PS2 Linux での安全なユーザレベル排他制御の実現
The Implementation of user level test-and-set on PS2 Linux

町田 浩之,　　　　篠原 孝夫
Hiroyuki Machida,　Takao Shinohara

(株)ソニー・コンピュータエンタテインメント
クリエイティブステーション本部　開発部☆
R&D Dept., Creative Station Div., Sony Computer Entertainment Inc

In the multi-thread environment like Linux, a fast user-level mutual exclusion mechanism is strongly required. But MIPS chips designed for embedded and single processor, like the Emotion Engine, have no atomic test-and-set instruction. We implemented the fast user-level mutual exclusion without invoking system-call and its costs, on the PS2 Linux. This method utilizes the memory protection facility of Operating System, to detect preemption and nullify the operation. In this paper, we present the method and its evaluation.

1　はじめに

1.1　背景

PlayStation 2 には Emotion Engine[1]と呼ばれる CPU が搭載されている. Emotion Engine は CPU コア以外に VPU(ベクタプロセッシングユニット)等を搭載している. CPU コア(EE Core または単に EE と省略する) は, MIPS 命令セットアーキテクチャのプロセッサで, 64bit MIPS Ⅲ 命令セットと一部の MIPS Ⅳ命令セットを持つ. 128bit マルチメディア命令等が追加されている反面 test and set 専用命令(ll, sc)や 64bit 乗除算命令などの一部の命令が削除されている.

Linux を EE 上に移植するにあたり, スレッド間の排他制御のために安全で効率的な test and set の実装は必要不可欠なものであった. しかし専用命令 (ll, sc)を使用しない効率のよい従来の方式には, 安全性や汎用性の面で問題があり, 効率の悪いシステムコールによる実装が多く用いられてきた. そこでオペレーティングシステムのメモリ保護機構を利用し, 専用命令なしにこれらの問題を解決できる新しい test and set の方式を開発した.

1.2　スレッド間の排他制御

Linux の様なメモリ保護機能を持ち, プリエンプティブ スケジュールを行うオペレーティングシステム(以降 OS)上では, スレッド間の相互排他を効率的に実現するため, ユーザプロセルレベルで アトミックな（不可分な）test　and　set の実装が必要となる.

test and set は, アトミックに対象となる変数の値をテストし, 新たな値をセットする手続きである[2]. 値が 0 なら 1 にセットし, もとの値を返す. この test and set に渡される変数は排他変数と呼ばれ, 値 1 をクリティカルセクション(同時に突入してはいけない部分)が占有されているという意味に用いる. test and set 手続き tst() の呼び出し例を図 1 に示す.

```
while (tst(&mutex_var)) ;

CRITICAL SECTION

mutex_var=0;
```

Figure 1. Usage of test and set

test and set がアトミックに実装されていない場合には相互排他に失敗する. 図 2 はアトミックでない test and set の例である.

```
tst_nonatomic:      // a0 is address of mutex var.
        lw        v0, (a0)   // v0 = *a0;
        li        t0, 1      // t0 = 1;
        bnez      v0, 1f     // if v0 != 0 goto 1
        nop
        sw        t0, (a0)   // *a0 = t0;
1:
        jr        ra         // return v0
        nop
      The surrounded operation must be atomic.
```

Figure 2. Non-atomic (flaw) test-and-set

☆ 執筆当時の所属. 現在は, ソニー株式会社 ネットワーク＆ソフトウェアテクノロジーセンターに在籍.

これを利用してユーザプロセス Proc. A, Proc. B は, クリティカル セクションへ進むとする. 排他変数の初期値を0とし, 図1の点線内で相手に割り込まれると, 双方とも 0 を返して 2 つが同時にクリティカルセクションへ突入してしまう場合がある. これを図3に示す.

