WAIT_SSP アルゴリズム説明書

― 第1.0版 ―

承認	査閲	作成
		高橋和浩

2014年7月12日

1. 概要

シュリンク SSP から WAIT_SSP に改造したアリゴリズムをここに記載します。
できるだけ図表を交えて、わかりやすく解説します。

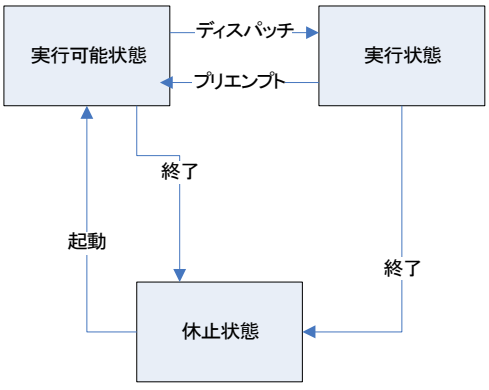
2. 実装された機能

カーネルとしての機能を記述します。

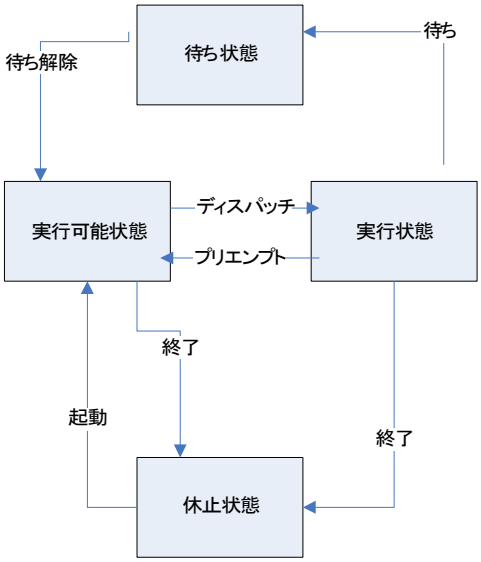
#	システムコール	概要	仕様
1	wai_tsk()	タスク WAIT	呼び出したタスクを WAIT 状態になる
2	go_tsk(id)	タスク Go	指定タスクを待ち状態から解除

表 1　追加されたシステムコール

実装したシステムコールとしては、wai_tsk() および　go_tsk()の 2 つです。
wai_tsk()は呼び出したタスクの待ち状態への遷移、go_tsk()は待ちタスクの
待ちの解除のシステムコールです。これに伴い、タスクとして待ち状態のサポート
をするものとなりました。　SSP は、実行可能状態、実行状態と休止状態の 3 状態し
か持ちませんでしたが、待ち状態を今回新たに加わりました。
待ち状態とは、タスクが頭から動くのではなく、途中から動くことになります。
そのため、SSP では実装されていなかった、コンテキストの保存/復元を行う
機能を追加しています。



