オペレーティングシステム
第５章 プロセス同期

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）

同時に複数のプロセス（スレッド）が実行すると競合が発生する部分（クリティカルリージョン（Critical Region）とも呼ぶ）

1. 二つ以上のプロセス（スレッド）が同時にクリティカルセクションに入らない.
2. クリティカルセクションに入っているプロセス（スレッド）がない時は，待たされることなくクリティカルセクションに入ることができる.
3. クリティカルセクションに入るために永遠に待たされることがない.

相互排他（mutual exclusion）

- 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

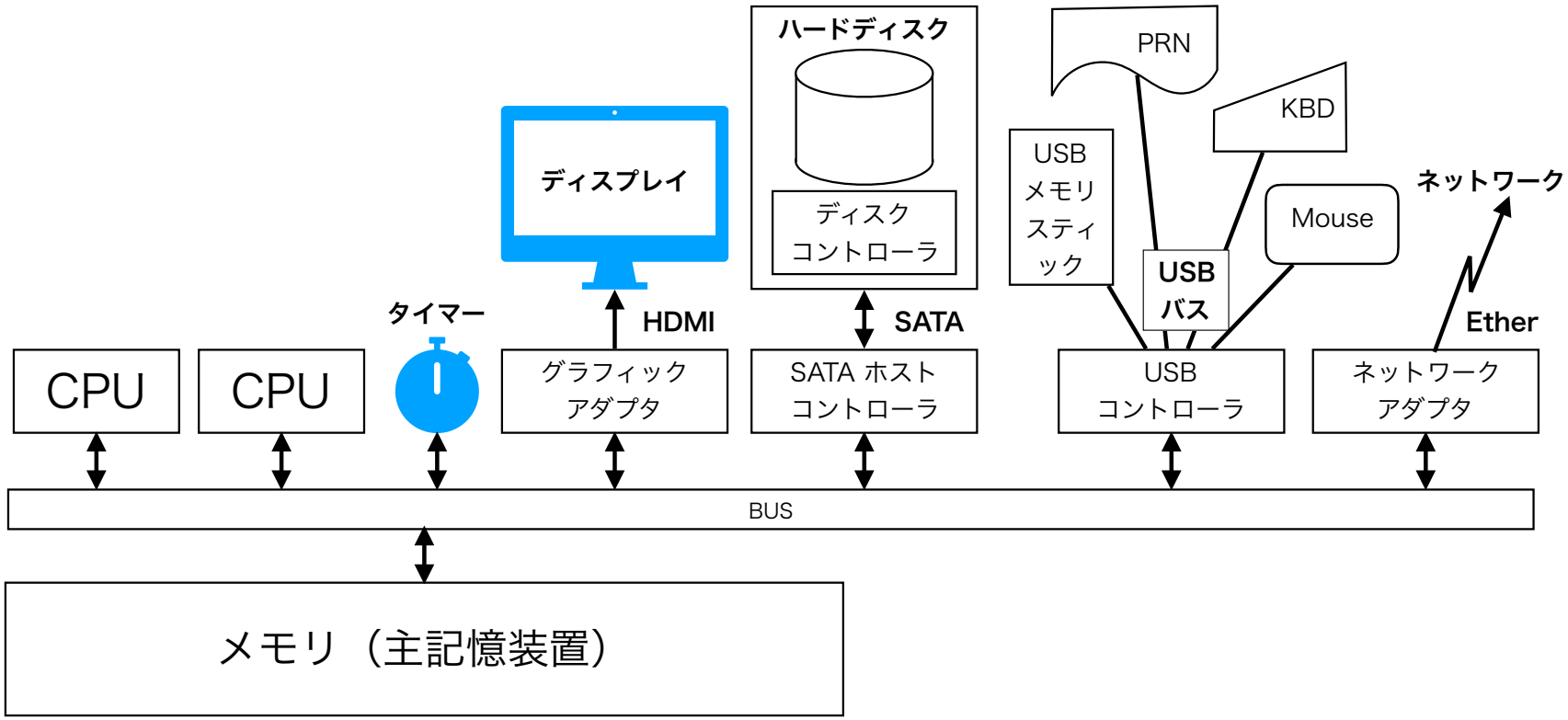
割込み禁止

```
// 口座 account に足し込む
      DI          // Entry Section
(1)   LD          GR0,account
(2)   ADD         GR0,receipt
(3)   ST          GR0,account
      EI          // Exit Section
```

```
// 口座 account から差し引く
      DI          // Entry Section
 (a)  LD          GR0,account
 (b)  SUB         GR0,payment
 (c)  ST          GR0,account
      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

ハードウェア構成



- SMP（Symmetric Multiprocessing）
- CPU はメモリを共有する

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)` $z \leftarrow 1$ `else` $z \leftarrow 0$
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
        ...

// フラグ
FLG     DC      0       // 初期値ゼロ(TS 命令により 1 に書き換えられる)
```

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）

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
        ...