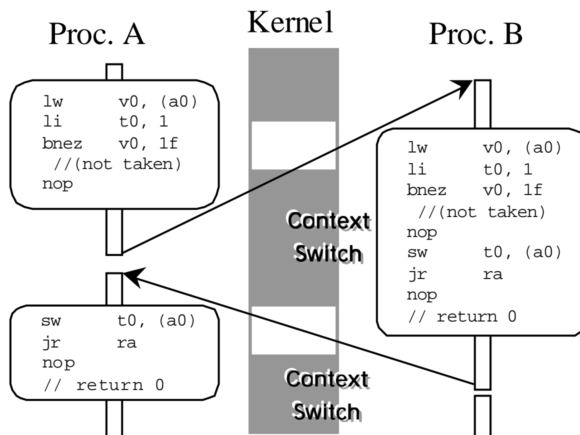


Figure 3. Senario of race condition with non-atomic test-and-set

1.2 test-and-set 専用命令

アトミックな test and set のために, 多くの場合、CPU に専用命令が用意される. MIPS 2 ISA (Instruction Set Architecture) の CPU では アトミックな動作を保証するために ll (Load Linked), sc (Store Conditional) 命令[3]が用意されている. 図 4 は, これを使った test and set の例である. CPU は, ll で ロードしたときのアドレスを覚えており, sc を実行するまでの間に他の CPU がこの近傍に書き込みを行ったか, 例外が起きた場合には, sc で書き込みを行わない.

```
tst:
0:                         // a0 is address of mutex var.
   ll     v0,  (a0)        // v0 = *a0

   li     t0,  1           // t0 = 1
   bnez   v0,  1f          // if v0 != 0 goto 1
   nop

   sc     t0,  (a0)        // (*a0 = t0, t0=1), if the
                           // surrounded operation is atomic.
                           // Otherwise t0=0
   beqz   t0, 0b           // if (!t0) goto 0
   nop
1:
   j      ra               // return v0
```

Figure 4. Test-and-set using ll and sc

1.3 専用命令不要な排他制御方式

しかし, 組込用途に設計されたシングルプロセッサ専用 MIPS 系 CPU の多くは, ll, sc 命令を持たない. 例えば EE や NEC Vr4100 や東芝 Tx3900 などが, これにあたる. そこで test and set の実装のため, 以下のような方法が用いられてきた.

1. 割り込み禁止・許可命令(di, ei)の利用
2. システムコールによる実装
3. Lamport アルゴリズム[4]の利用

残念ながら, 各々には大きな問題点がある.

1 の命令を OS のユーザレベルで許可すると, 図5の様なプログラムで一般ユーザが CPU を割り込み禁止状態で無限ループさせることができる. つまり特権なしにだれでも簡単にシステム全体を停止できてしまう. また, Linux の様な仮想記憶を使う環境下では, 割り込み禁止区間中に TLB miss や, ページフォルト等の割り込みも発生しないように実装を工夫する必要がある.

```
halt:
    di          // disable interrupt
    b    halt   // goto halt
    nop
```

Figure 5. Halting entire system using DI instraction

2 のシステムコールによる実装では, 競合しない場合でも必ずシステムコール例外が発生し, コンテキストスイッチが起きる. コンテキストスイッチが起きると, 他のユーザプロセスへ制御が移ってしまい, 大きなオーバヘッドが生じる. (図 6)

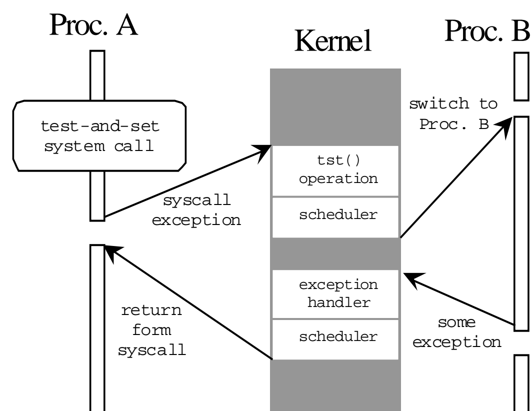


Figure 6. Overhead of tst (test-and-set) system-call

3 の Lamport のアルゴリズムでは，図 7 の様に本来の排他変数とは別に参加する全プロセスの状態を表すための変数を全てのプロセスが共有する必要があり，その確保，保護や後処理が複雑となる．具体的には以下のような問題が挙げられる．

