数理解析研究所講究録
第 470 巻 1982 年 14-28

14

Ada の待ち合わせの標準形について

石畑　清　（東京大学・理学部・情報科学科）
筧　捷彦　（立教大学・理学部・数学科）

Ada の並行処理機能が実際的な問題の記述にどの程度有効であるかを調べるために PV-chunk と呼ばれる問題に対する Ada の解を検討した。PV-chunk は従来の技法ではうまい解が得られなかったが、ここで報告する新しい技法によって解決できることがわかった。この技法は、メッセージの仲介を専門に行なうタスクを介在させるもので、制御プログラムが各タスクの状態を完全に把握することを可能にする。PV-chunk の問題だけでなく Ada における並行処理一般に適用できる１つのプログラミング上の指針を与えるのではないかと考えている。この技法の特徴、実際のシステムプログラムに使用する場合に留意すべき点、および Ada における並行処理の問題点を中心に論ずる。

1　はじめに

Ada〔3〕における並行処理は、メッセージの交換というただ１つの手段でタスク間の通信と同期の両方の問題を解決しようとしている。このような考え方は、CSP〔1〕に端を発したもので、Ada 自身 CSP に強い影響を受けている。

メッセージ交換方式の並行処理が実際的な問題の記述にどの程度適しているか、まだ明らかでない点が多い。特に、Ada は従来の並行処理機能だけを強調した実験的言語とは違い、実用を目ざした本格的言語である。並行処理プログラミングにおいても、通信や同期の問題だけでなく、型の扱い方、名前の有効範囲などへも配慮が必要とされる。Ada の並行処理の長所、欠点については、すでに数多くの議論があるが、共通の理解を得るためには、さらに多くの研究を積み重ねる必要がある。

本報告は、実験的な手法、すなわち特定の例題を設定しそれに対する Ada の解を検討するという方法で、Ada の並行処理のいろいろな側面を考える。ここで取り上げる例題は、PV-chunk と呼ばれる問題である。

本報告の構成は次のとおりである。第２節では、PV-chunk の問題を紹介し、Ada で記述することが困難である理由を説明する。第３節では、Ada Rationale の解を示し、批判を加える。第４節では、仲介タスクの技法とそれを使った解を示す。第５節では、Ada による並行処理プログラミングを容易にするための技法について論ずる。第６節では、異常事態に対処する方法を論じ、現在の Ada の文法では記述しきれない場合があることを示す。第７節はまとめである。

なお、第２節以下では読者が Ada の文法に関する知識を持っていることを仮定している。

2 PV-chunk

PV-chunk は資源の排他制御問題の一種である。通常の semaphore の場合、資源の要求は必ず１個ずつ行なわれる。これに対し、複数の資源を同時に要求できるように拡張した semaphore を PV-chunk と呼ぶ。PV-chunk では、資源の要求は

P(n)

という形で行なわれる。n は使いたい資源の数である。資源管理プログラムは n 個以上の資源が利用可能な時に限り利用許可を与える。資源が不足している時は、要求された量の資源が回復するまでこのユーザーを待たせる。資源の返却は

V(n)

の形式を用いる。なお、PV-chunk が同一種類の複数個の資源を扱うのに対し、複数種類の資源が１個ずつある場合を考えることがあり、これを PV-multiple と呼ぶ。PV-chunk と PV-multiple の間に記述の上で本質的な違いはないので、以下では PV-chunk だけを扱うことにする。（以下のプログラムでは P 命令を REQUEST、V 命令を RELEASE という名前で呼ぶ）

以下に示すように、Ada で PV-chunk を記述するのは容易ではない。このため、逆に Ada の並行処理機能の記述能力を検討するための格好の例題だと思われる。この節では、Ada で PV-chunk を記述するのがどのような点で困難であるかを明らかにするために、いくつかの解を順次示すことにする。最初に最も素朴な解を示す。

```
task RESOURCE_CONTROL is
   entry REQUEST(1..INIT_AVAIL);
   entry RELEASE(SIZE:INTEGER range 1..INIT_AVAIL);
end RESOURCE_CONTROL;

task body RESOURCE_CONTROL is
   AVAIL:INTEGER range 0..INIT_AVAIL;
begin
   AVAIL:=INIT_AVAIL;
   loop
      select
         when AVAIL>=1 =>
            accept REQUEST(1) do AVAIL:=AVAIL-1; end;
      or
         when AVAIL>=2 =>
            accept REQUEST(2) do AVAIL:=AVAIL-2; end;
      or
         ...
      or
         when AVAIL>=INIT_AVAIL =>
            accept REQUEST(INIT_AVAIL) do AVAIL:=AVAIL-INIT_AVAIL; end;
      or
         accept RELEASE(SIZE:INTEGER range 1..INIT_AVAIL) do
            AVAIL:=AVAIL+SIZE;
         end RELEASE;
      end select;
   end loop;
end RESOURCE_CONTROL;
```