SSP

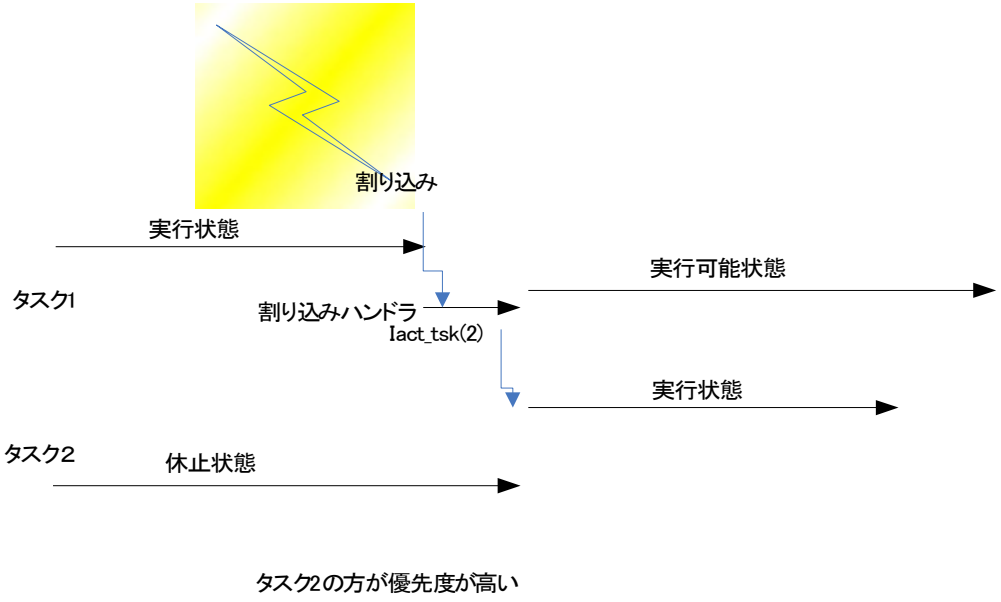


WAIT_SSP

3. アルゴリズム説明

3.1. タスク切り替えの基本

タスクが切り替わるとはどういうことでしょうか？
それは、何らかの事象により、実行中のタスクよりも優先度の高いタスクが
実行可能状態になり、タスクがスイッチすることです。
その「何らかの事象」とはどんなことで、どうやって発生するのか？
ですが、これは基本的には割り込みです。割り込みから、何らかの形のでカーネルに対し、レディーキューと呼ばれるタスクをスケジュールする順番を並び替えることが発生するからです。その結果レディーキューの並び替え後のスケジュールディスパッチでタスクが切り替わります。



3.2.コンテキストについて

コンテキストとは、基本的には、CPU のレジスタ全部のことと考えればほぼ正しいです。ほぼというのは、それに加えてカーネル固有の情報が加えられる場合があるということです。タスクが切り替わると、今まで実行が中断され、また切り替わって戻ってきた場合には、そこから引き続き、何事もなかったかの如く実行ができるればいいのです。マルチタスクのカーネル下でなくともそれを行っている部分があります。それは
1)関数呼び出し
2)割り込み
の 2 つです。その場合にコンパイラが適切にレジスタを保存/復元してくれているから

問題ないのです。ただし、関数呼び出しの場合と、割り込みでは少しコンデションが違うので保存/復元するデータ量が違っています。データ量が違うにせよ、コンパイラがここでコンテキストの保存と復元をしているわけです。これに加えてカーネルタスクスイッチする場合の保存と復元が追加されると考えればよいでしょう。

3.3. オリジナル SSP の場合のタスク切り替え時のコンテキスト

オリジナル SSP はほかの待ち状態のあるカーネルと違い、カーネルによるコンテキストの保存と復元は関数による保存と復元と仕組みそのままです。

理由は、プリエンプトと言っても、関数呼び出しと同じで、切り替わった先のタスク終了により、そのまま関数リターンと同じだからです。つまり、あるタスクから単純に優先度の高いタスクのメイン関数を関数呼び出しするのと同じでだからです。

これは待ち状態がないためこの実装が可能になっています。

3.4. 待ち状態がある場合のコンテキスト

SSP のもう一つの特徴としてタスク毎にスタック領域も確保していない点です。

通常の待ちのあるカーネルの場合は、タスク毎にスタック領域を持ちタスク切り替え時にタスクスタックに切り替えを行います。もちろんスタックポインタもコンテキストですのでタスク切り替え時に保存と復元が行われます。

タスクがプリエンプトされた場合および待ち状態になった時にコンテキストの保存が行われます。本来、カーネルのコンテキスト保存は割り込み動作の時と同じだけのコンテキストを保存すれば問題ありません。

まじめにカーネルのコンテキスト保存を行う場合は、割り込みハンドラでのコンパイラが出すコードを見本にすべてのレジスタを保存をおこなえばいいです。

RX の場合は

```
pushm   r1-r5           ; スクラッチレジスタをタスクスタックへ退避
pushm   r6-r15          ; スクラッチレジスタを退避
pushc   fpsw            ; FPU ステータスレジスタ退避
mvfacmi         r5
shll    #16, r5         ; ACC 最下位 16bit は 0 とする
mvfachi r4
pushm   r4-r5           ; アキュムレータ退避
```

これに加えて、スタックポインタを保存/復元すればいいです。

3.5. WAIT_SSP の場合のコンテキスト

具体的には、2 ケースに分けています。

1)タスク自らプリエンプトする場合

2)割り込みによってプリエンプトされてしまう場合
の 2 ケースです。
上記 3.4 で説明したまじめにコンテキストを保存するケースは WAIT_SSP では
割り込みの場合のみ利用します。

3.5.1. タスク自らプリエンプトする場合

これは、setjmp()/longjmp()を利用します。
setjmp()/longjmp()は基本的にはすべてのコンテキストの保存/復元を行います。
setjmp()の引数の変数にコンテキストが保存されます。
復元には、その引数だった変数を指定することで、setjmp()の false の
処理によって続行されます。

