第 5 回 並行処理とソフトウェアによる排他制御

こんにちは。今回は並行性とは切っても切れない排他制御に関して話したいと思います。
改めて説明すると、排他制御とはタスクのように並行動作する複数のプログラムが同じデータにアクセスする場合、同時にそのデータにアクセスしてはならないタイミングがあり、その競合を防ぐことです。たとえばあるタスクがデータを生産して、他のタスクがそのデータを消費する場合、データを生産している最中にそのデータを使わせない、データを消費している最中に新たなデータを生産させないための制御です。

排他制御の必要性に関しては皆さんもいろいろ聞いてきているかもしれませんが、今回は最初に現実の組込みソフトウェア、それもリアルタイム OS 自身の排他制御の必要性に関して話します。
現役の RTOS の実例を出すには差し障りも多いので、例によって昔の RTOS の例です。時期はおそらく 1985 年頃の話です。
その当時開発していた製品ソフトウェアに機能を追加すると、1 日に 1 回程度システムダウンを起こすようになってしまいました。結局原因を突き止めるのに 3 日ほど要しましたが、原因は当時使用していた RTOS がタスク切り替え時の排他制御を行っていないことに伴う不具合でした。その時に使用していた CPU は i8086 でしたが、この CPU は実行する命令のアドレスを CS（コードセグメントレジスタ）と IP（インストラクションポインタ）の 2 つの 16 ビットレジスタの組み合わせ（CS を 4 ビット左シフトしたものと IP の和です）で 20 ビットのアドレスを指定していました。つまりプログラムのアドレスは CS:IP の両者の整合性のもとで成り立っていました。しかし OS の割り込みを許可している部分で実行命令のアドレスを変更する部分に先に IP レジスタを書き換えてから、CS レジスタを書き換えている箇所があり、その 2 つの命令の間に割り込みが発生することにより、不正なアドレスのメモリを実行しようとして暴走していたのです。これは CS レジスタと IP レジスタの排他制御ができていなかったことが原因でしたが、そのためにシステムダウンまで引き起こすことになりました。結局この不具合は CS レジスタを書き換えると次の命令実行では割り込みを禁止するという i8086 の機能を利用し、レジスタを書き換える順番を CS レジスタを書き換えてから IP レジスタを書き換えるように OS の修正を行い修正することができましたが、排他制御の重要性を実感できた良い経験でした。
つまりこのケースでは CS と IP がアトミックデータ（これ以上分割できない最小単位のデータ）であるにも関わらず、その扱いが出来ていなかったということです。

もう一つ排他制御の重要性に関する組込みシステム開発時の実例をお話ししておきます。
実は基本的なことなのですが、C や C++言語などの高級言語で書いても現実の CPU が実行するのはその CPU が命令と認識できるビットのパターン（機械語）であり、高級言語の 1 命令が機械語で 1 命令になるとは限りません。というより複数命令になる場合のほうが多いです。
機械語であればほとんどの命令は、1 つの命令実行中に割り込まれないことは保障されていますが、高級言語

の 1 命令は複数機械語命令になるのですから、当然 1 つの命令の実行中に割り込まれます。なお、ここでいう割り込みとはタスクのスイッチや文字通り割り込みなど、並列処理が切り替わることをいいます。

ここでたとえば次のようなプログラムがあったとします。
タスク_A とタスク_B は実行されるタイミングは様々な場合がありますが、とにかく何らかのタイミングで両方のタスクが実行された後だとします。ほとんどの場合は flag の値は初期値 0、タスク_A がビット 0 をオン、タスク_B がビット 1 をオンにしているわけですから、この時点で flag の値は 0x3 になっているはずです。もちろん 0x3 になっている場合が多いのですが、たまに 0x1 や 0x2 になっている場合もあります。
理由は下に書いていますが、まず原因を考えてみてください。ヒントは CISC 系ではあまり発生しない（ただし後述の通り，アクセスするアドレスによっては，あるいは最適化のオプションによっては発生する場合はあります）や、RISC 系はよく発生する、です。実際に CISC で正常に動作していた類所のソフトウェアを RISC に移植したとたんに同様の現象が発生しました。

```
volatile unsigned long flag = 0x0;

タスク_A( )
{
      . . .
    flag |= 0x1; /* ビット 0 オン */
      . . .
}
タスク_B( )
{
      . . .
    flag |= 0x2; /* ビット 1 オン */
      . . .
}
```