プログラム 1

PV-chunk は要求された資源の量によって、要求を受け付けるべきかどうかを判断する必要がある。ところが、Ada の accept 文にはパラメーターの値によって accept するかどうかを選択する機能がない。このため、単純に資源要求量を entry 呼び出しのパラメーターとするだけではうまくいかない。どういう形で判断の材料を資源管理タスクに伝えるかがプログラミング上の問題点となる。

プログラム１では、この問題を REQUEST という entry を entry family とすることによって解決している。REQUEST に属する個々の entry にはパラメーターはない。REQUEST のパラメーターを entry family の添字に置き換えることによって、accept する前に when 条件で判定できるようにしたわけである。

この解の問題点は、資源の数が変わるたびに select 文の選択肢の数を変えなくてはならないことである。これでは実用プログラムとして使いものにならないであろう。マクロのような機能を持った前処理系を使えば、パラメーターの数だけ選択肢をコピーするようなことも可能だろうが、ここでは Ada 言語から逸出することは避けることにする。Ada の generic 機能は一種のマクロであるが、大きな構文単位の扱いを目的としているため、このような細かな点まで記述することはできない。なお、CSP にはこのような同一形態の複数の選択肢をまとめて書く記法が用意されている。

プログラム１のようなプログラムを書く際に、初心者はよく次のような誤りを犯す。

```
task body RESOURCE_CONTROL is
   AVAIL:INTEGER range 0..INIT_AVAIL;
begin
   AVAIL:=INIT_AVAIL;
   loop
      select
         when N<=AVAIL =>
            accept REQUEST(N) do        -- wrong !!
               AVAIL:=AVAIL-N;
            end REQUEST;
      or
         accept RELEASE(SIZE:INTEGER range 0..INIT_AVAIL) do
            AVAIL:=AVAIL+SIZE;
         end RELEASE;
      end select;
   end loop;
end RESOURCE_CONTROL;
```

プログラム　2

初心者は、entry family に対する accept 文が、その family 全部に対する entry 呼び出しの中から適当なものを選んで accept してくれると考えがちである。この考えが正しければ、プログラム２はプログラム１と等価であり、ずっとエレガントなのだが、残念ながらこの考えは誤りである。accept 文の実行が始まると、最初に entry family の添字の式が評価される。そしてその１個の entry だけしか調べに行ってくれないのである。

プログラム２のような記法が許されないとすると次のようにループによって順番に調べることが考えられる。

```
task RESOURCE_CONTROL is
   entry REQUEST(1..INIT_AVAIL);
   entry RELEASE(SIZE:INTEGER range 1..INIT_AVAIL);
```

```
end RESOURCE_CONTROL;

task body RESOURCE_CONTROL is
   AVAIL:INTEGER range 0..INIT_AVAIL;
begin
   loop
      for I in 1..INIT_AVAIL loop
         select
            when I<=AVAIL =>
               accept REQUEST(I) do
                  AVAIL:=AVAIL-I;
               end REQUEST;
         or
            accept RELEASE(SIZE:INTEGER range 1..INIT_AVAIL) do
               AVAIL:=AVAIL+SIZE;
            end RELEASE;
         else
            null;
         end select;
      end loop;
   end loop;
end RESOURCE_CONTROL;
```

プログラム　3

この解では、資源管理タスクは REQUEST に対する entry 呼び出しがあるかどうかをループを回りながら調べている。ユーザーからの要求がなくとも、常に走り続けているわけである。このような方式を busy wait と言い、これを使えば解けるのは当然である。busy wait は CPU の使用効率の点で受け入れ難いので、以下ではなるべく busy wait を使わない方法、すなわち各タスクとも仕事がない限り眠っているような方法を考える。

今までの３つの解の共通点は entry のパラメーターを entry family の添字で置き換えたことであった。このようにしたのは、accept するまで entry 呼び出しのパラメーターの内容を知ることができないからである。本当は、パラメーターの内容を accept する前に知って、それによって accept するかどうかを判断できれば、余計な苦労がなくなって一番良いわけである。この意味で、 select 文の when 条件中に entry のパラメーターを含んだ式を書けるようにした方が良いという意見がある。これによれば、PV-chunk は次のように書ける。

```
task RESOURCE_CONTROL is
   entry REQUEST(SIZE:INTEGER range 1..INIT_AVAIL);
   entry RELEASE(SIZE:INTEGER range 1..INIT_AVAIL);
end RESOURCE_CONTROL;

task body RESOURCE_CONTROL is
   AVAIL:INTEGER range 0..INIT_AVAIL;
begin
   AVAIL:=INIT_AVAIL;
   loop
      select
```

```
            when SIZE<=AVAIL =>                    -- not allowed in Ada
               accept REQUEST(SIZE:INTEGER range 1..INIT_AVAIL) do
                  AVAIL:=AVAIL-SIZE;
               end REQUEST;
         or
            accept RELEASE(SIZE:INTEGER range 1..INIT_AVAIL) do
               AVAIL:=AVAIL+SIZE;
            end RELEASE;
         end select;
      end loop;
   end RESOURCE_CONTROL;
```

