オペレーティング・システム 33－2
（1986. 12. 12）

並行プロセス実験キット

多田　好克*　　寺田　実**

*　電気通信大学　電子情報学科
**　東京大学　　計数工学科

本稿では，Unixの一ユーザプロセス上で動く「小さなプロセス」実現法を説明する．この方法を使えば，Unixのプログラミング環境下でスケジューリングやプロセス間通信等の実験を行うことができる．

本実現法では，言語Cの一部の関数がプロセスのように振舞う．また，それらはCPU横取りによって継続的な実行を制限される．これらの仕組みは言語Cのみで記述されており，Unixのカーネルを変更する必要はない．なお，現在，この仕組みは，VAX-11 (4.2BSD), VAX-11 (Ultrix), Sunワークステーション(4.2BSD),METHEUS Lambda-710 (4.1BSD), SHARP IX-5 (System V), NEC PC-UX (System Ⅲ)等，様々のUnixシステム上で稼働している．

Primitives for Testing Environment of Concurrent Processes

Yoshikatsu TADA* and Minoru TERADA**

*　The University of Electro-Communications,
1-5-1 Chofugaoka Chofu-shi, Tokyo, 182, JAPAN
**　The University of Tokyo,
7-3-1 Hongo Bunkyo-ku, Tokyo, 113, JAPAN

This paper presents a simple method of running small processes on a user process of the Unix. Using this method, we can implement testing environment of concurrent processes, where we can evaluate, for example, scheduling algorithms or mechanisms of interprocess communication, and nevertheless all programming environments of the Unix are still available.

On our implementation, some C functions run concurrently like processes and the preemption takes place after running certain period. Entire implementation is described using C language without the kernel code modification. This method is now implemented on various Unix systems including VAX-11 (4.2BSD), VAX-11 (Ultrix), Sun Workstation (4.2BSD), METHEUS Lambda-710 (4.1BSD), SHARP IX-5 (System V) and NEC PC-UX (System Ⅲ).

1. はじめに

近年，Modula-2[1]やOccam[2]，Concurrent Euclid[3]等の並行プログラミング言語を利用したシステム記述法が盛んに議論され始めている．また，筆者らも昨年のコンピュータシステムシンポジウムにおいて，２種類の並行プロセスを用いたシステム記述法を提案した[4]．現在，我々はその方法に従って各種実験を試みているが，これらの実験環境では実験可能範囲と実験効率との間に以下に示すようなトレードオフが成立する．

Ⅰ．　実験を既存のオペレーティングシステム上で行う場合．

メモリ，I/O装置，プロセス等は当該オペレーティングシステムの管理下におかれる．したがって，メモリの割当てやI/O装置の制御，プロセス間通信等の実験は制限を受ける．

Ⅱ．　実験を裸の計算機上で行う場合．

オペレーティングシステムが存在しないため，実験ソフトウェアの開発や管理は他計算機上で，または，同一計算機のシステムを切り換えて行う．この場合，計算機間でのプログラムの移動やシステムの切換えに余分な時間を必要とする．

また，これらⅠ，Ⅱそれぞれの長所を利用するための方法として仮想機械[5]も提案されているが，仮想機械の実現にはハードウェア，ソフトウェア両面にわたる支援が必要となり現実的ではない．

筆者らは既存のUnixオペレーティングシステム［注］を利用して上述の実験を行っているが，当初，スケジューリング，プロセス間通信等の実験は行うことができなかった．そこで言語Ｃの関数をプロセスのように振舞わせる仕組みを開発し，現在は，Unixの１ユーザプロセス内で複数の「小さなプロセス」を動かしている．この方法を利用することによりUnixの開発環境を利用しながら並行プロセスに関する様々な実験を行うことが可能となった．

筆者らの開発した方法は，

i)　Unixカーネルの変更を必要としない；
ii)　システム依存性が低い；
iii)　すべての仕組みを言語Ｃで記述できる；
iv)　仕組みの基本部分は数十行と小さい；

といった特長を備えている．また，現在では，VAX-11(4.2 BSD)，VAX-11(Ultrix)，Sunワークステーション(4.2BSD)，METHEUS Lambda-710(4.1BSD)，SHARP IX-5(SystemⅤ)，NEC PC-UX(SystemⅢ)の各Unixシステム上で実際に稼働しており，各種実験にも利用されている．