原因は次の通りです。
下に上記 C 言語が機械語（厳密には機械語ニーモニックコード）に変換されたあとの様子を示します。機械語は CPU によって異なりますが、ここでは一例として MIPS 風のニーモニックコードを示します。
C 言語では一命令に見えた OR 演算が 3 命令になっています。
割り込みであれ、タスクであれ並列処理が切り替わる前後ではコンテキスト（ここではレジスタセットなど）が保存復元されるので、下記の処理がオーバーラップしている場合には、両方のタスクが書き戻す flag の値が異なることになり、あとで実行されたタスクの値が flag に残ることになります。

```
タスク_A
ld reg1, flag ; flag の値を reg1 にコピーする
or reg1, 0x1 ; ビット 0 をオン
st reg1, flag ; reg1 の値を flag にコピーする

タスク_B
```

```
ld reg1, flag ; flag の値を reg1 にコピーする
or reg1, 0x2 ; ビット 1 をオン
st reg1, flag ; reg1 の値を flag にコピーする
```

具体的にいうと、タスク_A が flag の値を読み取って reg1 に保存した直後にタスク_B に切り替わる場合、まずタスク_A のレジスタの値をタスク_A の TCB に保存しますが、この時は flag が 0 だったので reg1 も 0 です。
タスク_B を実行する前にタスク_B の TCB から保存してある値をレジスタに復元します。そして値が 0 の flag の値を reg1 にコピーして、OR 演算により reg1 の値を 0x2 にしてその値を flag に書き戻すので、その時点で flag の値は 0x2 になっています。
そしていずれタスク_A に処理は戻りますが、その時はまず先ほど TCB に保存したコンテキストを復元するところから処理を再開します。すると reg1 の値は再び 0 に戻り、それに対して OR を行うため regi1 の値は 0x1 になり、それを flag に書き戻すと flag の値は 0x1 になってしまい、タスク_B がセットしたビット 1 のデータは消えてしまいました。
逆にタスク_B が実行中にタスク_A に割り込まれた場合は flag の値は 0x2 になり、タスク_A が書き込んだビット 0 の値が消えてしまいます。上の命令の実行がオーバーラップしない場合は、両方のタスクがセットしたデータはどちらも残るため flag の値は正しく 3 になります。
タスクでなく割り込みの場合は、コンテキストを保存する場所が TCB でなく割り込みスタックになる以外は同様の動きになり、おなじく不具合が発生します。

ではどうすればいいでしょうか？
上記のようなデータの数が少なく、RTOS を使用しているのであれば、セマフォにより排他制御すればいいかもしれません。
次回セマフォに関して説明しますが、セマフォを使用してもスケジューラに制御が移るとは限らないので、セマフォ呼び出しによるオーバーヘッドも関数呼び出し程度で済むかもしれません。しかし上記のようなデータが多い場合はそのオーバーヘッドも無視することが出来ませんし、RTOS を使用していない場合は、そもそもセマフォを使用できません。
一種の妥協の産物として解決策の一例を次に示しますが、一言で言うと割り込み禁止/許可を使用します。

```
割り込み処理_A( )
{
    DI；
    flag |= 0x1; /* OR 処理 */
    EI；
}
割り込み処理_B( )
{
    DI;
    flag |= 0x2; /* OR 処理 */
    EI;
}
```

DI や EI を次のようにマクロで定義しておき、インラインアセンブラと組み合わせれば、メモリ上のデータとのビット演算命令を備える CPU（この場合 OR 命令は機械語でも 1 命令なので排他制御不要）でも、メモリとのビット演算命令を持たない CPU でもソースコードを統一することが出来ます。

```
#ifndef M16C // ビット演算命令なし
#define DI asm("^t fclr i")
#define EI asm("\t fset i")
#else
#define DI
#define EI
#endif
```

なお、ここで注意しなければならないことは、メモリとのビット演算命令を備える CPU であっても、コンパイラの最適化オプションやあるいは near と far があるような区別がある CPU の場合は、データのアドレスが near 領域の場合は排他制御不要でも、far 領域の場合は排他制御が必要になるなどの場合もあり、生成された機械語の確認は欠かすことはできません。

この割り込み禁止/許可を使用する方法は、割り込み禁止を使用するとシステムのリアルタイム性に影響を与えるので注意して使用しなければなりませんが、それ以外に大きな使用上の制限があります。
それはマルチコアやマルチ CPU では割り込み禁止が役に立たないということです。同じアーキテクチャの CPU で構成されるマルチ CPU システムの場合は、ハードウェアレベルで提供される古典的な TAS 命令や CAS 命令や、新しい CPU では MIPS の LL、SC 命令セット、パワーアーキテクチャ系の LWARX、STWCX 命令セットなどの TAS や CAS 相当のハードウェアレベルでマルチ CPU の排他制御を行うための処理を自由に作れるマルチ CPU サポート命令セットを使用して排他制御を行うことも可能です。
しかし組込みシステムの場合はリアルタイム性確保のために AMP（非対称型マルチプロセッサ）が採用されることが多く、上記ハードウェアレベルでのマルチ CPU サポート命令は同じアーキテクチャの CPU 間での制御が前提であるため、これもまた利用できない場合が多いという問題があります。
ただしシステムの排他制御が必要な CPU がすべて同じ仕組みを利用できる場合には、これから説明するソフトウェアによる排他制御に比べて、
・ソフトウェアが単純になる
・CPU の個数に左右されない
という利点があるためハードウェアレベルの CPU サポート命令を使用するべきです。