プログラム　4

この方が問題の本質を明らかにしたエレガントな解であると言えよう。Ada にこのような機能が採用されなかったのは、名前の有効範囲、インプリメントの都合などの理由のためと考えられるが、双方ともさほど解決が困難だとは思えない。プログラム 4 のような簡明な解の可能性を閉ざしているのは遺憾である。このような記法が今後 Ada に取り入れられる可能性はないと思われるので、これについてはこれ以上触れないことにする。

もちろん、Ada のランデブーを使って semaphore やモニターを実現できるから原理的には Ada の世界でなんでも記述できるわけだが、それではあまりに不便である。次節以降、現在の Ada の文法で許される範囲で、より現実的な PV-chunk の解を追求する。

3　Preliminary Ada Rationale の解

Preliminary Ada の Rationale ((2), pp.11-23～24) に PV-multiple の解が載っている。問題を PV-chunk に変えて、現在の Ada の文法に合うように書き直すと次のようになる。

```
package RESOURCE_CONTROL is
   INIT_AVAIL:constant INTEGER:=100;          -- some predetermined size
   subtype REQ_RANGE is INTEGER range 1..INIT_AVAIL;
   procedure REQUEST(SIZE:REQ_RANGE);
   procedure RELEASE(SIZE:REQ_RANGE);
end RESOURCE_CONTROL;

package body RESOURCE_CONTROL is
   task MANAGER is
      entry FIRST(SIZE:REQ_RANGE; OK:out BOOLEAN);
      entry AGAIN(SIZE:REQ_RANGE; OK:out BOOLEAN);
      entry FREE(SIZE:REQ_RANGE);
   end MANAGER;

   task body MANAGER is
      AVAIL:INTEGER range 0..INIT_AVAIL;
   begin
      AVAIL:=INIT_AVAIL;
```

```
      loop
         select
            accept FIRST(SIZE:REQ_RANGE; OK:out BOOLEAN) do
               if SIZE<=AVAIL then
                  AVAIL:=AVAIL-SIZE;
                  OK:=TRUE;
               else
                  OK:=FALSE;
               end if;
            end FIRST;
         or
            accept FREE(SIZE:REQ_RANGE) do
               AVAIL:=AVAIL+SIZE;
            end FREE;
            for I in 1..AGAIN'COUNT loop
               select
                  accept AGAIN(SIZE:REQ_RANGE; OK:out BOOLEAN) do
                     if SIZE<=AVAIL then
                        AVAIL:=AVAIL-SIZE;
                        OK:=TRUE;
                     else
                        OK:=FALSE;
                     end if;
                  end AGAIN;
               else
                  exit;
               end select;
            end loop;
         end select;
      end loop;
   end MANAGER;

   procedure REQUEST(SIZE:REQ_RANGE) is
      OK:BOOLEAN;
   begin
      MANAGER.FIRST(SIZE,OK);
      while not OK loop
         MANAGER.AGAIN(SIZE,OK);
      end loop;
   end REQUEST;

   procedure RELEASE(SIZE:REQ_RANGE) is
   begin
      MANAGER.FREE(SIZE);
   end RELEASE;
end RESOURCE_CONTROL;
```

プログラム　5

この解では、ユーザータスクは package の中の手続きを通して資源管理タスクの entry を呼び出す。このうち RELEASE は FREE という entry を呼ぶだけで問題はない。一方、REQUEST は、資源管理タスクの entry FIRST を最初に１回呼び、その１回で許可が得られない時は許可が出るまで AGAIN を繰り返し呼び出すという仕掛けになっている。資源管理タスクは、FREE を受け付けるたびに AGAIN で待っているタスクをいったんすべて起こす。そして、それぞれの要求が新しく解放された資源によって満たされるかどうか調べ、可能なタスクに対しては使用許可を与える。

この解の特徴は、資源要求を受け付けるかどうかの判断を select 文の when 条件で行なわず、accept 文の中で行なうようにしたことである。これによって entry family を使わずに解くことに成功している。しかし、entry 呼び出しを受け付けた上で拒否し、もう一度呼び出させるというやり方では、どうしてもどこかに無理が生じる。

ユーザータスクは資源のあきを待っている間、RELEASE のたびにいったん起きて他のタスクと新しく解放された資源の取り合いをする。そして、資源を確保できなければ再び待ちに入る。これは、ほとんどの時間眠っているという点で通常と異なるが、一種の busy wait にほかならない。起こされたらすぐに資源を使えるようになっていなければ、本当の意味で busy wait を避けたとは言えないのである。このため、次に示すような問題が起こる。

