待ち行列を作ることができる．
- セマフォ型の変数に *P* 操作と *V* 操作ができる．
- *P* 操作（*Proberen:try*）
- *V* 操作（*Verhogen:raise*）
- ユーザプロセスには *P*，*V* システムコールが提供される
- サービスモジュールやデバイスドライバには *P*，*V* サブルーチン

セマフォはプロセス（スレッド）の状態を待ち *(Waiting)* 状態に変える．
ビジーウェイティング（*Busy Waiting*）では無いので CPU を無駄遣いすることはない．

セマフォ（Semaphore）の P 操作

P 操作（*P(S)*）

1. セマフォ（S）の値が 1 以上ならセマフォの値を 1 減らす．
2. 値が 0 ならプロセス（スレッド）を待ち（Waiting）状態にし，
3. セマフォの待ち行列に追加する．

クリティカルセクションのエントリーセクション等で使用できる．

```
void P(Semaphore S) {
  if (S > 0) {
    S--;
  } else {
    プロセスを待ち(Waiting)状態にする;
    プロセスを S の待ち行列に追加する;
  }
}
```


セマフォ（Semaphore）の V 操作

V 操作（*V(S)*）

1. 待っているプロセス（スレッド）が無い場合は，セマフォ（S）の値を 1 増やす．
2. セマフォ（S）の待ち行列にプロセス（スレッド）がある場合は，それらの一つを起床させる．

クリティカルセクションのエグジットセクション等で使用できる．

```
void V(Semaphore S) {
  if (S の待ち行列は空) {
    S++;
  } else {
    一つのプロセスを待ち行列から取り出す;
    そのプロセスを実行可能(Ready)状態にする;
  }
}
```

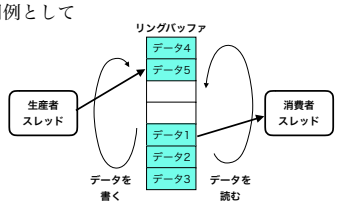

セマフォの使用例（相互排除問題）

```
int       account;                  // スレッド間の共有変数（残高）
Semaphore accSem = 1;               // 初期値1のセマフォaccSem（accountのロック用）
void receiveThread() {              // 入金管理スレッド
  for ( ; ; ) {                     // 入金管理スレッドは以下を繰り返す
    int receipt = receiveMoney();   // ネットワークから入金を受信する
    P( &accSem );                   // account変数をロックするための P 操作
    account = account + receipt;    // account変数を変更する(クリティカルセクション)
    V( &accSem );                   // account変数をロック解除するための V 操作
  }
}
void payThread() {                  // 引落し管理スレッド
  for ( ; ; ) {                     // 引落し管理スレッドは以下を繰り返す
    int payment = payMoney();       // ネットワークから支払い額を受信する
    P( &accSem );                   // account変数をロックするための P 操作
    account = account - payment;    // account変数を変更する(クリティカルセクション)
    V( &accSem );                   // account変数をロック解除するための V 操作
  }
}
```

初期値が 1 のセマフォを用いる．

生産者・消費者問題

セマフォの使用例として



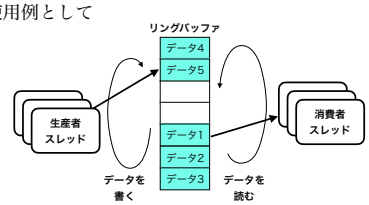
- 生産者スレッドはデータをバッファに書き込む．
- 消費者スレッドはデータをバッファから読む．
- バッファが溢れないように両者で歩調を合わせる必要がある．