- 異常終了時の後処理に関する問題
- b[ ] の確保に関する問題
- x, y, b[ ]の領域と排他変数の領域を分離した場合，生じる領域管理と保護の問題

```
start:
        <b[i]=true>;
        <x=i>;

        if (<y!=0>) {
                    <b[i]=false>;
                    await(y==0);
                    goto start;
        }
        <y=i>;
        if (<x!=i >) {
                    <b[i]=false>;
                    for (j=1; j<=N; j++) {
                              await(!b[j]);
                    }
                    if <y!=i> {
                              await(y==0);
                              goto start;
                    }
        }

        CRITICAL SECTION

        <y=0>;
        <b[i]=false>;

i    : own process number
x, y : shared state variables
b[i] : shared boolean variables, initaily set to false
<>   : denotes atomic operation
```

Figure 7. Lamport's Fast Mutual Exclusion Algorithm

1.4 目的

以上のように従来の方式には，安全性，効率，資源管理の点で問題がある．そこで以下を目標にユーザプロセルレベル test and set の新しい方法を考案し，実装した．

- Linux thread に必要な一般的な機能を提供する
- 競合のない場合のオーバヘッドを極力減らす
- セキュリティやシステムの安定度を低くしない
- メインテナンス性に考慮する,不用な場合には機能を簡単に切り放せる

対象は ll, sc 命令を持たないシングルプロセッサ専用 MIPS 系 CPU とする

2 設計と実装

2.1 動作原理

対象がシングル CPU のユーザレベルなので，ある操作中に他のユーザプロセスに割り込まれなければ，アトミック性を保証できる．仮に割り込まれたとしても，現在実行中の操作をなかったことにし，はじめからやり直せれば，操作のアトミック性を保証できる．ここで, 割込みの検出と操作の無効化はアトミックに行われなければならない．

割り込みの検出だけで，操作の無効化を行わない方法はうまく行かない．誤解が多いので図 8 に例を挙げ，詳しく説明する．この例では，カーネルにより プロセスが割り込まれた場合には必ず preempted が 1 にセットされるとする．

```
flaw_tst:          // a0 is address of mutex var.
       lui       t0, %hi(preempted)
       addui     t0, %lo(preempted)
       sw        zero, (t0) // preempted = 0;

       lw        v0, (a0)  // v0 = *a0;
       li        t1, 1     // t1 = 1;
       bnez      v0, 1f    // if v0 != 0 goto 1;
       nop
       sw        t1, (a0)  // *a0 = t1
       lw        t2, (t0)  // t2 = preempted
       nop
                   // if (t2!=0) goto flaw_tst
       bnez      t2,  flaw_tst
       nop
1:
       jp        ra         // return v0;
       nop

The kernel always sets non-zero value to 'preempted'
on transit to the user mode.
Due to lack of nullifing store operation, this flaw_tst()
may fail, if preempted in the surrounded portion.
```

Figure 8. Preemptition sensitive (flaw) test-and-set

ここで，競合が起きない 以下のような流れを考えてみる．

- flaw_tst( ) 呼び出し時の排他変数の内容は 0
- flaw_tst( ) の点線内で他プロセスが割り込む
- 他プロセスは排他変数を操作しない
- flaw_tst( ) 内へ復帰 (このときカーネルにより preempted が 1 にセットされる)

すると，自分自身が 排他変数を 1 に変更したにも関わらず, flaw_tst()の戻り値として 1 が返ってしまう．つまり，自分でロックを行ったにも関わらず，他プロセスによりロックが行われていると報告することになり，デットロックを生じる．このように割り込まれた場合には排他変数への書込み操作が行われない(無効化される)ことが必要となる．

本方式では, MIPS系CPUのカーネル作業用レジスタがユーザからもアクセスできることとOSが提供するメモリ保護機能を利用して，必要とされる アトミックな“割込みの検出と操作の無効化” を実現した.