１回の AGAIN 呼び出しから次回の AGAIN 呼び出しまでの間、各タスクはそれぞれ自分のペースで走る。スケジューリング上の問題や CPU が遅い、メッセージ転送に時間がかかるなどの理由で他のタスクより出足の遅いタスクがあると、このタスクは自分の要求が実際には満たされる瞬間が何回もあるにもかかわらず、常に他のタスクに追い越されてしまうという可能性がある。これでは公平な資源割り当てとは言えない。このような問題を individual starvation と言う。individual starvation は、この解のようにユーザーに自由競争をさせている限り避けられない。何らかの形で資源管理タスクがユーザーの順序を指定してやることが必要である。

プログラム５にはもう１つもっと重大な問題点がある。２つのユーザータスクがあったとして、一方が資源を使用中にもう一方が資源の利用を要求して拒否されたとする。するとこのタスクは FIRST の呼び出しから戻って AGAIN の呼び出しへ行くわけだが、AGAIN に到達する前に第１のユーザーが RELEASE で資源を返却してしまったとする。資源管理タスクは AGAIN で待っているタスクを探すが、この時点では誰も待っていないので何もしない。しかる後に第２のタスクが AGAIN を呼び出すと、すでに資源は利用可能な状態にあるにもかかわらず、entry 呼び出しを accept してくれないので先へ進めない。この状態は誰か別のユーザーが資源を利用して返却してくれるまで続くわけで、これはかなり深刻な欠陥と言うべきであろう。

これを避けるために、AGAIN で待っているタスクの数を調べるかわりに FIRST または AGAIN で拒絶したユーザーの数を数えておくという方法や、誰も使用していない状態では AGAIN を受け付けられるようにしておくなどの対策が考えられるが、いずれにしてもプログラムは複雑になるし、dead lock の問題も面倒になる。

最後に、細かい点になるが、プログラム５について Rationale に指摘してある２つの注意事項を書いておく。第１点は、FREE の後で AGAIN を順番に受け付ける部分が for 文で AGAIN'COUNT （ AGAIN で待っているタスクの数を返す attribute ）回だけ繰り返すようになっていることである。for 文は上下限の値を最初に１回だけ評価するので、AGAIN'COUNT の値も１回しか評価されない。これを while 文で１回ごとに AGAIN'COUNT を評価することにすると、AGAIN で拒否されたユーザーがすぐ AGAIN の列の後に並ぶので、いつまでたっても AGAIN の列が尽きないことになる。もう１点は、AGAIN の受け付けをする accept 文が select 文の中に入れてあることである。これは、AGAIN を呼んだタスクが途中で abort などの原因によって死んでしまった場合にシステムが dead lock に陥ることがないように、else 部でバイパスできるようにしたものである。

4　仲介タスクを利用する解

PV-chunk　に対する今までの解は、資源利用の要求を１回のランデブーで受け付けていた。これを改めて、２回以上のランデブーをあるプロトコルに従って組み合わせて初めて資源が利用できるようにすればうまくいく可能性がある。１回目のランデブーによってユーザーの必要とする資源の量がわかるので、２回目のランデブーを受け付けるかどうかを資源管理タスクが主体性を持って決めることができるからである。

１つの要求を２回のランデブーで受け付ける場合、資源管理タスクの側から見て、次の２つの手順が考えられる。

1　accept　->　accept
2　accept　->　entry 呼び出し

１、２ともに最初の accept 文によってユーザーの要求量を知り、２回目のランデブーを受け付けるかどうかを決める点は同じである。２つの方法が違うのは、２回目のランデブーをユーザーからの entry 呼び出しにするか、逆に資源管理タスクからの entry 呼び出しにするかという点である。

第１の方法では、２回目のランデブーの相手を資源管理タスクが指定することができない。accept　文からは呼び出しタスクが誰であるかを意識しないことになっているからである。特定のユーザータスクを選んで accept したいのだからこれでは具合が悪い。entry family　にして、各ユーザーに添字を１つずつ割り振ることも考えられるが、ユーザーが誤って違う添字を使った場合に困るし、ユーザータスクの数の上限をあらかじめ知っていなくてはならないのも好ましくない。

これに対して、第２の方法では資源管理タスクからユーザータスク側の entry　を呼び出すことになる。entry 呼び出しは entry　を持つタスクの名前を指定して行なうから、別人に使用許可を出してしまうおそれはなく、タスクの選択は完全に行なえる。そこで、ここでは第２の方法を使うことにする。