他の手段として、排他制御が必要な複数の CPU に渡って同時に割り込みを禁止にできるような独自デバイスを作るという解決策もありますが、結局 CPU 間のそのデバイスの割り込み禁止にするためのアクセスの排他制御をどうするかという問題をハードウェアレベルで解決しなければならないため、簡単な解決策とは言えません。

排他制御にハードウェアの助けを借りられない場合に有効なのが、スピンロックやビジーウェイトと呼ばれるソフトウェアによる排他制御です。
これは古くからの方法であり、CPU の使用効率の面で問題はありますが、マルチコア対応の Linux の排他制

御にも使われるなど、実は現在でもかなり有用な方法で、使われています。
ソフトウェアによる排他制御には複数のアルゴリズムがあり、いくつかのやり方があります。

リスト 1 が 1965 年に Dekker が発表した、ソフトウェアで 2 つの並行処理の排他制御を行うためのアルゴリズムです。

```
volatile int turn = 0; /* whose turn is it now ? */
volatile int interested[2] = {FALSE, FALSE};

/* CS に入る前に呼び出す */
enter_cs(int taskID) { /* taskID : A/B = 0/1 */
      int other;
      other = 1 - taskID;
      interested[taskID] = TRUE;
      while(interest[other]) {  /* spin lock */
            if (turn == other)  {
                  interest[taskID] = FALSE;
                  while(turn == other) ; /* spin lock */
                  interest[taskID] = TRUE;
      }
      }
}

/* CS から抜けた後に呼び出す */
leave_cs(int taskID) {
      int other;
      other = 1 - taskID;
      turn = other;
      interested[taskID] = FALSE;
}
```

リスト 1. Dekker のアルゴリズム

そしてリスト 2 が 1981 年に Peterson が発表した、より簡潔なアルゴリズムです。

```
volatile int turn = 0;  /* 現在どちらの番か？ ? */
volatile int interested[2] = {FALSE, FALSE};

/* CS に入る前に呼び出す */
enter_cs(int taskID) { /* taskID : A/B = 0/1 */
      int other;
      other = 1 - taskID;
      interested[taskID] = TRUE;
      turn = taskID;
      while (turn == taskID && interested[other] == TRUE)
                                    ;                  /* wait spin lock */
}
/* CS から抜けた後に呼び出す */
leave_cs(int taskID) {
      interested[taskID] = FALSE;
}

taskA() {
      outside_cs1();      /* CS と関係の無い処理の実行 */
      enter_cs(0);
      inside_cs1();   /* CS 内での処理を実行 */
      leave_cs(0);
      outside_cs1();  /* CS と関係の無い処理の実行 */
}

taskB() {
      outside_cs2();      /* CS と関係の無い処理の実行 */
      enter_cs(1);
      inside_cs2();   /* CS 内での処理を実行 */
      leave_cs(1);
      outside_cs2();      /* CS と関係の無い処理の実行 */

}
```

リスト 2. Peterson のアルゴリズム

どちらも特殊な命令なしに排他制御を行うためのアルゴリズムです。
リストは 2 つの並行処理の排他制御を行うためのアルゴリズムですが、3 つ以上の並行処理の排他制御を行うことが可能なように拡張することも可能です。
リストの解説を行うと分量が多くなるために、このアルゴリズムがどのように動作するのかを考えてみてくだ

さい。なお、命令のリオーダーが起こらないこと、機械語レベルの代入命令、比較命令はアトミック命令であることは前提です。

図 1 にソフトウェアによる排他制御を使用した例を示します。

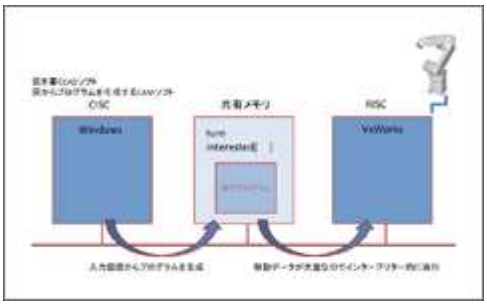

図 1. ソフトウェアによる排他制御の使用例

この例では RISC で実行される VxWorks 側と CISC で実行される Windows 側の排他制御に Peterson の排他制御を使用しています。
なお、図ではデータをコピーする間ずっと相手をブロックしているようで、ブロックしている時間が長すぎるように感じますが、実際には両者のデータの受け渡しにはリングバッファを用いるので、本当に相手をブロックしている時間はリングバッファのポインタを書き換えている間だけです。

次回は VxWorks などの RTOS が提供する排他制御、そして同期にも使われる、最も基本的な仕組みであるセマフォを中心にお話ししたいと思います。