本稿の目的は「小さなプロセス」の具体的な実現法を示すことにある．以下，第２章では並行プロセス実現法の導入として，まず，Unix上での言語Ｃによるコルーチン実現法を論じる．続く第３章では並行プロセスの実現法を示し，その正当性を議論する．第４章では実現した仕組みにおけるユーザ関数，ライブラリ関数，各種変数等の意味を考える．また，最後の第５章では本稿で提示した仕組みの活用法を考え，今後の展望を述べる．付録には「小さなプロセス」を使った簡単なプログラム例を示した．

なお，本稿ではシステム開発の歴史的事情に合わせ，主にIX-5における実現法を議論する．その他のシステムにおける実現法は近々発表する予定なのでそちらを参照していただきたい．

［注］　Unix：　米国ではAT&Tの商標である．

2. コルーチンの実現法

本章ではＣの関数をUnix上でコルーチンとして実行する方法を示す．すなわち，

```
void    resume(func);
int     func;
```

という関数を導入し，コルーチンとして定義された関数内からこの関数を呼び出すことによりfunc番目のコルーチンの実行を再開できるようにする．（resume()を呼び出した関数は，他のコルーチンからresume()によって指定されるまで，その実行を停止する．）

コルーチンを実現するためには，実行中の関数のコンテクストを保存・回復する必要がある．ここでいうコルーチンのコンテクストには以下のものが考えられる．

i)　一時レジスタ以外のレジスタの値
ii)　コルーチンに固有の変数の値
iii)　制御の履歴

なお，i)にはPC,SP,FPとレジスタ変数用レジスタが含まれる．ちなみに，IX-5ではCPUに68000が使われており，PCとa2からa7,d2からd7の13個のレジスタがこれに該当する．ii)にはＣのauto変数が該当する．また，iii)にはコルーチンから呼び出された関数の各フレームが該当する．

これらのコンテクストの内，ii)とiii)に関する情報はスタックに積まれている．したがって，ii)，iii)の保存・回復は，

a)　コルーチンごとにスタックを確保する；
b)　スタックポインタの保存・回復をする；

というふたつの処理を行えば良い．スタックポインタの保存・回復はi)の処理に含まれているので，結局，コルーチンは，

Ⅰ)　resume()実行時にi)に示したレジスタの保存・回復を行う；
Ⅱ)　コルーチン生成時にコルーチンごとのスタック領域を確保する；

ことにより実現できる．

2.1. resume()の実現法

Unix上の言語Ｃには，大域ジャンプに使用するふたつの関数，

```
int     setjmp(env);
jmp_buf env;
```

と

```
        void    longjmp(env,val);
        jmp_buf env;
        int     val;
```

とが用意されている（Unixマニュアル[6]のsetjmp(3)を参照のこと）．

jmp_bufは取込みファイルsetjmp.h内で定義されたデータ型で，大域ジャンプに必要なレジスタ情報を格納するためのバッファ型を定めている．IX-5のjmp_bufはint型の13個の配列で，ここに，PC,a2からa7,d2からd7の各レジスタ値が格納される．

関数setjmp()を実行すると，バッファenvにその時点でのレジスタ値が保存される．この時，関数setjmp()は値0を返す．他方，longjmp()を実行すると，バッファenvに保存されていたレジスタ値が各レジスタに回復され，その結果として最後にenvを指定したsetjmp()の位置からプログラムの実行が再開される．この時，再開された関数setjmp()は値valを返したように振舞う．

これらの関数を使うと，resume()は以下のように書ける．

```
jmp_buf p[NFUNC];
        /* コルーチンごとのレジスタ保存場所 */
int     curfunc;        /* 実行中のコルーチン番号 */

void    resume(i)
        int     i;
{
        if (setjmp(p[curfunc]))
                /* ここから実行が再開される */ ;
        else
                longjmp(p[curfunc = i],1);
}
```