セマフォの使用例（生産者・消費者問題）

```
Data      buffer[N];                  // スレッド間で共有するリングバッファ
Semaphore emptySem = N;               // リングバッファの空きスロット数を表すセマフォ
Semaphore fullSem  = 0;               // リングバッファの使用中スロット数を表すセマフォ
void producerThread() {               // 生産者スレッド
  int in = 0;                         // リングバッファの次回格納位置
  for ( ; ; ) {                       // 生産者スレッドは以下を繰り返す
    Data d = produce();               // 新しいデータを作る
    P( &emptySem );                   // リングバッファの空き数をデクリメント
    buffer[ in ] = d;                 // リングバッファにデータを格納
    in = (in + 1) % N;                // 次回格納位置を更新
    V( &fullSem );                    // リングバッファのデータ数をインクリメント
  }
}
void consumerThread() {               // 消費者スレッド
  int out = 0;                        // リングバッファの次回取り出し位置
  for ( ; ; ) {                       // 消費者スレッドは以下を繰り返す
    P( &fullSem );                    // リングバッファのデータ数をデクリメント
    Data d = buffer[ out ];           // リングバッファからデータを取り出す
    out = (out + 1) % N;              // 次回取出し位置を更新
    V( &emptySem );                   // リングバッファの空き数をインクリメント
    consume( d );                     // データを使用する
  }
}
```


複数生産者・複数消費者問題

セマフォの使用例として



- 生産者スレッド同士で相互排除が必要．
- 消費者スレッド同士でも相互排除が必要．

セマフォの使用例（複数生産者・複数消費者問題 1/2）

```
Data      buffer[N];                  // スレッド間で共有するリングバッファ
Semaphore emptySem = N;               // リングバッファの空きスロット数を表すセマフォ
Semaphore fullSem  = 0;               // リングバッファの使用中スロット数を表すセマフォ
int       in = 0;                     // リングバッファの次回格納位置
Semaphore inSem = 1;                  // in の排他制御用セマフォ
void producerThread() {               // 生産者スレッド（複数のスレッドで並列実行する）
  for ( ; ; ) {                       // 生産者スレッドは以下を繰り返す
    Data d = produce();               // 新しいデータを作る
    P( &emptySem );                   // リングバッファの空き数をデクリメント
    P( &inSem );                      // in にロックを掛ける
    buffer[ in ] = d;                 // リングバッファにデータを格納
    in = (in + 1) % N;                // 次回格納位置を更新
    V( &inSem );                      // in のロックを外す
    V( &fullSem );                    // リングバッファのデータ数をインクリメント
  }
}
```

- 複数の生産者スレッドが producerThread() を実行する．
- in 変数は生産者スレッド間の共有変数になった．
- in 変数の使用で相互排除するために inSem を準備した．

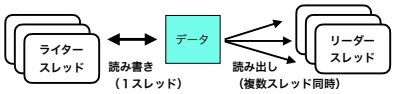
セマフォの使用例（複数生産者・複数消費者問題 2/2）

```
int out = 0;                          // リングバッファの次回取り出し位置
Semaphore outSem = 1;                 // out の排他制御用セマフォ
void consumerThread() {               // 消費者スレッド（複数のスレッドで並列実行する）
  for ( ; ; ) {                       // 消費者スレッドは以下を繰り返す
    P( &fullSem );                    // リングバッファのデータ数をデクリメント
    P( &outSem );                     // out にロックを掛ける
    Data d = buffer[ out ];           // リングバッファからデータを取り出す
    out = (out + 1) % N;              // 次回取出し位置を更新
    V( &outSem );                     // out のロックを外す
    V( &emptySem );                   // リングバッファの空き数をインクリメント
    consume( d );                     // データを使用する
  }
}
```

- 複数の消費者スレッドが consumerThread() を実行する．
- out 変数は消費者スレッド間の共有変数になった．
- out 変数の使用で相互排除するために outSem を準備した．

リーダ・ライタ問題

次の場合，単にロックするより並行性（*concurrency*）を高くできる．



- ライタスレッドはデータ読んでを変更して書き込む．
- データ変更中は他のスレッドはデータにアクセスしてはならない．
  （排他ロック（*exclusive lock*））

- リーダスレッドはデータを変更しない．
- 複数のリーダスレッドが同時にデータにアクセスしても良い．
  しかし，その間，ライタスレッドはデータにアクセスできない．
  （共有ロック（*shared lock*））

