# C++に基づくシミュレーション言語のためのコルーチンの実現

3Q-8

越田 一郎
東京工科大学

## 1．はじめに

筆者らは，C++に基づいた離散型シミュレーション環境の開発を行っている[1]．C++はオブジェクト指向の言語であり，シミュレーションプログラムの作成に適しているが，並行動作するプロセスの記述能力を持たない．そのため，プロセスを基本としてシミュレーションプログラムを作成する場合に不都合が生じる．この問題を解決するため，C++の処理系内部に手を加えることなくコルーチンを実現したので報告する．

## 2．なぜコルーチンが必要か

離散型シミュレーションは離散的な時間間隔で生起するイベントを処理していくことによって実行される．シミュレーションプログラムを作成するには，シミュレートするシステム内で生起するイベントを記述すれば良い．しかし，イベントを基本単位としてシステムを記述するのは難しく，人間にとっては，プロセスを基本単位とした方が理解しやすい．この場合でも，プロセスの内部状態を保持しつつ他のプロセスを実行できる機構がないと，イベント単位の記述に近い形となってしまう．プロセスを基本としてシミュレーションプログラムを作成するためには，何等かの並行動作機構が必要であり，今回はそれをコルーチンで実現した．

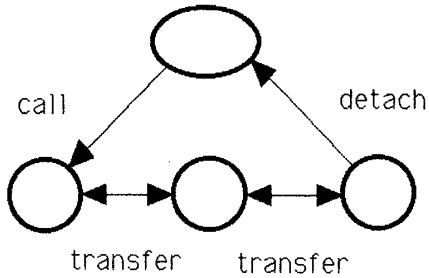


図1プリミティブの関係

## 3．コルーチンのためのプリミティブ

コルーチンのプリミティブを次に示す．これは Simula67 と同様な形式である．

resume( X )･･･他のコルーチン X に制御を移す．
detach( X )･･･コルーチンを生成したブロックに制御を戻す．
call( X )･････detach で戻ってきたコルーチンを再開する．

コルーチンとそれらを生成するブロックの関係を図1に示す．

## 4．実現方法

コルーチンはC++のクラスとして扱う．コルーチンを実現するのに必要なデータ構造を持った process クラスがあり，すべてのコルーチンは process クラスの導出クラスとして定義される．コルーチン内部で使用するローカル変数はクラス宣言の中で private な変数として宣言する．また，すべてのコルーチンは activate と呼ばれるメンバ関数を持たなければならない．コルーチンの動作はこの関数で記述する．

Process クラスには，中断したコルーチンの内部状態をセーブするための領域がある．Resume や detach を行うコルーチンは，そこに自分の現在の内部状態を保存した後で，他のコルーチンあるいは自分を生成したブロックに移る．

コルーチン間のデータのやりとりには，グローバル変数を用いてもかまわないが，コルーチンクラスのなかにデータ受け渡し用の変数と，それに対するアクセス関数を定義しておく方が汎用性のあるプログラムとなろう．

実際のインプリメントは Zortech 社の Zortech C++ を用いて実現した．このコンパイラのスタックフレームは図2のようになっているので，リターンアドレスと SI, DI レ

---

Coroutines for C++ Based Event Driven Simulation Language
Ichiro KOSHIDA
Tokyo Engineering University

ジスタの内容およびローカルな作業領域を保存する．他のコルーチンを resume する場合は，保存したデータを使ってスタックフレームを再構成した後にリターンする．

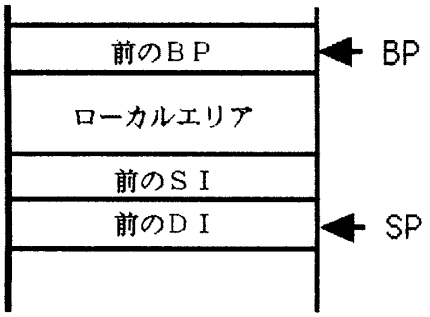


図２スタックフレーム

## ５．例

以上のようにして作成したコルーチンを使った例を図３に示す．Source コルーチンは 1 から 100 までの整数を次々と生成し，一個ずつ buffer コルーチンに渡す．buffer コルーチンは，受け取った整数をバッファに溜め，１０個ずつまとめて出力する．

## ６．まとめ

Ｃ＋＋の処理系自体に手を加えずにコルーチンを実現する方法について述べた．他の C++ 処理系に対しても同様にしてコルーチンを実現することを検討している．

## ＜参考文献＞

[1] 越田：C++ に基づくシミュレーション言語の試作，1989年電子情報通信学会春期全国大会講演論文集，1989

```
class source : public process {
    int i;
    buffer* p;
public:
    source() : () { i = 0; }

    void set_buffer( buffer* q ) { p = q; }
    void activate();
};

const int BufferSize = 10;

class buffer : public process {
    int buf[ BufferSize ];
    source* p;
    int n;
public:
    buffer() : () { }

    void put( int x ) { n = x; }
    void set_source( source* q ) { p = q; }
    void activate();
};

void source::activate()
{
    PROCESS_HEADER;

    detach();

    for( i = 1; i <= 100; i++ ) {
        p->put( i );
        resume( p );
    };

    resume( p );
    detach();
}

void buffer::activate()
{
    int i;
    PROCESS_HEADER;

    detach();
    for(;;) {
        for( i = 0; i < BufferSize; i++ ) {
            buf[ i ] = n;
            resume( p );
        }
        for ( i = 0; i < BufferSize; i++ ) {
            cout << buf[i] << " ";

        }
        cout << "¥n";
    }
}

source x;
buffer y;

main()
{
    x.set_buffer( &y );
    y.set_source( &x );

    x.activate();
    y.activate();

    call( &x );
}
```

図３プログラム例