Continuation based C 言語による OS システムコールの意味記述

宮國 渡[†] 河野真治[††]

現在の OS システムコールは、厳密あるいは曖昧な、日常言語あるいは形式的記述によって定義される。本稿では、継続を基本とする言語 Continuation based C を用いて、OS システムコールの意味記述を行い、その有効性を検証する。

A description of the semantics of OS system call by Continuation based C language

Wataru Miyaguni[†] and Shinji Kono[††]

A present OS system call is defined by a daily language or a formal description that is strict or vague. In this paper, CbC language describes a description of the semantics of OS system call and this verifies the validity.

1. はじめに

並列分散システムやマルチスレッドプログラムは、本質的に非決定的な動作をする。非決定的な動作の詳細は、`write()` などの OS のシステムコールと通信に密接に依存している。そのため、gdb などのデバッガを用いたデバッグは非常に困難である。

通常、システムコールのセマンティクスは自然言語で記述されている。そのため、例えば、図 1 を見ても、`write()` がいつ待ち状態に入るかを、形式的に調べることは難しい。

> Write() attempts to write nbytes of data to the object referenced by the descriptor d from the buffer pointed to by buf. Writev() performs the same action, but gathers the output data from the iovcnt buffers specified by the members of the iov array: iov[0], iov[1], ..., iov[iovcnt-1]. Pwrite() performs the same function, but writes to the specified position in the file without modifying the file pointer.
>
> .....

図 1 write() manual の一部

このような、厳密あるいは曖昧な、日常言語あるいは形式的記述によって定義されるセマンティクスに従って、プログラミングを行い、それに従ってプログラマはデバッグなどを行う必要がある。

本論文では、Continuation based C (CbC) という継続を基本とする言語を用いて、OS システムコールの意味記述を行い、その有効性を示す。

CbC が状態遷移記述と相性の良い言語であることに着目し、操作的意味論を用いたプログラム意味記述を行う。

CbC による意味記述プログラムを用いることによって、非決定的な動作をするプログラムの、決定的な動作をするシミュレータを作成でき、それを用いた、デバッグや検証を行うことが可能となる。

2. Continuation based C (CbC)

Continuation based C (以下 CbC) は河野真治により提案されているアセンブラよりも上位で C よりも下位な記述言語である[2)]。

2.1 CbC の要求仕様

CbC は、以下のような要求仕様に従って設計されている。

- ハードウェアとスタックマシンの中間言語
  インタプリタ記述やコンパイラターゲットとして優れていること。アーキテクチャ依存性が少ないこと。また、アーキテクチャ依存性をモデル化できること。
- C 言語よりも下位の言語
  アセンブラよりも汎用性と記述性に優れ、C 言語と互換性があること。C を CbC にコンパイルでき、ハンドコンパイルの結果を同値なコードに変

† 琉球大学理工学研究科情報工学専攻
Interdisciplinary Infomation Engineering, Graduate School of Engineering and Science, University of the Ryukyus.
†† 琉球大学工学部情報工学科
Infomation Engineering, University of the Ryukyus.

換できる。
- 明確な実行モデル
  C++やPrologのような複雑な実行モデルは好ましくなく、ハードウェアに実行順序の変更を許す範囲を広くする。
- 状態遷移を直接記述できる
  Yaccのような表駆動やCのような巨大なswitch文ではなく、直接に状態遷移が実行できる。
- Threadを実行モデルに内蔵できる。
  並列処理記述法ではなく、状態遷移として実現できる。
- クリティカルパスの最適化
  全体を散漫に最適化するコンパイラではなく、クリティカルパスを見つけ出して最適化できる。

これらの仕様は、ハードウェア記述とソフトウェア記述の両方を同時に行いつつ、C言語よりも精密な実行記述を可能にするためのものである。また、CbCはプログラム変換やコンパイラターゲットとして使うことを意識している。状態遷移記述のみでは制御構造は静的なものになってしまう。CbCでは、状態遷移記述に適した言語を作ることを考え、スタックマシンを避けてContinuation(継続)が導入されている。