セマフォの使用例（リーダ・ライタ問題 1/2）

データとライタスレッド部分

```
Data      records;                  // 共有するデータ
Semaphore rwSem = 1;                // リーダとライタの排他用セマフォ
void writerThread() {               // ライタスレッド（複数のスレッドで並列実行する）
  for ( ; ; ) {                     // ライタスレッドは以下を繰り返す
    Data d = produce();             // 新しいデータを作る
    P( &rwSem );                    // 共有データにロックを掛ける
    writeRecores( d );              // データを書換える
    V( &rwSem );                    // 共有データのロックを外す
  }
}
```

- 複数のライタスレッドが `writerThread()` を実行する．
- `rwSem=1` は，普通の相互排除と同じ．
- ライタスレッドは，排他ロック（*exclusive lock*）を用いる．

次ページのスライドがリーダスレッド部分

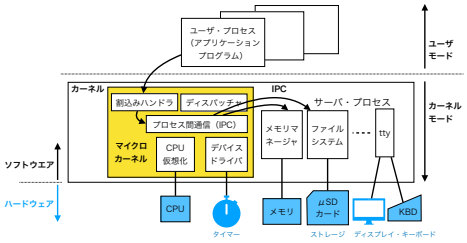
セマフォの使用例（リーダ・ライタ問題 2/2）

```
int       cnt = 0;                  // リーダ間の共有変数（読出し中のリーダ数）
Semaphore cntSem = 1;               // cnt の排他制御用セマフォ
void readerThread() {               // リータスレッド（複数のスレッドで並列実行する）
  for ( ; ; ) {                     // リーダスレッドは以下を繰り返す
    P( &cntSem );                   // cnt にロックを掛ける
    if ( cnt == 0 ) P( &rwSem );    // 自分が最初のリーダなら，代表してロックする
    cnt = cnt + 1;                  // cnt をインクリメント
    V( &cntSem );                   // cnt のロックを外す
    Data d = readRecords();         // データを読みだす
    P( &cntSem );                   // cnt にロックを掛ける
    cnt = cnt - 1;                  // cnt をデクリメント
    if ( cnt == 0 ) V( &rwSem );    // 自分が最後のリーダなら，代表してロックを外す
    V( &cntSem );                   // cnt のロックを外す
    consume( d );                   // データを使用する
  }
}
```

- 複数のリーダスレッドが `readerThread()` を実行する．
- `cnt` は，クリティカルセクション中のリーダスレッド数．
- `cntSem=1` は `cnt` の相互排除に用いる．
- リーダスレッドは，共有ロック（*shared lock*）を用いる．

実装例

第２０章
TacOSのセマフォ

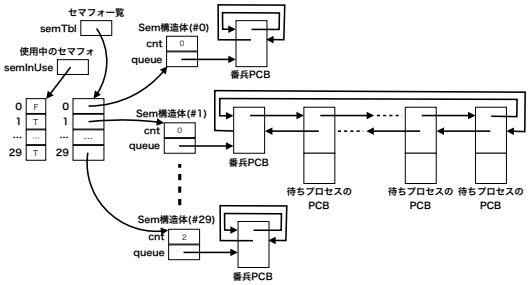


TacOS のセマフォ構造体（カーネル内）

```
#define SEM_MAX 30          // セマフォは最大 30 個

struct Sem {                // セマフォを表す構造体
  int cnt;                  // カウンタ
  PCB queue;                // 待ち行列
};
```

- セマフォは最大 30 個（TaC のメモリは小さい）
- セマフォ構造体の名前は `Sem`
- `cnt` がセマフォの値（0 以上）
- `queue` に，このセマフォを待っているプロセスの待ち行列を作る．