MIPS 系 CPU は 32 個の 汎用レジスタを持っており，スタックポインタ専用のレジスタや，メモリアドレス専用のレジスタなど，用途別のレジスタはない．各々はハードウェア的には (レジスタ 0 が常に 0 が読めることを除き)全く等価である．しかしソフトウェアを書くときにはこれでは困るので，使用方法を決めている. 26, 27番目のレジスタは k0, k1 と呼ばれカーネル内で行われる例外処理時の一時利用のために予約されている．ユーザプロセスでもこれらを参照できるが 非同期に内容が破壊されるため，通常はユーザプロセスで使われることはない. 本方式ではこの通常ユーザプロセスでは使われない汎用レジスタを“割込みの検出と操作の無効化” に利用することにした.

2.2 アルゴリズム

Linux thread で必要とされる test and set 関数 tst( ) を用意する．この tst( ) は, 引数で与えられた排他変数の値が 0 なら，1 で置き換える. そして，もとの排他変数の値を戻り値とする． ライブラリは，排他変数の値 1 を使用中の意味で用いる.

この tst( ) 内での割込みを検出するために，カーネル作業用レジスタ k1 を用いる. ユーザプログラムに制御を移すとき，必ず k1 に特定の値 ACCESS_MAGIC をセットするようカーネルを変更する．アドレス ACCESS_MAGIC に, ユーザプロセスが アクセスすると，無効アドレス空間アクセス違反(以降 SEGV)の例外が発生するように，この値を決めておく． さらにカーネル内の SEGV 例外処理に次を追加する.

『tst( ) 内で ACCESS_MAGIC アドレスへのアクセス違反が発生したことを検出し，その場合 SEGV を発生させた命令ではなく tst( ) の先頭へ制御を戻す』

```
tst:                    // a0 is address of mutex var.
   move   k1, a0        // k1 = a0

   lw     v0, (a0)      // v0 = *a0
   nop
   bnez   v0, 1f        // if v0 != 0 goto 1
   nop
   li     t0,  1        // t0 = 1

   sw     t0,  (k1)     // *a0=t0, if the surrounded
                        // operation is atomic.
                        // Otherwise goto tst.
 1:
   j      ra            // return v0
NOTE:
The kernel always sets ACCESS_MAGIC to k1, on transit
to the user mode. The SEGV exception handler detects
writing to ACCESS_MAGIC in tst(), then it makes tst()
restart.
```

Figure 9. Test-and-set without ll and sc

図9の点線内で他のプロセスが割り込んだ場合，カーネルにより k1 の値が変更される．このため，排他変数へのストア命令 “sw t0, (k1)” で SEGV が発生し，tst()の最初からやり直される． すなわち, sw 命令が無効化される．これを図 10 に示す． sw 命令が成功するのは割り込みによって k1 の値が変化しない場合に限られるので，この関数はアトミックである.

本アルゴリズムでは，本来の排他変数以外の共有変数は不要なため, Lamport のアルゴリズムの実装で問題となる共有変数の扱いを気にする必要がない． さらに競合がない場合，単に手続き tst( )をユーザレベルで実行するだけなので，コンテキストスイッチは発生しない. 一般に競合が起きることは希なので，システムコールによる実装に比べて大きな性能改善が期待できる.

従来 k1 に不定の値を残したままユーザプロセスへ遷移していたのに対して, k1 の値を特定の値へ設定して遷移するだけなので， 本アルゴリズムはカーネルの他の部分やユーザプロセスに対して悪影響を与えることはない.