ただし，NFUNCはコルーチンの総数を示す記号定数である．

resume()の動作は次のようになる．

まず，resume()を呼び出したコルーチン（curfuncにその番号が入っている）は，if内のsetjmp()によってコンテクストを保存する．setjmp()の返す値は0であるから，else部のlongjmp()が実行される．この時，curfuncの値は実行を再開するコルーチンの番号iによって置き換えられる．longjmp()実行の結果，制御は（過去に実行された）if内のsetjmp()に移り，resume()の実行が終了して目的のコルーチンが再開される．

スタック領域がコルーチンごとに確保されている場合には，それぞれのコルーチンがresume()を呼び出すたびに制御が移行する．

2.2. スタック領域の設定法

スタック領域の確保を言語Cのみで記述するには，多少のトリックが必要である．ここでは，まず，アセンブラを併用した方法を述べ，その後，改めてCしか使わない方法を示すことにする．

アセンブラを併用することによりスタックポインタへの値設定が簡単になる．IX-5にはCプログラム内からアセンブラを出力する擬関数asm()があるので，これを利用すればスタックの確保，及び，コルーチンの初期化は以下のように記述できる．

```
main()
{
        if (setjmp(p[0])) {
                asm("   sub.l   &1024,%sp");
                func0();
        }
        if (setjmp(p[1])) {
                asm("   sub.l   &2048,%sp");
                func1();
        }
        /* 以下，使用するコルーチンの
                        個数だけ繰り返す */
        curfunc = 0;
        longjmp(p[curfunc],1);
}
```

ここで，func0(),func1()等はCの関数名で，それぞれがコルーチンとして実行される．各コルーチンのスタックはmain()の実行に使われているスタック領域の上方にそれぞれ1Kbyteの大きさで確保され，各コルーチンはこのスタック上で実行される．

スタック領域を動的に配置できないシステムにおいては，スタック領域に対応する実メモリをあらかじめ配置しておく必要がある．IX-5では，関数の実行前に十分なスタック領域が有るかどうかを調べるので，その部分のプログラムは省略した．

Cのみを使ってスタックを確保する場合は，下請け関数setcoroutine()を使って以下のように記述できる．

```
#define STKSIZE 1024
void    setcoroutine(fn,func,i)
        int     fn,i;
        void    (*func)();
{
        void    (*funcsave)();
        char    stack[STKSIZE];
        if (i == 0) {
                funcsave = func;
                if (setjmp(p[fn]))
                        (*funcsave)();
        } else
                setcoroutine(fn,func,--i);
}
```

```
main()
{
        setcoroutine(0,func0,0);
        setcoroutine(1,func1,1);
        /* 以下，使用する
                コルーチンの個数だけ繰り返す */
        curfunc = 0;
        longjmp(p[curfunc],1);
}
```

この方法では，setcoroutine()を再帰的に呼び出すことにより，それぞれのコルーチンに固有のスタックを割り付ける．なお，setcoroutine()内でコルーチンの実行開始アドレスfuncの値をsavefuncに代入しているのは，仮引数funcの値が再帰によって失われるのを防ぐためである．

3. 並行プロセスの実現法

前述のコルーチンを並行プロセスに仕立てるには，CPUを横取りする仕組み［注］が必要である．また，プロセスの実行を任意の時点で中断・再開するためには，コルーチンのコンテクストに一時レジスタの値をも含めなければならない．以下の各節では，これらの実現法とその正当性を議論する．

3.1. CPU横取りの実現法

CPUの横取りには，シグナルを利用した（Unixマニュアルのsignal(2)を参照のこと）．実現に際してはライブラリ関数alarm()を流用する方法とUnixの別プロセスからkill()を使ってシグナルを送る方法とが考えられる．ここでは後者，すなわち，kill()を使ってユーザ用のシグナルSIGUSR1を送出する方法を示す．

シグナルを受けると，システムはCPU横取りのための関数preenpt()を実行する．この設定は初期化関数内で行う．また，SystemⅤではシグナルを受け取るたびに，シグナル処理関数の設定を解除するので，preempt()内でも再設定を行う．以下に，シグナル生成，初期化プログラムの概要を示す．

```
main()
{
        int     pid;
        if (pid = fork())
                /* 一定の間隔でシグナルを発生する */
                while (1) {
                        sleep(1);
                        kill(pid,SIGUSR1);
                }
        else {
                /* 初期化 */
                if (setjmp(p[0])) {
                        asm("   sub.l   &1024,%sp");
                        signal(SIGUSR1,preempt);
                        proc0();
                }
                /* 以下，使用するプロセスの
                                個数だけ繰り返す */
                curproc = 0;
                longjmp(p[curproc],1);
        }
}
```

proc0()等はCの関数名で，それぞれがプロセスとして実行される．また，スタックの確保にはアセンブラを併用しているが，コルーチンの場合と同様にCのみでも記述できる．