2.2 軽量継続

Schemeや C++、Java、あるいは、Cも、大域脱出という形で継続を導入している。これらの言語では、継続は、必ず環境(入れ子になった局所変数を格納するスタック)のセーブを伴う。しかし、CbCでは関数呼び出しが存在しないために環境は存在しない。継続専用コード内部の局所変数は存在するが、それのネストは起こらない。そのため、環境抜きの継続を使用することができる。これは、基本的には、引数付きの直接または間接goto命令である。これを軽量継続(light-weight continuation)という。

CbCはcode-segmentを引数付きgoto()で接続することで継続を実現する。code-segmentはキーワードcodeを用いることで関数のように定義される。引数部分はinterfaceと呼ぶ。code-segmentからの脱出は引数付きgoto()である。よって、CbCのプログラムは複数のcode-segmentがgoto()で接続されたものになる(図2)。

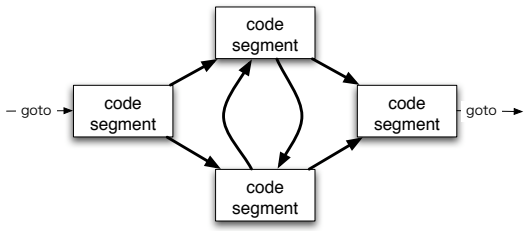

図2 CbCプログラムの構成

2.3 C with Continuation

C言語に対して、code segmenttoと引数付きgoto文を追加するためには、Cのサブルー チンへ戻るための環境付き継続を導入する必要がある。環境付き継続を導入した言語はC with Continuation(以下CwC)と呼ぶ。CwCはC言語の上位言語である(図3)。CbCは、CwCの仕様の一部に制限されたものとみなすことができるので、CwCをCbCとして使うことが可能である。

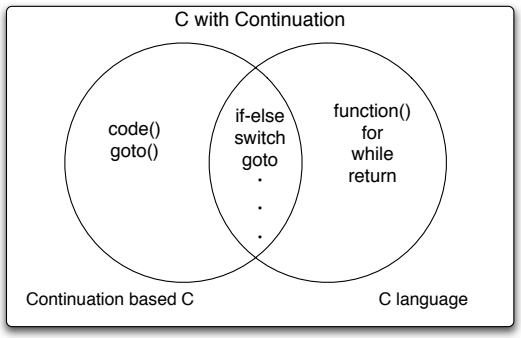

図3 C with Continuation

3. プログラム意味記述

プログラム意味記述(意味論)にはいくつかの手法があり、以下の3種類に大別される。

- 表示的意味論
- 操作的意味論
- 公理的意味論

その中で、操作的意味論(Operational Semantics)[5]とは、抽象的な計算機を定義し、プログラミング言語の意味を抽象的計算機の動作(状態遷移)として記述する手法である。

3.1 CbCによる操作的意味記述

3.1.1 オートマトン

オートマトンとは、外からの入力に対して内部の状態に応じた処理を行い、その結果を出力するシステムである。オートマトンは、複数の状態で構成されており、それぞれの状態は入力に対してどうのような処理を行うかが定義されている(図4)。

3.1.2 CbCによるオートマトン記述

CbCのcode segment、interface、引数付きgoto()は、それぞれ、オートマトンの状態遷移と入力および出力に対応している(図5)。

これより、CbCはオートマトンを記述するのに適していると言える。よって、CbCは操作的意味論を用いた意味記述を行うことができる。

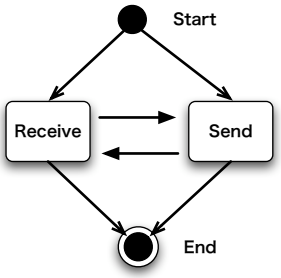

図 4 オートマトンの例

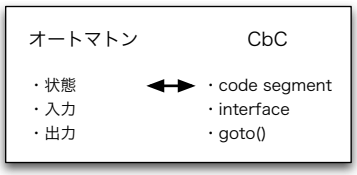

図 5 オートマトンと CbC の対応

4. CbC を用いたシステムコールの意味記述

今回は、CbC を用いて `write()`、`read()`、`select()`、`flock()` の 4 つのシステムコールの意味記述を行った。これらの引数には、通常の引数と、実行後あるいは実行と同時に継続する code segment の `next()`、next に対応する interface である `next_intf` を持つ。