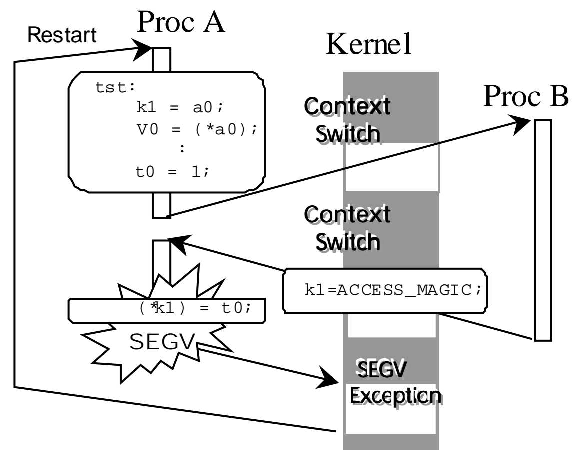


Figure 10. Restarting tst( )

2.3 実装

セキュリティの確保，SEGV 処理時の tst 手続き内の検出， 資源管理の簡易化のため, tst 手続き(図 9 相当)の提供を行うソフトウェア デバイス ドライバ “/dev/tst” をつくり， 各プロセスはこれを mmap( )でマップし，利用することにした.

セキュリティの確保とSEGV処理時のtst 手続き内の検出を簡単にするために，mmap( )のマップ先のアドレスを固定し，そのアドレスから 1 ページのみを実行専用としてマップ可能とした．そして，このアドレスから 1

ページには他のデバイス等はマップできないようにした．さらにメインテナンス性を考え，マップ先の変更が必要となってもライブラリ側で同じバイナリを使えるように，デバイスから read( ) により mmap( ) すべきアドレスを供給する設計とした．

ライブラリの linux thread のサポートルーチンは，最初の 1 回のみ以下を行う．

- /dev/tst を open( )
- read( ) を使いマップ先アドレスを読む
- mmap( )を使い，1 ページだけ，実行専用として直前に得たアドレスへマップ
- /dev/tst を close( )

そして従来提供されていたシステムコールの test and set の代わりに mmap( ) により プロセス空間にマップされた tst 手続きを呼び出す．

最初の初期化以外には，tst 手続きの呼出しにシステムコールは不用である．プロセスが 異常終了，強制終了した際にも mmap( ) を利用しているため不用な資源が残ったままになることはない．つまり，Lamport のアルゴリズムの実装で問題となった強制終了，異常終了時の後始末の問題は，発生しない．

カーネル内の SEGV 例外処理では，以下の条件 全てが満たされる場合，ユーザプロセスへの遷移先をマップされたページの先頭アドレスへ変更し処理を続行する．

- 書込み違反
- 書込み先アドレスは ACCESS_MAGIC,
- 発生場所は /dev/tst がマップされたアドレスの 1 ページ内

無論マップ先のアドレスを固定しない実装も可能である．この場合にはカーネル内の SEGV 例外処理での『発生場所は /dev/tst がマップされたアドレスの 1 ページ内』という判断が多少繁雑になる．

tst 手続きのメインテナンス性を上げるため, tst 手続きはローダブル モジュールとして提供される．つまり SEGV 例外処理を含むカーネル本体とは分離されたバイナリとなる． 上記の SEGV 例外処理内の判定を使えば，カーネル本体を再コンパイルせずに，ローダブル モジュール側 の tst 手続きの詳細を変更できる． ただし tst 手続きは，次の条件を満たしている必要がある．

- 再実行時に影響を及ぼす変更を行わない (例えば引数のレジスタ a0 を破壊しない)
- 手続きのサイズが 1 ページを越えない
- 競合時に無効化する命令は k1 を使った書込み

実際の実装では，図 9 で示した tst 手続きを改良し，ストア命令 ”sw t0, (k1)” 実行前に k1 をテストするようにした． k1 が a0 と異なるときに tst をやり直す (図 11)． これで，競合状態や排他変数のロード命令 “lw v0, (a0)” で ページフォルトが起きた場合でも，極力コンテキストスイッチが起きないようにできる．

```
tst:        // a0 is address of mutex var.
    move    k1, a0

    lw      v0, (a0)         // v0 = *a0
    li      t0, 1            // t0 = 1
    bnez    v0, 1f           // if v0!=0 goto 1
    nop
    bne     k1, a0, tst     // if k1 != a0 goto tst
    nop
    sw      t0, (k1)         // *a0=t0, if the surrounded
                             // operation is atomic.
                             // Otherwise goto tst.
1:
    jr      ra
    nop
```