ところで、entry を呼び出すには entry　を持つタスクの型がわかっていなくてはならない。ユーザータスクの型を全部知っておくのは不可能であるから、直接ユーザータスクを呼ぶわけにはいかない。この問題を解決するために、ユーザータスクと資源管理タスクの間にメッセージの仲介だけを行なう専用のタスクを置くことにする。以上の方針で書いたのがプログラム６である。

```
package RESOURCE_CONTROL is
   INIT_AVAIL:constant INTEGER:=100;        -- some predetermined size
   subtype REQ_RANGE is INTEGER range 1..INIT_AVAIL;
   generic package RESOURCE_USER is
      procedure REQUEST(SIZE:REQ_RANGE);
      procedure RELEASE(SIZE:REQ_RANGE);
   end RESOURCE_USER;
end RESOURCE_CONTROL;

package body RESOURCE_CONTROL is
   type MESSENGER_TASK;
   type MESSENGER is access MESSENGER_TASK;
   task type MESSENGER_TASK is
      entry REQUEST(SIZE:REQ_RANGE);
      entry RELEASE(SIZE:REQ_RANGE);
      entry ACKNOWLEDGE;
      entry INIT(ID:MESSENGER);
```

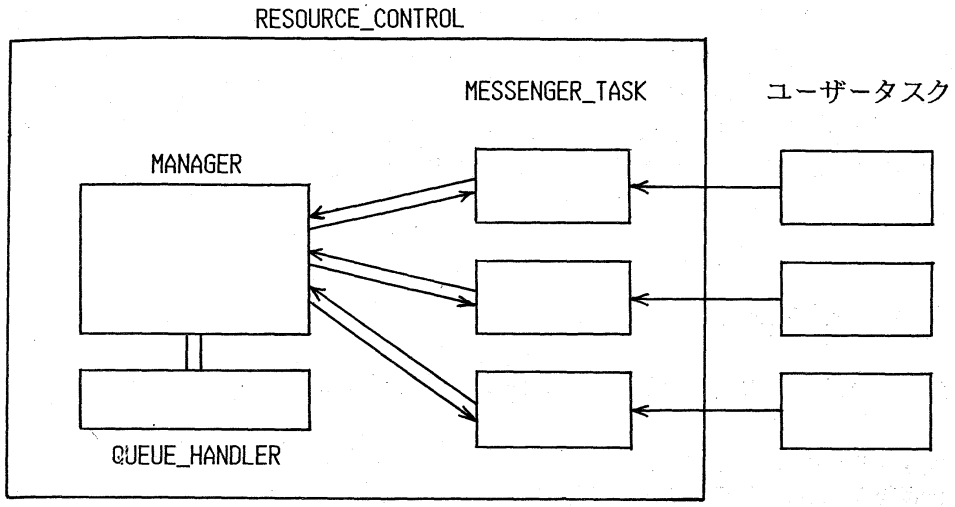
```
end MESSENGER_TASK;

task MANAGER is
   entry MARK(SIZE:REQ_RANGE; ID:MESSENGER);
   entry FREE(SIZE:REQ_RANGE);
end MANAGER;

task body MANAGER is
   AVAIL:INTEGER range 0..INIT_AVAIL;

   package QUEUE_HANDLER is
      procedure ENTER_QUEUE(SIZE:REQ_RANGE; ID:MESSENGER);
      procedure REMOVE_QUEUE(SIZE:out REQ_RANGE; ID:out MESSENGER);
   end QUEUE_HANDLER;

   package body QUEUE_HANDLER is
      type ELEMENT;
      type LIST is access ELEMENT;
      type ELEMENT is
         record
            SIZE:REQ_RANGE;
            CALLER:MESSENGER;
            PRED,SUCC:LIST;
         end record;
      FREE_LIST,QUEUE_HEAD,QUEUE_TAIL:LIST;

      procedure ENTER_QUEUE(SIZE:REQ_RANGE; ID:MESSENGER) is
         P:LIST;
      begin
         if FREE_LIST=null then
            P:=new ELEMENT;
         else
            P:=FREE_LIST;
            FREE_LIST:=FREE_LIST.SUCC;
         end if;
         P.SIZE:=SIZE;
         P.CALLER:=ID;
         P.PRED:=QUEUE_TAIL.PRED;
         P.SUCC:=QUEUE_TAIL;
         QUEUE_TAIL.PRED:=P;
      end ENTER_QUEUE;

      procedure REMOVE_QUEUE(SIZE:out REQ_RANGE; ID:out MESSENGER) is
         P:LIST;
      begin
         P:=QUEUE_HEAD.SUCC;          -- very simple scheduling
         while P/=QUEUE_TAIL loop
            if P.SIZE<=AVAIL then
               SIZE:=P.SIZE;
               ID:=P.CALLER;
```

```
                P.PRED.SUCC:=P.SUCC;
                P.SUCC.PRED:=P.PRED;
                P.SUCC:=FREE_LIST;
                FREE_LIST:=P;
                return;
             end if;
             P:=P.SUCC;
          end loop;
          ID:=null;
       end REMOVE_QUEUE;

    begin
       FREE_LIST:=null;
       QUEUE_HEAD:=new ELEMENT;
       QUEUE_TAIL:=new ELEMENT;
       QUEUE_HEAD.SUCC:=QUEUE_TAIL;
       QUEUE_TAIL.PRED:=QUEUE_HEAD;
    end QUEUE_HANDLER;

begin
    AVAIL:=INIT_AVAIL;
    loop
       select
          accept MARK(SIZE:REQ_RANGE; ID:MESSENGER) do
             QUEUE_HANDLER.ENTER_QUEUE(SIZE,ID);
          end MARK;
       or
          accept FREE(SIZE:REQ_RANGE) do
             AVAIL:=AVAIL+SIZE;
          end FREE;
       end select;
       loop
          declare
             SIZE:REQ_RANGE;
             ID:MESSENGER;
          begin
             QUEUE_HANDLER.REMOVE_QUEUE(SIZE,ID);
             exit when ID=null;
             AVAIL:=AVAIL-SIZE;
             ID.all.ACKNOWLEDGE;
          exception
             when TASKING_ERROR => AVAIL:=AVAIL+SIZE;
          end;
       end loop;
    end loop;
end MANAGER;

task body MESSENGER_TASK is
    THIS_TASK:MESSENGER;
begin
```

```
        accept INIT(ID:MESSENGER) do
           THIS_TASK:=ID;
        end INIT;
        loop
           select
              accept REQUEST(SIZE:REQ_RANGE) do
                 MANAGER.MARK(SIZE,THIS_TASK);
                 accept ACKNOWLEDGE;
              end REQUEST;
           or
              accept RELEASE(SIZE:REQ_RANGE) do
                 MANAGER.FREE(SIZE);
              end RELEASE;
           end select;
        end loop;
     end MESSENGER_TASK;

     package body RESOURCE_USER is
        P:MESSENGER;

        procedure REQUEST(SIZE:REQ_RANGE) is
        begin
           P.all.REQUEST(SIZE);
        end REQUEST;

        procedure RELEASE(SIZE:REQ_RANGE) is
        begin
           P.all.RELEASE(SIZE);
        end RELEASE;

     begin
        P:=new MESSENGER_TASK;
        P.all.INIT(P);
     end RESOURCE_USER;
  end RESOURCE_CONTROL;
```