4.1 待ち状態の表現

システムコールが待ち状態に入るとき、そのプロセスは scheduler に制御を渡す。その後、scheduler によって再びそのプロセスに制御が戻り、処理を再開する。

CbC では、scheduler への遷移を以下のように記述できる。

```
goto scheduler();
```

C などの軽量継続を持たない言語では、スケジューラを直接記述することはできない。例えば、ABCL/1[6] のように、プロシジャーを分割し、その分割したプロシジャーを順に呼び出すような手法が必要となる。CbC では、この分割が code segment 単位で前もって実現されているので、一対一の記述が可能になっている。

4.2 write()

`write()` 内部の状態遷移図を図 6 `write()` の意味記述プログラムである `cbc_write()` を以下に示す。

```
code
cbc_write(unsigned int fd, const char *buf,
          int count,
          code (*next)(),void *next_intf)
{
    struct file *file;
    int ret = -EBADF;
    file = fget(fd);
    if (!file) {
      goto cbc_vfs_write(file, buf, count,
                         next,next_intf);
    } else {
      goto next(ret, next_interface);
    }
}

code
cbc_vfs_write(struct file *file,
              const char *buf, int count,
              code (*next)(),void *next_intf)
{
    int ret;

    ret = rw_verify_area(WRITE,file,count);
    if (ret < 0) { // 待ち状態
      goto scheduler();
    } else {
      count = ret;
      ret = put(file, buf, count);
    }
    goto next(ret, next_intf);
}
```

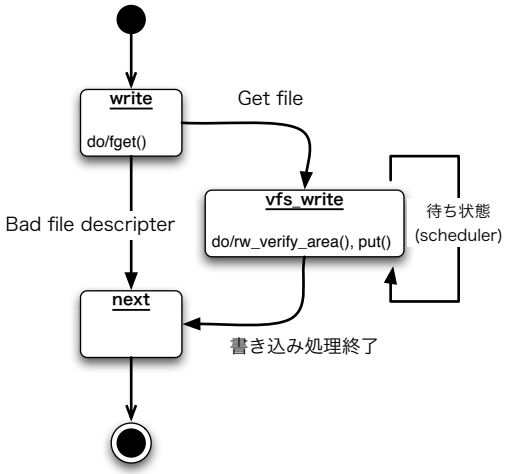

図 6 write() の状態遷移

ファイル構造体とは、オープンされたファイルに関する構造体である。現在のファイルの状態 (ロックの有無など) や、データが書き込まれた位置 (`file->buf_pos`) などを保持する。`fget()` でファイル構造体が取得できない場合 (fd が無効) は EBADF (Bad file descriptor) を返す。

`rw_verify_area()` で書き込む領域のロックを行い、実際に書き込めるサイズを返す。このとき、すでに flie が他プロセスにロックされている場合、false(-1) を返す。`rw_verify_area()` が false を返した場合、このプログラムは待ち状態に入らなければならないため、`scheduler()` に goto する。`scheduler()` によって制

御が戻ってきたとき、cbc_vfs_write() に goto するので、再び rw_verify_area() を行う。この繰り返しによって待ち状態を表現している。

put() で file に buf を書き込み、実際に書き込んだサイズを返す。このとき、エラーが起きれば、対応するエラー値を返す (書き込むスペースがない：ENOSPC など)。

code segment は return しないので、cbc_write() の return は、cbc_write() の goto に置き換わり、戻り値は、次の code segment の interface に追加することで受ける。

```
// C
ret = write(fd, buf, size);

// CbC
code cbc_write(fd, buf, size, next) {
    ....
    goto next(ret);
}
```