3.5.2. 実装上のロジック

コンテキスト情報は、

```
/*-----------------------------------------
 * タスク別コンテキスト保存領域                takahashi
 */
jmp_buf task_ctx[TNUM_TSKID];
/*-----------------------------------------
 * ディスパッチャーのコンテキスト
 */
jmp_buf disp_ctx;              //ディスパッチャコンテキスト
```

の 2 種類を持ちます。タスクのコンテキストは、タスクにディスパッチする場合は、このコンテキストを指定して longjmp()すればよいようなコンテキストを作る必要があります。それが、make_ctx()です。

```
/*-------------------------------------------------
 *  コンテキストの準備をしておく
  takahashi
 */
static jmp_buf jmpp;                    //このルーチンのセーブ用

void make_ctx(uint_t ipri)
{
  t_lock_cpu();
//printf("make_ctx ipri= %d\n",ipri);
  if (setjmp(jmpp) == 0)       //ここに戻り用
  {
            //続き
```

```
            // タスクスタックに切り替える
            set_task_stack(TOPPERS_TASKSTKPT(ipri ));
            if (setjmp(task_ctx[ipri]) == 0)
            {
                    /*登録した場合*/
                    longjmp(jmpp,1);//戻る
            }
            else
            {
                    /* タスク起動時 */
                    t_unlock_cpu();
                    /* タスクに来ました*/
                    /* タスク実行開始 */
                    (*((TASK)(tinib_task[ipri])))(tinib_exinf[ipri]);
                    disdsp = false;
                    /* ビットマップクリア.  */
                    primap_clear(ipri);

                    //タスクが終わった場合どうするのか？--> このあとは
                    //dispatcher()に行く
                    longjmp(disp_ctx,1);        //sta_ker の続きに行く
            }
    }
    else
    {
            //登録終了
            t_unlock_cpu();
    }
}
```

setjmp()の false の場合にタスクを頭から起動するように、コンテキスト情報が設定されていることがソースからわかるかと思います。make_ctx は登録だけしてすぐリターンします。jmpp を使って単純にリターンせず、一度 longjmp()で戻っているのは、タスクスタックの切り替え前に戻るためです。ディスパッチャーのコンテキストは、パワーオンからスケジューラに行くところのコンテキストを保存しています。タスクが正常に終了した場合は、ディスパッチャーのコンテキストを使って最初からスケジューラを経由してディスパッチ処理に行くよう

になります。ディスパッチャのコンテキストは、以下の部分で設定されています。

```
void
sta_ker(void)
{
  initialize_object();                              //必要
  kerflg = true;
  intnest = 0;
  setjmp(disp_ctx);//登録時も jump してもいずれも次へ行く
  dispatcher();
}
```

実際のスケジューラ dispatcher()は以下の通り

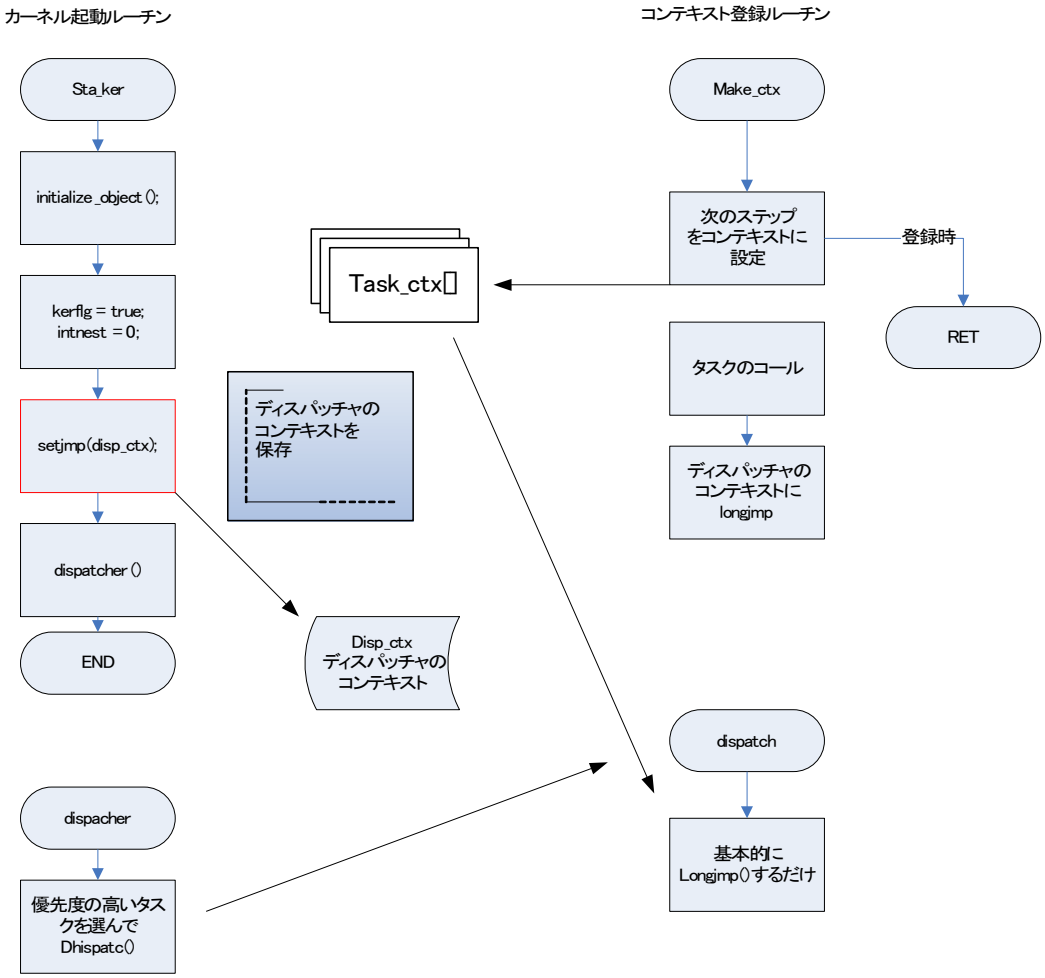
```
void
dispatcher(void)
{
  do {
            if(!primap_empty()) {
                      /* タスクの開始 */
                      //run_task(search_schedtsk());
                      dispatch(search_schedtsk());   //これからは帰ってこない
            }
            else {
                      /* 実行時優先度の初期化 */
                      runtsk_ipri = IPRI_NULL;
                      last_ipri = IPRI_NULL;              //ありえない値にする
                      idle_loop();
            }
  } while(true);
}
```