プログラム　6

プログラム６の構造は次の図のようになっている。



ユーザーには今までと同様 REQUEST と RELEASE という手続きを見せるが、実際はこの手続き呼び出しは各ユーザーごとに作られた仲介タスク（MESSENGER_TASK）への呼び出しになる。仲介タスクは渡された要求と自分自身の access 値を資源管理タスク（MANAGER）に渡して、許可の返事があるまで accept 文で待つ。この accept 文はユーザータスクからの要求を受け付ける accept 文の中にあるので、この間ユーザータスクも待ちの状態にある。許可が出ると仲介タスクは accept 文からぬけ、ユーザータスクも実行を再開する。

一方、資源管理タスクは要求タスクの名前と要求量を組にして queue に登録する。そして、利用要求や返却を受け付けるたびに queue を調べて利用を許可すべきユーザーを適当に選び、ACKNOWLEDGE という entry を呼んで利用許可を与える。

ここでは、ユーザーを選ぶアルゴリズムとして、長く待たされているユーザーを優先する簡単な方法を採用しているが、これは本質的ではない。スケジューリングアルゴリズムの選び方は自由である。全ユーザーの状態を把握しているため、スケジューリングはごく普通のデータ処理の問題に還元されている。第３節までに述べたような微妙なケースへの配慮は不要である。

プログラム６の中で技巧的なのは、仲介タスクの名前（この場合は access 値）を資源管理タスクに教えてやる方法である。使用許可の entry 呼び出しをする時、資源管理タスクは個々の仲介タスクを指名するので、前もってこの仲介タスクの名前を知っている必要がある。１回目のランデブーの時に、accept 側の資源管理タスクから呼び出し側の仲介タスクの名前を調べられるようになっていれば良いが、Ada にこの機能はない。このため、仲介タスクはユーザーの要求量のほかに自分自身の access 値をパラメーターとして資源管理タスクを呼び出している。資源管理タスクはこの値をローカル変数にしまっておいて、entry の呼び出しの際に使う。

ところで、仲介タスクはどうやって自分自身の access 値を知ることができるのか？ここでも、Ada はタスクが自分の access 値を知る方法を提供していない。そこで、最初に仲介タスクを作った直後に、access 値を教えてやる目的で初期化用の entry INITIALIZE を呼び出すことにする。仲介タスクは渡された access 値をローカル変数 THIS_TASK にとっておいて、資源管理タスクに対する要求の際に使う。この点がこのプログラムのみそと言えよう。