CPU横取りのための関数preempt()は以下のようになる．

```
void    preempt()
{
        if (setjmp(p[curproc]))
                signal(SIGUSR1,preempt);
        else {
                /* スケジューリングの一例 */
                curproc = ++curproc % NPROC;
                longjmp(p[curproc],1);
        }
}
```

preempt()はコルーチンの場合のresume()と同じような格好をしている．シグナル処理関数preempt()を再設定することと，実行を再開する関数の決定法とが違うだけである．

ところで，コルーチンの場合，制御の切替えはresume()を呼び出した時点，すなわち，言語Cの式のレベルでしか行われなかった．しかし，preempt()は任意の時点で制御を切り替えるので，言語Cの一時レジスタの値を保存・回復しなければ正しく作動しない．次節では，この一時レジスタの扱い方について議論する．

3.2. 一時レジスタの扱い

結論から先に言えば，本方式では一時レジスタの保存・回復を陽に実行する必要がない．というのは，Unixカーネルのシグナル処理時に必要な処理が行われるからである．以下では，Unixのシグナル処理手順を追ってその正当性を示す．なお，説明の都合上，proc0()からproc1()に制御が切り替わる場面を想定するが，他の場合も処理手順は同じである．

［注］ preemptionのこと．ここでは，タイマ割込みによるCPU横取りを想定している．

1) proc0()の実行中：

proc0()はproc0()に固有のスタックを使って実行される．この時，非同期のシグナル割込みが発生すると，本システムはやがてUnixカーネル実行中の状態になる．

2) Unixカーネルの実行中：

Unixはカーネル状態になった時点で，本システムのコンテクストをカーネルスタックに退避する．この中にはproc0()で使用中の一時レジスタの値も含まれている．カーネル実行の最後にはシグナルの処理が行われる．

3) シグナルの処理の準備：

シグナル処理関数preempt()はユーザ状態で実行される．また，その実行が終了すると（カーネル状態を経ることなしに）本システムの実行を再開する．そのため，カーネルはシグナル処理終了時に，カーネルスタックにある一時レジスタの値をユーザスタックに複写する．この時のユーザスタックはproc0()のスタックである．

4) preempt()の実行中：

preempt()の前半部分はproc0()のスタックを使って実行される．しかし，プロセス切替え後の後半部分はproc1()のスタックが使用される．したがってpreempt()の実行終了後は，proc1()のスタックにある一時レジスタの値を本システムの一時レジスタに回復する．

5) proc1()の実行中：

以上の結果，proc1()の実行が再開された時には，過去，proc1()の実行中に行われたCPU横取りの時の一時レジスタの値が回復されることになる．

つまり，本システムではプロセス切替え時の一時レジスタの保存・回復を，Unixカーネルが暗黙裏に実行しているわけである．したがって，この節の冒頭に述べたように，本方式では一時レジスタの保存・回復を陽に実行する必要はない．

4. 本システムにおける関数・変数の意味

本方式で実現した並行プロセス実行環境とUnixの実行環境との間には様々な類比が成り立つ．本章では本システムの動作をより明確にするために各関数，各変数の意味について議論する．

まず，関数について論じる．

Unixのカーネルには，本システムのいくつかの関数群が対応している．特に上述のmain()とpreempt()とはプロセスの初期化とスケジューリングとを行うカーネルプログラムと考えられる．前章ではスケジューリングアルゴリズムの一例として最も単純な方法を示したが，preempt()内で各種のアルゴリズムを実現することができる．また，これら以外の関数を追加して本システムの拡張を図ることも可能である．

Unixのシステムコールにはある種の関数呼出しが対応する．また，同様にライブラリ関数にもある種の関数を対応づけることができる．ただし，Unixの場合と違い本システムではひとつのライブラリ関数を複数のプロセスで共用できる．なお，後述する理由により本システムではprintf()等のUnixライブラリ関数を利用できないが，それらも類似のライブラリ関数を用意することによって実現可能となる．