4.3 read()

read() の意味記述である cbc_read() を以下に示す。

```
code
cbc_read(unsigned int fd,
         const char *buf, int count,
         code (*next)(),void *next_intf)
{
    struct file *file;
    int ret = -EBADF;

    file = fget(fd);
    if (!file) {
      goto cbc_vfs_read(file, buf, count,
                        next,next_intf);
    } else {
      goto next(ret, next_interface);
    }
}

code
cbc_vfs_read(struct file *file,
             const char *buf, int count,
             code (*next)(),void *next_intf)
{
    int ret;

    ret = rw_verify_area(READ,file,count);
    if (ret < 0) { // 待ち状態
      goto scheduler();
    } else {
      count = ret;
      ret = get(file, buf, count);
    }
    goto next(ret, next_intf);
}
```

rw_verify_area() で読み込む領域のロックを行い、実際に読み込めるサイズを返す。このとき、すでに flie がロックされている場合、false を返す。

get() で file にあるデータを buf に書き込む。このとき、エラーが起きれば、対応するエラー値を返す (buf のアドレスが適切でない：EFAULT など)。

4.4 select()

select() の意味記述である cbc_select() を以下に示す。

```
code
cbc_select(int nfds, fd_set readfds,
           code (*next)(), void *next_intf)
{
    goto do_select(nfds,readfds,0,0);
}

code
do_select(int nfds,fd_set inp,int n,int retval,
      code (*next)(), void *n_i)
{
  struct file *file;

  file = fget(n);
  if (file) {
    if (file->buf_pos > 0) {
      retval++;
      FD_SET(n, &inp);
    } else {
      FD_CLR(n, &inp);
    }
  } else {
    FD_CLR(n, &inp);
  }

  if (++n > nfds)
    goto select_finish(retval,next,n_i);
  else
    goto do_select(nfds,inp,n,retval,next,n_i);
}

code
select_finish(int ret,code (*next)(),void *n_i)
{
    if (ret != 0) { // データ有り
        goto next(ret, n_i);
    } else {
        goto scheduler();
    }
}
```

今回は、読み込み可能なファイルディスクリプタ集合だけを監視し、タイムアウト機構は組み込んでいない。

do_select() で、nfds までのファイルディスクリプタを調べる。file->buf_pos を見て、読み込み可能なデータがあれば readfds に対して FD_SET(n)、読み込み可能なデータが無い場合は FD_CLR(n) を行う。全て調べ終わった時点で、読み込み可能なデータがあるファイルディスクリプタの数 retval が 1 以上であれば next() に継続する。もし一つもない場合は、データを受け取るまで待ち状態に入る。今回は、タイムアウト機構がないので、すぐに scheduler に goto する。

4.5 flock()

flock() の意味記述である cbc_flock() を以下に示す。

```
code cbc_flock(unsigned int fd, unsigned int cmd
               code (*next)(), void *next_intf)
{
    struct file *file;
    int ret = -EBADF;

    file = fget(fd);
    if (!file)
        goto out;

    // can_sleep == 1 ? block : non block
    can_sleep = !(cmd & LOCK_NB);
    cmd &= ~LOCK_NB;

    ret = -EINVAL;
    if ((cmd & LOCK_SH) && (cmd & LOCK_EX))
        goto out;

    ret = file_lock(file, cmd);
    if (ret < 0) {
        if (!can_sleep) {
            ret = -EWOULDBLOCK;
        } else {
            goto scheduler();
        }
    }

out:
    goto next(ret, next_intf);
}
```

一つのファイルに共有ロックと排他ロックを同時に設定することはできないので、その場合は EINVAL を返す。

file_lock() でロックを行う。もし file がすでに他プロセスからロックされている場合、false を返す。このとき、cmd に LOCK_NB が設定されている場合、ブロックされない (待ち状態に入らない) ので EWOULDBLOCK を戻り値に設定して継続する。LOCK_NB が設定されていなければ、ブロックされる (待ち状態に入る) ので、scheduler に goto する。

5. 意味記述プログラムのシミュレータへの組み込み

作成した意味記述プログラムは、CbC で記述されたシミュレーションプログラムのシステムコールとして、大きな変換を必要とせずに、使うことができる。CbC を用いた分散プログラムのシミュレータ[3] では、本来、非決定的である分散プログラムを、決定的な動作をするプログラムとして設計とデバッグを行うことができる。その際、意味記述プログラムを使うことによって、write や read などのシステムコール内部に対しても、gdb などによりデバッグを行うことができる。