プログラム６では、ユーザーに generic package RESOURCE_USER だけを見せており、ユーザーはこの package を instantiate しないと資源要求ができないようになっている。これには、仲介タスクなどの資源管理の詳細を隠すだけでなく、ユーザーに仲介タスクを作ってもらう手続きを自動化するという目的がある。仲介タスクと資源管理タスクの交信

のプロトコルや、仲介タスクの型自身は資源管理プログラムの一部であるが、個々の仲介タスクの実体はユーザータスクに作ってもらわなければならない。ところが、仲介タスクを作る際、初期化用の entry を呼び出すなど手順が複雑になるので、仲介タスクの作り方は資源管理プログラムが提供する形にしたい。以上の点をすっきり解決するには generic package を使うのが良いと考えた。各ユーザーは generic package を instantiate しなければ REQUEST、RELEASE を呼べない。instantiate すれば、package の本体で自動的に仲介タスクが作られるという仕掛けである。

5 Ada における並行処理について

プログラム６は、次の２点でそれ以前の解より優れている。

1 busy wait が１つもない。各タスクは要求が受け付けられるまで眠っており、起きた直後から資源を使うことができる。
2 スケジューリングを制御できる。今までの解は、ある意味で行きあたりばったりに走らせるタスクを選び、資源管理タスクの意志を表現することができなかった。

この節では、このような成果を生んだ要因を考察する。

5.1 ランデブーの分割

Ada の言語仕様として備わっている待ちの機能はランデブー１つしかない。このことの意味はモニターと比較するとはっきりするだろう。モニターではモニター手続きの入口における待ちと条件変数による待ちの２種類を用意している。このため、PV-chunk の問題もモニターを使えば比較的簡単に解ける。資源不足のユーザーを待たせる条件変数（スケジューリング上必要があれば要求資源の数を添字とする配列にする）を作ればよい。条件変数による待ちはモニター手続きのパラメーターを知った上で使えるから、第２節で述べたような苦労はしなくてすむ。

これに対して、Ada ではランデブー、すなわち entry 呼び出しと accept 文でしか待つことができない。accept 文中で呼び出し側のタスクを待たせたまま別のランデブーをすることはできるが、複数のランデブーが完全に last in first out の関係になっていなければならない。したがって、一方のユーザーは待たせるがもう一方のユーザーは走らせるということが不可能であり、複数のユーザーの要求を総合的に判断するようなことが困難になる。これがプログラム１～５がうまくいかない理由であった。これを克服するためには、プログラム６のように１つの仕事を２回以上のランデブーに分割してメッセージ交換の機会を増やすことが有効である。これによって２回のランデブーの間、プログラムは自由にふるまうことが可能になり、複雑なアルゴリズムを記述できる。

なお、CSP ではランデブー時の情報の流れが１方向だったのに対し、Ada はパラメーターを通して両方向のデータの転送が可能であり、ランデブーの回数を減らす効果があるとされてきた。しかし、これも場合によりけりで、Ada でも CSP のように要求とそれに対する返事を２回のランデブーに分けた方がうまくいく場合が多いのではないかと考えている。

5.2 タスクの分割

プログラム６で仲介タスクを導入したのは、交信するタスクの型を統一するためであったが、ユーザーごとにタスクを作るという方針には６．１に示すように利点が多い。一般に、問題が困難な時はタスクを分割してみるべきである。タスクは多めに作った方がプログラムのわかりやすさの点でも、busy wait や dead lock を避ける上でも好都合であることが多い。このようなことは、言語の設計時から意識されていたようである。

タスクをたくさん作ることによって、個々のタスクは自分の仕事に専念できるようになる。他のタスクとの交信の種類も減り、その処理の記述がなかば決定的（deterministic）な、理解しやすいものとなる。複雑で非決定的（non-deterministic ）な部分は、スケジューリングの問題に帰着させて、Ada の実行時ルーチンにまかせてしまうわけである。

なお、このような方針をとると、タスク間の制御の切り換え（context switching ）が増え、システム全体の効率が落ちるおそれがある。多くのタスクを含むシステムの最適化の研究が必要になると思われる。

6　異常事態に対する対策

資源管理プログラムをシステムプログラムとして使う場合、ユーザーがどんなことをしてもシステムとしての機能は損なわれないようにしておく必要がある。その例として、ここでは次の２つのケースに対する対策を考える。

1 REQUEST していないユーザータスクが不当な RELEASE 命令を出してシステム全体を混乱させるのをどうやって防ぐか。
2 REQUEST して RELEASE せずに死んでしまうタスクがあったらどうするか。

6．1　タスクの識別の問題

Ada では、entry を呼び出す側のタスクは entry を持っているタスクの名前を指定しなければならないが、逆に accept する側のタスクは呼び出すタスクの名前を指定しない。Ada の並行処理が CSP と最も大きく異なるのはこの点である。このため、悪意を持った、または虫のあるユーザータスクが存在して、不当な RELEASE 命令を実行した場合、CSP では無視するだけで問題とならないが、Ada ではシステムに異常をきたしてしまう。