TacOS のセマフォ関連データ構造（カーネル内）



- TacOS では，セマフォを `semTbl` のインデクスで識別する．
- Sem 構造体（#0，#1，#29）は，未使用，待ちあり，待ちなしの例

TacOS でのセマフォの架空の使用例

```
1  #incude <kernel.h>
2  int       account;              // スレッド間の共有変数（残高）
3  int       accSem;               // accountのロック用セマフォの番号
4  void initProc() {               // プロセスの初期化ルーチン
5    accSem = newSem(1);           // 初期値1のセマフォを確保する
6  }
7  void receiveThread() {          // 入金管理スレッド
8    for ( ; ; ) {                 // 入金管理スレッドは以下を繰り返す
9      int receipt = receiveMoney(); // ネットワークから入金を受信する
10     semP( accSem );             // account変数をロックするための P 操作
11     account = account + receipt; // account変数を変更する(クリティカルセクション
12     semV( accSem );             // account変数をロック解除するための V 操作
13   }
14 }
15 void payThread() {              // 引落し管理スレッド
16   for ( ; ; ) {                 // 引落し管理スレッドは以下を繰り返す
17     int payment = payMoney();   // ネットワークから入金を受信する
18     semP( accSem );             // account変数をロックするための P 操作
19     account = account - payment; // account変数を変更する(クリティカルセクション
20     semV( accSem );             // account変数をロック解除するための V 操作
21   }
22 }
```

TacOS のセマフォ割当て解放ルーチン（カーネル内）

```
1  Sem[] semTbl=array(SEM_MAX);                    // セマフォの一覧表
2  boolean[] semInUse=array(SEM_MAX);              // どれが使用中か（falseで初期
3
4  // セマフォの割当て
5  public int newSem(int init) {
6    int r = setPri(DI|KERN);                      // 割り込み禁止、カーネル
7    for (int i=0; i<SEM_MAX; i=i+1) {             // 全てのセマフォについて
8      if (!semInUse[i]) {                         //   未使用のものを見つけたら
9        semInUse[i] = true;                       //     使用中に変更し
10       semTbl[i].cnt = init;                     //     カウンタを初期化し
11       setPri(r);                                //     割込み状態を復元し
12       return i;                                 //     セマフォ番号を返す
13     }
14   }
15   panic("newSem");                              // 未使用が見つからなかった
16   return -1;                                    // ここは実行されない
17 }
18
19 // セマフォの解放
20 // (書き込み1回で仕事が終わるので割込み許可でも大丈夫)
21 public void freeSem(int s) {
22   semInUse[s] = false;                          // 未使用に変更
23 }
```


TacOS の P 操作ルーチン（カーネル内）

```
1  public void semP(int sd) {
2    int r = setPri(DI|KERN);                      // 割り込み禁止、カーネル
3    if (sd<0 || SEM_MAX<=sd || !semInUse[sd])     // 不正なセマフォ番号
4      panic("semP(%d)", sd);
5
6    Sem s = semTbl[sd];
7    if(s.cnt>0) {                                 // カウンタから引けるなら
8      s.cnt = s.cnt - 1;                          //   カウンタから引く
9    } else {                                      // カウンタから引けないなら
10     delProc(curProc);                           //   実行可能列から外し
11     curProc.stat = P_WAIT;                      //   待ち状態に変更する
12     insProc(s.queue,curProc);                   //   セマフォの行列に登録
13     yield();                                    //   CPUを解放し
14   }                                             //     他プロセスに切換える
15   setPri(r);                                    // 割り込み状態を復元する
16 }
```


TacOS の PCB リスト操作関数（カーネル内）

```
1  // プロセスキューでp1の前にp2を挿入する  p2 -> p1
2  void insProc(PCB p1, PCB p2) {
3    p2.next=p1;
4    p2.prev=p1.prev;
5    p1.prev=p2;
6    p2.prev.next=p2;
7  }
8
9  // プロセスキュー(実行可能列やセマフォの待ち行列)で p を削除する
10 void delProc(PCB p) {
11   p.prev.next=p.next;
12   p.next.prev=p.prev;
13 }
```