もう一度復習すると、

1)各タスクのディスパッチは、task_ctx[TNUM_TSKID]によって longjmp()するだけ longjmp()できるように準備しておく

2)タスクの終了時は、disp_ctx によって longjmp()すれば、新たにスケジューラによって　次のタスクが起動される。

ということになります。



3.5.3. タスクが途中でプリエンプトする場合はどうするのか

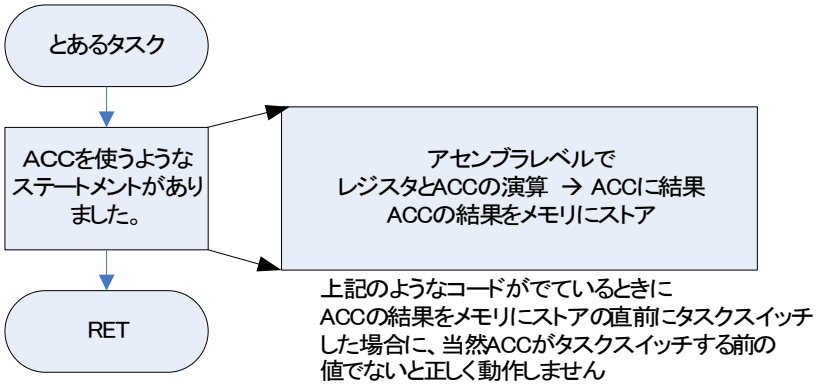
go_tsk()wai_tsk()のケースになります。切り替わるところで、自タスクのコンテキストを保存すれば、go_tsk()および wai_tsk()の続きの場所から復帰します。

```
ER
wai_tsk(void)
{
  ER                ercd;
  uint_t    tskpri;

//LOG_ACT_TSK_ENTER(tskid);
```

```
//CHECK_TSKCTX_UNL();
//CHECK_TSKID_SELF(tskid);

  tskpri = get_ipri_self(TSK_SELF);
  //tskpri = runtsk_ipri;
  t_lock_cpu();
  task_wait[tskpri] = 1;                //wait 状態
  primap_clear(tskpri);                 //レディ Q から削除

  //ここのコンテキクストを登録
  if (setjmp(task_ctx[tskpri]) == 0)
  {
          /*登録した場合*/
          longjmp(disp_ctx,1);          //sta_ker の続きに行く
  }
  else
  {
          // タスク復帰した場合
          t_unlock_cpu();
          return(ercd);
  }

  t_unlock_cpu();

   error_exit:

  return(ercd);
}
```