初期化プログラムによって指定された関数はプロセスとして作動する．本実現法ではプロセスの個数を静的に定めておく必要があるが，動的なプロセス生成ができるようにカーネル用関数を追加することは容易である．

他の関数は各プロセスから呼び出されるユーザ関数と考えられる．ただし，上述のライブラリ関数とユーザ関数との間に特別な違いはない．ライブラリ関数同様，複数のプロセスがひとつのユーザ関数を共用することもできる．

カーネル用関数，システムコール用関数，ライブラリ関数，プロセス用関数，ユーザ関数の間には実質的な違いはない．カーネル用関数が使用している変数をユーザ関数から変更することも可能である．また，各プロセスのスタック領域は固定長であり，スタック溢れの検出も行っていない．したがって，システムの動作はプログラムで保証しなければならない．

次に，変数について議論する．

各プロセスは，プロセスに固有のスタック領域を持っている．したがって，各関数の仮引数やauto変数等はプロセスごとにその実体が作られ，プロセス固有の変数となる．また，レジスタ変数もプロセス固有の変数である．他方，外部変数や静的変数はUnix実行環境のdata,bss領域に作られ，共用変数となる．

初期のUnixにおいてはパイプ以外にプロセス間通信を実現する方法がなかったが，本システムにおいては共用変数を使ったプロセス間通信が可能である．ただし，共用変数を排他的に使用するにはなんらかの機構をカーネル関数として用意しなければならない．

Unixのライブラリ関数printf()やputchar()等は出力バッファのポインタに外部変数を使用している．したがって，本システムのプロセスからこれらの関数を直接呼び出すことはできない．同等の機能を利用するには，外部変数を使用しないシステムコールwrite()を使って新しくライブラリ関数を書く必要がある．

本稿の終りには付録として簡単なプログラム例を載せた．そのプログラムの出力にwrite()システムコールを使っているのは，今述べた理由による．

5. おわりに

以上，Unixの一ユーザプログラム上で，並行して作動する「小さなプロセス」の実現法を示した．また，その仕組みにおける各関数の役割，各変数の持つ意味を論じた．これらの仕組みを利用すれば，並行プロセスの実験がUnixシステム上で効率的に行える．

ここで示した仕組みは，「小さなプロセス」を実現するための骨組みとなる．この仕組みを利用して複雑なシステムを構築する場合には，プロセス制御の基本機構，および，各種ライブラリ関数を整備しなければならない．我々は，現在，基本入出力，ファイルシステム等のライブラリ関数を作成中である．また，それと並行して，プロセスの生成・消滅を動的に行えるような仕組みも開発している．

一方，本システムの応用として，現在，Xinuオペレーティングシステム[7]をこの仕組みの上に実現している[8]．Unixの上で動く「小さな仮想オペレーティングシステム」は，オペレーティングシステムの教育に役立つと考えている．

ここで示した仕組みを，裸の計算機上に構築することはたやすい．将来は本システムを使って実験し，その結果は裸の計算機上で実行することになるであろう．また，並行処理の記述できる言語を定義し，そのコンパイル結果を本システムのCプログラムとして出力することも試みている．今後，言語定義から裸の計算機上での実行までを通して，様々なプロセスの実験を続けていく予定である．

＊＊＊　参考文献　＊＊＊

[1] Niklaus Wirth: PRGRAMMING IN MODULA-2, 2nd Ed., Springer-Verlag, Berlin Heidelberg New York, 1983.

[2] Occamプログラミングマニュアル，啓学出版(株)，東京，1984.

[3] Holt R. C.: CONCURRENT EUCLID, THE UNIX SYSTEM, AND TUNIS, Addison-Wesley, Massachusetts, 1983.

[4] 多田好克：　個人用操作系の未来像　－－－プロセス中心の記述法－－－，情報処理学会「コンピュータ・システム」シンポジウム報告集，pp. 177-182，1985年12月．

[5] 山谷正己，秋山義博：　仮想計算機，共立出版(株)，東京，1978.

[6] UNIX Programmer's Manual, Bell Telephone Laboratories, New Jersey.