ここでは、cbc_write() をシミュレータ記述に変換した記述を以下に示す。

```
struct write_interface {
    unsigned int fd;
    struct file *file;
    const char *buf;
    int count;
    code (*next)();
    void *next_intf;
};

code
cbc_write(TaskPtr task,write_interface *w)
{
    struct file *file;
    int ret = -EBADF;

    file = fget(fd);
    if (!file) {
        w->file = file;
        goto cbc_vfs_write(task, w);
    } else {
        w->next_intf->ret = ret;
        task->exec = next;
        task->intf = w->next_intf;
        goto scheduler(task);
    }
}

code
cbc_vfs_write(TaskPtr task,write_interface *w)
{
    int ret;

    ret = rw_verify_area(WRITE,file,count);
    if (ret < 0) { // 待ち状態
        goto scheduler(task);
    } else {
        count = ret;
        ret = put(w->file, w->buf, w->count);
    }
    w->next_intf->ret = ret;
    task->exec = next;
    task->intf = w->next_intf;
    goto scheduler(task);
}
```

スケジューラは以下のようになっている。

```
struct task {
  code (*exec)();
  void *intf;
};

code scheduler(struct task *old_task)
{
  struct task *cur_task;

  cur_task = get_next_task(cur_task);
  goto next->exec(cur_task,cur_task->intf);
}
```

exec が、タスクが継続する code segment で、intf が exec の interface である。

cbc_vfs_write() で待ち状態になったとき、task->exec に cbc_vfs_write() が設定されているので、scheduler によって再びこのタスクに制御が戻ってきたときは、cbc_vfs_write() から再開する。

シミュレータ記述の cbc_write() を実際に使用した記述を以下に示す。

```
code send(TaskPtr task, packet *pkt)
{
    /* set interface */
    struct write_interface *w;
    w->fd = pkt->desination;
    w->buf = pkt;
    w->count = sizeof(packet);
    w->next = send_finish;
    w->next_intf = interface;

    /* set task */
    task->exec = cbc_write;
    task->intf = w;

    goto scheduler(task);
}

code send_finish(TaskPtr task, interface *i)
{
    printf("write() return = %d\n", i->ret);
    goto hoge(task);
}
```

6. シミュレータの記述レベル

CbC を用いたシミュレータ記述には、いくつかの記述レベルがある。

高いレベルでは、今回作成した cbc_write() のような、待ち合わせを導入した、シミュレータとしての記述。そこから低いレベルになると、他プロセスとの干渉も考慮した記述。最終的には OS の記述となる。

高レベル記述では、シミュレータ記述となり、処理速度は速いが、精度が低くなる。逆に低レベル記述では、 OS そのものと言え、精度は高いが、処理速度は遅くなる。

このように、記述レベルによってトレードオフが起こるため、デバッグ環境に合わせて、記述レベルを合わせる必要がある。

7. ま と め

本稿では、継続を基本とする言語 CbC を用いて OS システムコールの意味記述を行い、その有効性を示した。

CbC は状態遷移記述に適している言語であるため、操作的意味論を用いたプログラム意味記述が可能である。

CbC を用いた意味記述プログラムを用いて、決定的な動作をするシミュレータを作成でき、それを用いたデバッグや検証ができるようになる。

参 考 文 献

1) 金城拓実. “ 軽量継続を用いたゲームプログラムの分割と再構成の考察 ”. 琉球大学工学部情報工学科平成 17 年度学位論文, 2006
2) 河野真治. “ 継続を持つ C の下位言語によるシステム記述 ”. 日本ソフトウェア科学会第 17 回大会, 2000.
3) 河野真治, 渕田良彦, 宮國渡. “ 継続を基本とする言語 CbC による分散プログラミング ”. 日本ソフトウェア科学会第 23 回大会論文集, Sep, 2006.
4) 下地篤樹, 河野真治. “ Continuation based C プログラムの検証 ”. 情報処理学会 システムソフトウェアとオペレーティング・システム研究会 (OS), May,2006.
5) Gordon Plotkin. “ A Structural Approach to Operational Semantics ”. Technical Report. DAIMI FN-19,Aarhus University, 1981.
6) A. Yonezawa and E. Shibayama and T. Takada and Y. Honda, Modeling and Programming in an Object-Oriented Concurrent Language ABCL/1, Object-Oriented Concurrent Programming, 1987, MIT Press