TacOS の V 操作ルーチン（1/2）（カーネル内）

```
1  // ディスパッチを発生しないセマフォのV操作
2  // (V 操作をしたあとまだ仕事があるとき使用する)
3  // (kernel 内部専用、割込み禁止で呼出す)
4  boolean iSemV(int sd) {
5    if (sd<0 || SEM_MAX<=sd || !semInUse[sd]) {   // 不正なセマフォ番号
6      panic("iSemV(%d)", sd);
7    }
8    boolean ret = false;                          // 起床するプロセスなし
9    Sem s = semTbl[sd];                           // 操作するセマフォ
10   PCB q = s.queue;                              // 待ち行列の番兵
11   PCB p = q.next;                               // 待ち行列の先頭プロセス
12   if(p==q) {                                    // 待ちプロセスが無いなら
13     s.cnt = s.cnt + 1;                          //   カウンタを足す
14   } else {                                      // 待ちプロセスがあるなら
15     delProc(p);                                 //   待ち行列から外す
16     p.stat = P_RUN;                             //   実行可能に変更
17     schProc(p);                                 //   実行可能列に登録
18     ret = true;                                 //   起床するプロセスあり
19   }
20   return ret;                                   // 実行可能列に変化があった
21 }
```

- iSemV() は割込禁止で呼び出す.

TacOS の V 操作ルーチン（2/2）（カーネル内）

```
23 // セマフォの V 操作
24 //   待ちプロセス無し : カウンタを1増やす
25 //   待ちプロセス有り : 待ち行列からプロセスを1つ外して実行可能にした後、
26 //                     ディスパッチャを呼び出す
27 public void semV(int sd) {
28   int r = setPri(DI|KERN);                      // 割り込み禁止、カーネル
29   if (iSemV(sd)) {                              // V 操作し必要なら
30     yield();                                    //   プロセスを切り替える
31   }
32   setPri(r);                                    // 割り込み状態を復元する
33 }
```

- iSemV() を呼び出す前に割込禁止にする.
- iSemV() が true で返ったらプロセスの切換えを試みる.
- yield() でプリエンプションしたプロセスは，yield() から実行が再開される.

TacOS の CPU フラグ操作関数（カーネル内）

```
1  ;; CPU のフラグの値を返すと同時に新しい値に変更
2  _setPri
3          ld      g0,2,sp     ; 引数の値を G0 に取り出す
4          push    g0          ; 新しい状態をスタックに積む
5          ld      g0,flag     ; 古いフラグの値を返す準備をする
6          reti                ; reti は FLAG と PC を同時に pop する
```

- CPU の PSW のフラグに割込禁止ビットがある.
- C--言語から setPri() 関数として呼び出せるようにするには，アセンブリ言語プログラムで_setPri ラベルを宣言する必要がある.
- C--言語プログラムは引数をスタックに積んで関数を CALL する.
- アセンブリ言語プログラムで引数を参照するには，(SP 相対で）スタックから取り出す．(SP+0 番地が PC，SP+2 番地が第 1 引数)
- 関数の返り値は，G0 レジスタに入れて返す.
- reti 命令はスタックからフラグと PC を一度に取り出す.

練習問題

練習問題

練習問題（1）

- 次の言葉の意味を説明しなさい．
  - 競合
  - クリティカルセクション
  - 相互排除
  - ビジーウェイティング
  - ロックフリーなアルゴリズム
  - セマフォ
  - 相互排除問題
  - 生産者と消費者問題
  - リーダライタ問題

練習問題（2）

- なぜ割込みを禁止することで相互排除ができるか？
- 割込み禁止による相互排除がマルチプロセッサシステムでは不十分な理由は？
- 割込み禁止による相互排除はクリティカルセクションの三つの条件を満たしているか？
- CPU が割込み禁止になっている間に発生した割込みはどのように扱われるか？
- DI 命令や EI 命令が特権命令でなかったら，どのような不都合が生じるか？
- シングルプロセッサシステムにおいて，機械語命令はアトミック（*atomic*）と言えるか？
- マルチプロセッサシステムにおいて，機械語命令はアトミック（*atomic*）と言えるか？

練習問題（3）

- TS 命令と SW 命令に共通な特長は何か？
- TS 命令を用いたビジーウェイティングはシングルプロセッサシステムでも使用できるか？
- セマフォを相互排除に使用する手順を説明しなさい．
- 生産者と消費者の問題において，二つのセマフォはどのような値に初期化されたか？
  二つのセマフォは何の役割を持っていたか？