[7] Comer D.: Operating System Design: The Xinu Approach, Prentice Hall, New Jersey, 1984.

[8] 多田好克：　Virtual Xinu --- Unix上で作動する仮想オペレーティングシステム，第28回プログラミング・シンポジウム予稿集，1987（予定）．

付録：　プログラム例と実行例

並行プロセスのプログラム

```
#include        <stdio.h>
#include        <signal.h>
#include        <setjmp.h>

#define NPROC   3
jmp_buf p[NPROC];
int     curproc;

void    proc0()
{
        int     i;
        char    s[4];
        s[0] = ' ';
        s[2] = ':';
        s[3] = '0';
        while (1) {
                for (i=0; i<10; i++) {
                        s[1] = (char)(i+(int)'0');
                        write(1,s,4);
                }
        }
}

void    proc1()
{
        int     i;
        char    s[4];
        s[0] = ' ';
        s[2] = ':';
        s[3] = '1';
        while (1) {
                for (i=0; i<10; i++) {
                        s[1] = (char)(i+(int)'0');
                        write(1,s,4);
                }
        }
}
```

```
void    proc2()
{
        int     i;
        char    s[4];
        s[0] = ' ';
        s[2] = ':';
        s[3] = '2';
        while (1) {
                for (i=0; i<10; i++) {
                        s[1] = (char)(i+(int)'0');
                        write(1,s,4);
                }
        }
}

void    preempt()
{
        if (setjmp(p[curproc]))
                /* resume here */
                signal(SIGUSR1,preempt);
        else {
                /* reschdule */
                curproc = ++curproc % NPROC;
                longjmp(p[curproc],1);
        }
}

main()
{
        int     pid;

        if (pid = fork()) {
                /* interrupt timer */
                while (1) {
                        sleep(1);
                        kill(pid,SIGUSR1);
                }
        } else {
                /* process initialization */
                if (setjmp(p[0])) {
                        asm("   sub.l   &1024,%sp");
                        signal(SIGUSR1,preempt);
                        proc0();
                }
                if (setjmp(p[1])) {
                        asm("   sub.l   &2048,%sp");
                        signal(SIGUSR1,preempt);
                        proc1();
                }
                if (setjmp(p[2])) {
                        asm("   sub.l   &3072,%sp");
                        signal(SIGUSR1,preempt);
                        proc2();
                }
                curproc = 0;
                longjmp(p[curproc],1);
        }
}
```

プログラムの実行例

（下線部でプロセス切替えが起こっている）

```
0:0 1:0 2:0 3:0 4:0 5:0 6:0 7:0 8:0 9:0 0:0 1:0 2:0
3:0 4:0 5:0 6:0 7:0 8:0 9:0 0:0 1:0 2:0 3:0 4:0 5:0
6:0 7:0 8:0 9:0 0:0 1:0 2:0 3:0 4:0 5:0 6:0 0:1 1:1
2:1 3:1 4:1 5:1 6:1 7:1 8:1 9:1 0:1 1:1 2:1 3:1 4:1
5:1 6:1 7:1 8:1 9:1 0:1 1:1 2:1 3:1 4:1 5:1 6:1 7:1
8:1 9:1 0:1 1:1 2:1 3:1 4:1 5:1 0:2 1:2 2:2 3:2 4:2
5:2 6:2 7:2 8:2 9:2 0:2 1:2 2:2 3:2 4:2 5:2 6:2 7:2
8:2 9:2 0:2 1:2 2:2 3:2 4:2 5:2 6:2 7:2 8:2 9:2 0:2
1:2 2:2 3:2 4:2 7:0 8:0 9:0 0:0 1:0 2:0 3:0 4:0 5:0
6:0 7:0 8:0 9:0 0:0 1:0 2:0 3:0 4:0 5:0 6:0 7:0 8:0
9:0 0:0 1:0 2:0 3:0 4:0 5:0 6:0 7:0 8:0 9:0 0:0 1:0
2:0 3:0 4:0 5:0 6:0 7:0 8:0 9:0 0:0 1:0 2:0 6:1 7:1
8:1 9:1 0:1 1:1 2:1 3:1 4:1 5:1 6:1 7:1 8:1 9:1 0:1
```