これを防ぐために、ユーザータスクの資格を判定するための情報をどこかに求めなければならない。まず考えられるのは、タスクを識別するようなパラメーターを entry に追加することである。資源管理タスクが REQUEST で所定のキーをセットし、RELEASE の際にチェックするわけである。このパラメーターは、通常の整数などでは他のタスクの偽装を見破れないから不適当で、limited private 型としてユーザーが変更できないようにしておく。ただし、limited private 型としても２つ以上のユーザータスクが共有変数を使っているとうまくいかないケースがあることを〔4〕は示している。

この問題は、本来 accept 文の中から呼び出したタスクを識別できるようになっていれば、簡単に解決するはずである。これは、accept 文がどのタスクからの呼び出しも差別せず受け付けるという方針と矛盾しない。そもそも、実行時ルーチンはどのタスクから呼ばれたかの情報を持っているはずだから、実現も難しくない。むしろ問題なのは、どのような形で言語仕様の中に取り入れれば言語全体の整合性が保たれるかということである。ここでは、accept 文中で entry を呼び出したタスクの名前を返す attribute を新設することを提案したい。呼び出しタスクの名前は代入と比較ができれば十分なので、private 型とするのが良いと思われる。

なお、プログラム６では、以上述べたような面倒なことを考える必要はなく、仲介タスクが REQUEST なしの RELEASE 要求を拒否するようにしておけばよい。ユーザーがどんなことをしても、仲介タスクの段階でチェックするので、中央には影響が及ばないようにできる。

6．2　タスクの死亡の検出

第２の問題は、基本的にはタスクが死んだことをどうやって他のタスクが検知するかということである。この問題は第１の問題より難しく、結論から言うと現在の Ada ではうまい解がないと思われる。

Ada の select 文には、他のタスクが死んで自分の存在理由がなくなった時に自分も死ぬための terminate という選択肢があるが、これは死ぬだけでその前に何らかの動作をすることを許していないので、この場合には使えない。目標とするタスクが死んだ時に RELEASE を代行することができなくては意味がないからである。terminate を検出するタスクを他のタスクが見張るようなことも考えてみたが、見張りタスクが動作状態にある限り terminate は選ばれないのでだめである。このほか、死んだタスクに向かって entry 呼び出しをすると例外が発生することを利用して、ユーザータスクに対して資源管理タスクが生存を確認するための entry 呼び出しをすることも考えられるが、第4節で述べたように、ユーザータスクの型は不定であるから一般には不可能である。結局、現在の Ada では time out を使って、ある時間以上 RELEASE を出さないタスクは死んだものと見なすしか手がないと思われる。

この問題を解決するには、タスクが死んだ時にその事実を他のタスクに知らせるような機構が必要である。しかも、busy wait を避けるためには、タスクの生死の情報を変数に入れておくような方法では不十分で、割り込みのような形で積極的に伝えてくれることが望ましい。これを満足する最も簡単な方法は、terminate が選ばれた時に任意の動作をできるようにすることである。こうすれば、該当タスクの内側に terminate を含むタスクを宣言し、その terminate が選ばれた時に entry を呼び出すようにすることで、タスクが死んだことを外界に伝えることができる。現在の terminate は、必要とされるタスクが早めに死ぬことがないようにきわめて消極的な方針をとっているため、プログラマーの手を縛っている感が否めない。タスクが生き続けるべきかどうかの判断はプログラマーの責任とする方がよいのではないかと思われる。

7 まとめ

Ada における並行処理の問題点を PV-chunk と呼ばれる例題を通して検討した。そして、メッセージを仲介するタスクを使った解がこの問題の記述として最も優れていることを示した。この解から得られる Ada における並行処理プログラミングの指針は次のとおりである。

1 非決定的な処理は複数のタスクに分割して個々のタスクを簡単にするように努める。
2 ランデブーによる交信の機会を必要に応じて増やす。
3 タスク自身の名前を他タスクに伝えるため、自分自身の access 値を使う。

このほか、異常な事態に対処できるようにするにはどうしたら良いかを論じ、タスクの死の検出などの点で現在の Ada の機能に不十分な部分があることを示した。

参考文献

(1) C. A. R. Hoare, Communicating sequential processes, Communications of the ACM, Vol. 21, No. 8, pp. 666-677 (1978).
(2) J. Ichbiah, et. al., Rationale for the design of the Ada programming language, Sigplan Notices, Vol. 14, No. 6, Part B (1979).
(3) U. S. Department of Defense, Reference manual for the Ada programming language, 1980.
(4) J. Welsh, A. Lister, A comparative study of task communication in Ada, Software - Practice and Experience, Vol. 11, pp. 257-290 (1981).