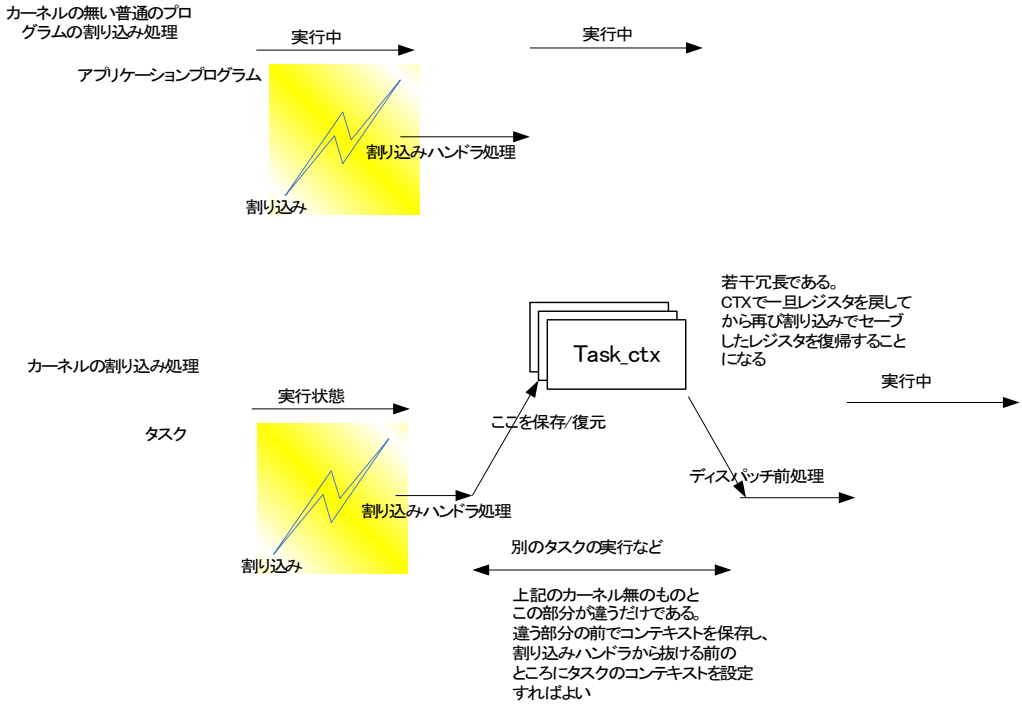
上記のように、自タスクコンテキストを保存して、登録時にスケジューラに longlmp()タスクに戻ってくるとこの wai_tsk()を正常に終了するような動作になります。

3.5.4. 割り込みによってプリエンプトされてしまう場合



あらかじめコンパイラ自身が判断できるところで切り替えが行われた場合にはsetjmp()/longjmp()にてコンテキストが保存されます。割り込みマスク状態やスタックポインタを含めすべてのレジスタが対象になってはいます。ですが、Cの関数として動作するので限界があります。戻り値としてレジスタが使われるのでそれは保存されません。ですが、コンパイラは保存されることを期待しない動作になりますので保存する必要がありません。また、RXにはFPUやアキュムレータというやっかいなレジスタが存在しますが、これはsetjmp()/longjmp()では保存されません。理由はCのコードの中でロードされて利用されるものでアセンブリコードレベルで、中断される場合には保存は必要ですがCのコードでsetjmp()でFPUをロードしものを保存する必要がないからです。つまり、Cのコンパイラに対して、「今からどっか飛んでいくけど戻ってくる場合はここよ」ということを言い渡しているようなことなので 、FPUなどのレジスタのロード状態が保存されていないものとして動作するので、問題になりません。一方、コンパイラに無断で、でていった場合には、これは厄介なレジスタを操作中の場合も考えられます。なのでコンパイラが知らないところで切り替えられた場合には全レジスタの保存をする必要があります。

3.5.5. 割り込みによってプリエンプトされてしまう場合の実装



考え方は、割りこまれたら割りこまれた状態に戻って、割り込みハンドラから戻ればいい。あくまで、割り込みによって今まで実行中だったタスクが、元に戻るときに全レジスタを保存復元して戻ればいいのです。つまり、一旦、割り込みハンドラの出口にコンテキストを保存します。そうすることで、ディスパッチされると割り込みハンドラの出口に来ます。再び割り込みハンドラが保存していたレジスタを全復帰して割り込み前に戻る仕組みです。
最後のこの割り込みを考慮したコンテキストの保存と復元がちょっと説明が難しいかもしれません。実際この説明をみるより、ソースコード prc_support.src を見た方が早いかもしれません。

以上