Figure 11. Revised test-and-set without ll and sc

3 評価

評価のために PlayStation 2 上で動作する PS2 Linux を使い，各方式を実装した． その実装を使い以下の項目について測定し， 比較を行った．

1. 競合がない場合の単純な操作速度
2. 競合の可能性がある場合の一定時間内で完了する処理数
3. コードサイズの増加量

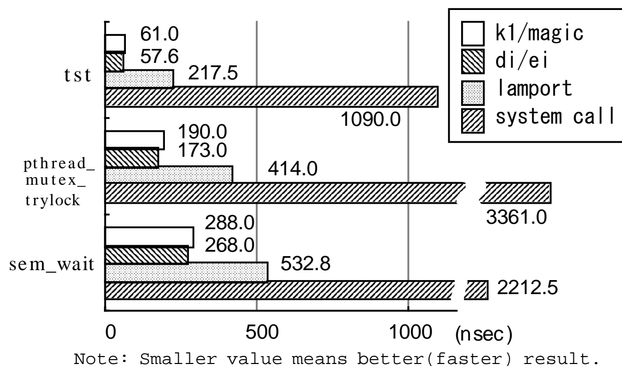
1 の『競合がない場合の単純な操作速度』測定のため，以下を単純に繰り返し呼ぶ評価用プログラムを作り比較した．

- POSIX 1003.1b semaphore の sem_wait( )
- POSIX 1003.1b pthread の pthread_mutex_trylock( )
- test and set 関数 tst( )

この結果を図 12 に示す．他にプロセスを動作させない状態で各々12 回測定し両端の値を除き，平均をとった．

以降，図表中で，本方式の実装を “k1/magic”， di, ei 命令を使った実装を “di/ei”， Lamport のアルゴリズムの実装を “lamport”， system call による実装を “system call”と略記する．

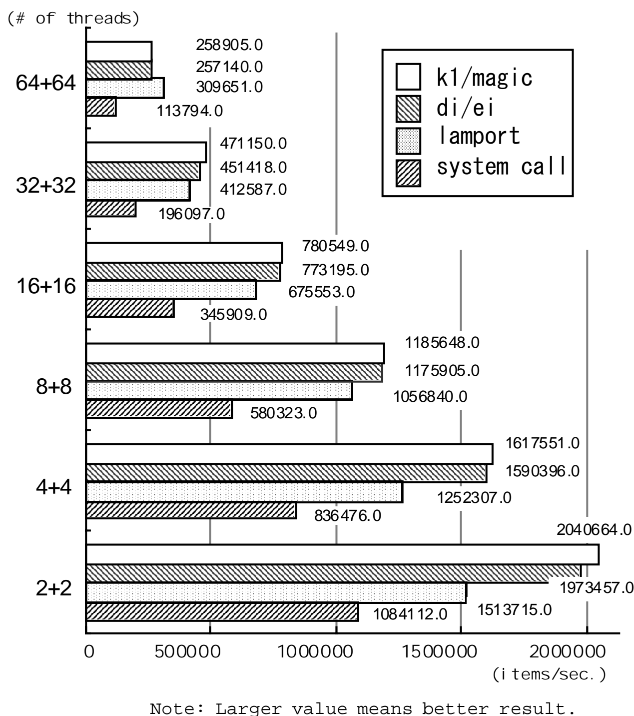
本方式により， 最も効率のよい di, ei 命令を使った実装と同程度の性能が得られた．従来多く用いられてきた system call の実装と比べると 7 倍以上の高速化を達成できた．



Note: Smaller value means better(faster) result.