// フラグ
FLG     DC      0       // 初期値ゼロ(TS 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（$T == R0$）{ $[M] \leftarrow R1$; $z \leftarrow 1$; } else { $R0 \leftarrow T$; $z \leftarrow 0$; }
4. バスのロックを解除する

```
// 口座 account に足し込む
        LD      G0,account
L1      LD      G1,G0
        ADD     G1,receipt
        CAS     G0,G1,account
        JNZ     L1
```

```
// 口座 account から差し引く
        LD      G0,account
 L2     LD      G1,G0
        SUB     G1,payment
        CAS     G0,G1,account
        JNZ     L2
```

ロックフリー（Lock-free）なアルゴリズム

Peterson のアルゴリズム

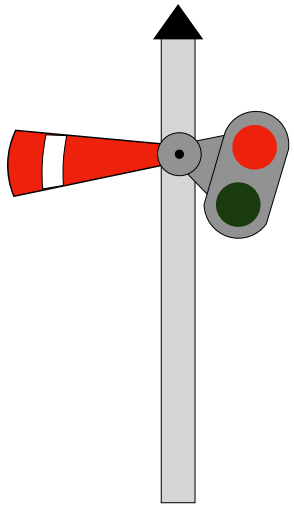
```
                    // スレッド間の共有変数
                    boolean flag[] = {false, false};  // クリティカルセクションに入りたい
                    int turn = 0;                     // 後でやってきたのはどちら

// 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] = flase;

// 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）

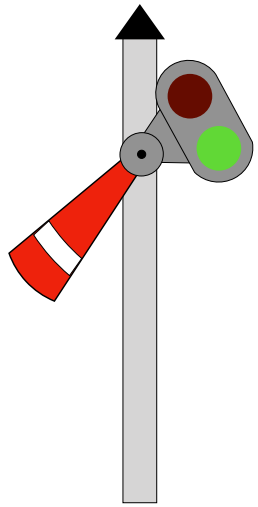
1965 年に E. W. Dijkstra が提案したデータ型である．

- **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！



停止



進行

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）
- 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**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 - 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.

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
  }
}
```

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
  }
}
```

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
  }
}
```

セマフォの使用例（リーダ・ライタ問題 1/2）

```
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
  }
}
```

排他ロック（exclusive lock）

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
  }
}
```

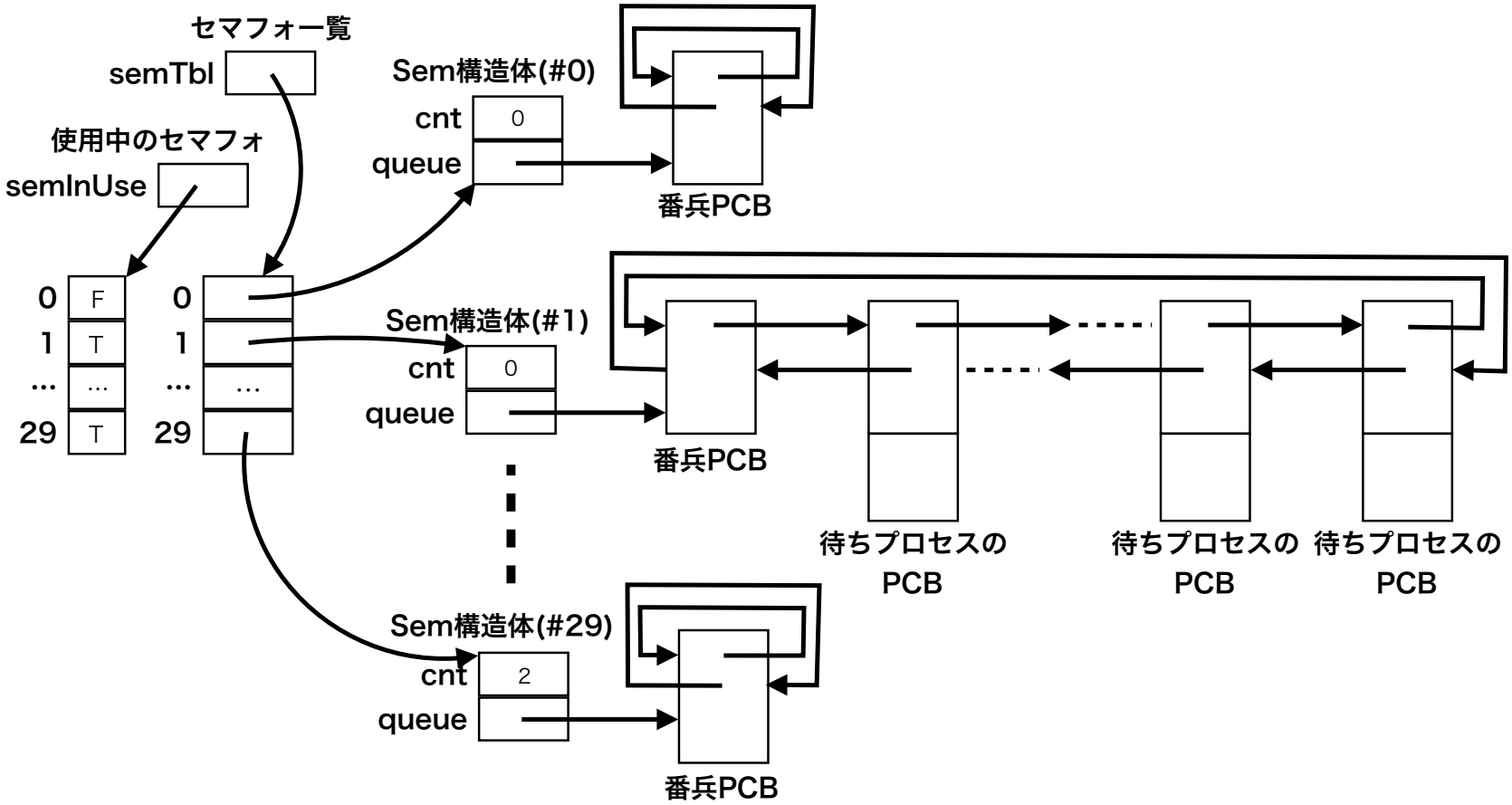
共有ロック（shared lock）

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

```
#incude <kernel.h>
int       account;                    // スレッド間の共有変数(残高)
int       accSem;                     // account のロック用セマフォの番号
void initProc() {                     // プロセスの初期化ルーチン
  accSem = newSem(1);                 // 初期値 1 のセマフォを確保する
}
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
    semP( 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)
    semV( 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();         // ネットワークから入金を受信する
    semP( 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)
    semV( accSem );                   // account 変数をロック解除するための V 操作
  }
}
```

TacOS のセマフォ割当て解放ルーチン（カーネル内）

```
Sem[] semTbl=array(SEM_MAX);                    // セマフォの一覧表
boolean[] semInUse=array(SEM_MAX);              // どれが使用中か(false で初期化)

// セマフォの割当て
public int newSem(int init) {
  int r = setPri(DI|KERN);                      // 割り込み禁止、カーネル
  for (int i=0; i<SEM_MAX; i=i+1) {             // 全てのセマフォについて
    if (!semInUse[i]) {                         //   未使用のものを見つけたら
      semInUse[i] = true;                       //     使用中に変更し
      semTbl[i].cnt = init;                     //     カウンタを初期化し
      setPri(r);                                //     割込み状態を復元し
      return i;                                 //     セマフォ番号を返す
    }
  }
  panic("newSem");                              // 未使用が見つからなかった
  return -1;                                    // ここは実行されない
}

// セマフォの解放
public void freeSem(int s) {
  semInUse[s] = false;                          // 未使用に変更(アトミック)
}
```

TacOS の P 操作ルーチン（カーネル内）

```
// 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
void delProc(PCB p) {
  p.prev.next=p.next;
  p.next.prev=p.prev;
}
// セマフォの P 操作
public void semP(int sd) {
  int r = setPri(DI|KERN);                          // 割り込み禁止、カーネル
  if (sd<0 || SEM_MAX<=sd || !semInUse[sd])         // 不正なセマフォ番号
    panic("semP(%d)", sd);

  Sem s = semTbl[sd];
  if(s.cnt>0) {                                     // カウンタから引けるなら
    s.cnt = s.cnt - 1;                              //   カウンタから引く
  } else {                                          // カウンタから引けないなら
    delProc(curProc);                               //   実行可能列から外し
    curProc.stat = P_WAIT;                          //   待ち状態に変更する
    insProc(s.queue,curProc);                       //   セマフォの行列に登録
    yield();                                        //   CPU を解放し
  }                                                 //     他プロセスに切換える
  setPri(r);                                        // 割り込み状態を復元する
}
```

TacOS の V 操作ルーチン（1/2）（カーネル内）

```
// ディスパッチを発生しないセマフォの V 操作
boolean iSemV(int sd) {
  if (sd<0 || SEM_MAX<=sd || !semInUse[sd]) {        // 不正なセマフォ番号
    panic("iSemV(%d)", sd);
  }
  boolean ret = false;                               // 起床するプロセスなし
  Sem s = semTbl[sd];                                // 操作するセマフォ
  PCB q = s.queue;                                   // 待ち行列の番兵
  PCB p = q.next;                                    // 待ち行列の先頭プロセス
  if(p==q) {                                         // 待ちプロセスが無いなら
    s.cnt = s.cnt + 1;                               //   カウンタを足す
  } else {                                           // 待ちプロセスがあるなら
    delProc(p);                                      //   待ち行列から外す
    p.stat = P_RUN;                                  //   実行可能に変更
    schProc(p);                                      //   実行可能列に登録
    ret = true;                                      //  起床するプロセスあり
  }
  return ret;                                        // 実行可能列に変化があった
}
```

- 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）

```
// セマフォの V 操作
public void semV(int sd) {
  int r = setPri(DI|KERN);                    // 割り込み禁止、カーネル
  if (iSemV(sd)) {                            // V 操作し必要なら
    yield();                                  //   プロセスを切り替える
  }
  setPri(r);                                  // 割り込み状態を復元する
}
```

- 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）

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