Figure 12. Elapsed time of simple operation

2 の『競合の可能性がある場合の一定時間内で完了する処理数』測定のため, POSIX 1003.1b semaphore インターフェースで典型的な生産者-消費者問題を実装し, 一定時間内の資源の処理回数 (生産消費数)を比較した. 生産者, 消費者の数を (2, 2) から 双方とも倍にしていき (64, 64) になるまで測定した. 結果を図 13 に示す. 一つの条件毎に 20 秒間の測定を 12 回繰り返し, 両端の値を除き, 平均をとった. この測定でも, 本方式により di, ei 命令を使った実装と同程度の結果が得られた. また system call の実装と比べると 1.9 倍以上 性能を向上できた.



Note: Larger value means better result.

Figure 13. Processed items per second at the consumer-producer problem

3 の『コードサイズの増加量』に関しては, 各実装で必要となるライブラリ, デーモン, ローダブルモジュールおよびカーネルの追加変更のコードサイズを測定し比較した. 結果を表 1 に示す.

Table 1. Comparison of footprint

(unit: Kbyte)

	k1/magic	di/ei	lamport	system call
library	0.68	0.29	12.70	0.08
daemon	—	—	1.66	—
module	2.34	—	—	—
kernel	0.16	0.04	—	0.49
total	3.18	0.33	14.36	0.57

Note: Smaller value means better result.

本方式による実装では, コードの増加量を 3Kbyte 程度に押さえることができた.

なお, Lamport のアルゴリズムの実装では資源の後処理のために デーモンと呼ばれる独立したプロセスを用意する必用があり, これも評価の対象とした. di, ei 命令を使った実装では, TLB miss が起きないように test and set 手続きを同一ページ内に収める必要がある. 手続きの先頭でアライメントをとると最悪1ページ分(PS2 Linux では 4Kbyte)の領域が無駄になるが, この増加は表には含めていない.

以上の測定結果と『1.3 専用命令不要な排他制御方式』で述べた安全性と資源管理の有無をまとめて比較したのが表 2 である. 下線部は深刻な問題点であることを示す. このように本方式は, 他の方式と異なり深刻な問題点がなく, 全ての評価基準を高いレベルで満足することが確認できた.

Table 2. Comparison of feature

	k1/magic	di/ei	lamport	system call
Safety	Safe	<u>UNSAFE</u>	Safe	Safe
Efficiency (Speed)	Excellent	Excellent	Good	<u>POOR</u>
Resource handling	Easy	Easy	<u>DIFFICULT</u>	Un-necessary
Foot-print	Mideum	Small	Large	Small

Note: Underlined item means serious disadvantage.

4　まとめ

専用命令のない EE 上の Linux で安全なユーザレベルの排他制御を実現した．この方法は，専用命令を用いない従来の方法のような欠点を持たない．専用命令がない場合に，多く用いられてきたシステムコールによる実装と比べ 7 倍以上の高速化を達成しながら，コード増加量は 3Kbyte 程度に押さえることができた．

また本方式は，他の組込み用 MIPS 系 CPU の 高機能な組込み OS にも適用可能である．

なお，本論文は (株)ソニー･コンピュータエンタテインメントに出向中，社内学会 Sony Research Forum 2000 で発表した内容に一部修正を加えたものある．また測定値は開発中途のリビジョンでのものである．

参考文献

[1] 鈴置 雅一 PlayStation 2 用マイクロプロセッサ Emotion Engine, bit Volume 32, Number 1 January 2000, Pages 11-18 共立出版

[2] Andrew S. Tanenbaum. OPERATING SYSTEMS: Design and Implementation, Prentice-Hall Inc. 1987.

[3] Dominic Sweetman. See MIPS Run, Morgan Kaufman Publishers Inc. 1999.

[4] Lislie Lamport. A Fast Mutual Exclusion Algorithm, ACM Trans. On Computer System Volume 5, Number 1 February 1987, Pages 1-11.