情報処理学会論文誌　プログラミング　Vol. 1　No. 2　57–70 (Sep. 2008)

抽象状態同期による高機能ロックの実装と評価

大木敦雄[†1]　久野　靖[†1]

抽象状態とは，データ構造ないしオブジェクトの状態を複数の（あまり多くない，抽象化された）集合に排他的に分割し，現在の状態がそのいずれに含まれるかを明示的に管理する方式をいう．抽象状態同期とは，排他領域によってガードされる対象の抽象状態を同期機構側で保持し，スレッドなどの実行主体が排他領域に入れる条件を抽象状態の集合として表現する方式である．筆者らは先行研究において，抽象状態同期をOSカーネル内で実装することで，従来の実装方式では難しかった，可読性と効率を両立させた条件同期記述が可能であることを示した．本論文では，読み書きロックやバリア同期に代表される，単純な排他領域よりも込み入ったセマンティクスを持つ同期機構が，抽象状態同期を用いて簡潔かつ効率良く実装可能であり，抽象状態同期がさまざまな同期機構を効率良く実装するための基本的枠組みとしての特性を備えることを示す．

Implementing Utility Locks Using State Abstraction-based Synchronization

Atsuo Ohki[†1] and Yasushi Kuno[†1]

"State Abstraction" is a framework in which states of a data structure (or an object) are divided into small number of exclusive sets ("Abstract States," or AST), and current abstract state is explicitly managed by the code. In "State Abstraction-based Synchronization" (or AST-sync), abstract states of the data structure guarded by a critical region are similary managed, and an activity can enter the region only when the current state is included in the pre-specified set. In our previous research, we have implemented AST-sync inside an OS-kernel and have shown its effectiviness (in readability and performance) for standard conditional synchronization (aka. bounded buffers). In this paper, we demonstrate use of AST-sync to construct more complex synchronization mechanisms such as read-write locks or barriers, in simple and efficient manner.

1. はじめに

マルチプロセサシステムや分散システムなどの並列システムでは，動作主体間で同期（synchronization）をとる手段が不可欠である．並列システムのモデルとして共有メモリモデルが使われている場合，共有対象となるデータ構造やオブジェクトの整合性を保つために，排他領域（critical regions）が使用される．この場合，同期は基本的に，排他領域に入ろうとする実行主体の実行を他の実行主体が排他領域中を実行している間遅延させる形で実現される．

現実のシステムにおいて使用される同期の形態には，単純な排他領域だけでは済まないものも多くある．たとえば有限バッファでは，データを追加しようとする実行主体はバッファが満杯の間待つ必要があるが，データを取り出す実行主体はその間にバッファにアクセスしてデータを取り出すことになる．このような，特定の条件に応じて待ちを制御する同期を条件同期（conditional synchronization）と呼ぶ．

条件同期のための機構としては，条件変数（condition variables）[6]，セマフォ（semaphores）[2]，条件つき排他領域（conditional critical regions, CCR）[1),5] などが広く知られている．しかしこれらの機構にはそれぞれ弱点があり，特定の場合の記述がうまく書けなかったり，書けても複雑だったり，効率が悪かったりする．

このため，多くのアプリケーションで使われる特定の形の（高機能な）条件同期について，OSやライブラリ側で専用のライブラリAPIを提供することがある．読み書きロック（read-write locks）やバリア同期（barrier synchronization）などはそのようなものの代表である．ただしそれらの場合，ライブラリAPIが提供する機能から少しでも外れたものはこれらを利用できないという弱点がある．

この問題に対する本質的な解決方法は，条件変数などと同程度に汎用的な条件同期機構でありながら，多くのアプリケーションで使われる特定の形の条件同期が簡潔に記述でき，アプリケーションの必要に応じた拡張や手直しも容易なものを提供することである．

筆者らは，既存の条件同期機構が持つ記述性の低さや実行時オーバヘッドの高さを解消する条件同期機構として，抽象状態同期を提案し[7),8]，抽象状態同期をスレッドライブラリから利用するためのAPIと実装の開発[11]，およびOSカーネルのサポートによる高効率な実

†1 筑波大学大学院ビジネス科学研究科
Graduate School of Business Sciences, University of Tsukuba

© 2008 Information Processing Society of Japan

装の開発[12] を行ってきた．

本論文では，FreeBSD オペレーティングシステムで上述のカーネルサポートつき抽象状態同期を用いて読み書きロックとバリア同期を実装したものと，OS に標準搭載されているライブラリ実装，および標準の mutex ロックと条件変数を用いた実装とを，記述性/実行効率/拡張性の各面から比較/分析する．

以下，2 章では準備として，スレッドライブラリ向け抽象状態同期 API とその実装について説明する．続いて，高機能な条件同期の実例として，3 章では読み書きロック，4 章ではバリア同期を取り上げ，これらの抽象状態同期を用いた実装と既存実装の比較を行い，5 章で性能改善のための試みについて説明する．最後に 6 章で議論とまとめを行う．

2. スレッドライブラリと抽象状態同期

2.1 抽象状態と抽象状態同期

オブジェクト指向言語では，オブジェクトの状態はインスタンス変数群の値によって定義されるが，そのオブジェクトを外部から（抽象データ型として）扱う場合，すべての状態を区別する必要はない．

たとえば読み書きロックをオブジェクトとした場合，その区別されるべき状態は「空いている」「読まれている」「読まれていて書き手が待っている」「書かれている」の 4 つであり，読み手が何人いるかという細かい状態の区分は内部的には（ロックを正しく空きに戻すために）必要だが，外部からは見えなくてよい．外部から見える状態を，オブジェクト内部の状態を抽象化したものであることから，「抽象状態（abstract states)」と呼ぶ．

抽象状態同期は抽象状態を条件同期に利用するものであり，オブジェクトごとに抽象状態を明示的な値として保持させ，オブジェクトに付随する排他領域に入る実行主体は，その入口で行おうとする操作が実行可能な抽象状態の集合を明示し，現在の抽象状態がその集合に含まれない場合には実行を遅延する（たとえば読み手は読み書きロックが「空いている」か「読まれている」状態の場合だけロックを獲得してよい)．また，排他領域に入った実行主体は，出口において次の抽象状態を設定する．その結果，その状態で実行を許される遅延中の実行主体から 1 つが選ばれて実行を再開する．

抽象状態同期では，抽象状態の数が 1 語のビット数以内であれば，各状態を 1 ビットに対応させ，ビット演算により遅延が必要かどうかを効率良く判定できる[7]．また，クラス間の継承関係がある言語に適用する場合でも，親クラスにおける抽象状態を子クラスの複数の抽象状態に系統的に分割することで継承異常問題を回避し，同期記述の再利用を可能にできる[8]．

表 1 抽象状態同期に基づく同期 API
Table 1 Synchronization API of AST-sync.

呼び出し API	機能
`int ast_mutex_init(struct ast_mutex *m, int state)`	初期化
`int ast_mutex_destroy(struct ast_mutex *m)`	廃棄
`void ast_mutex_enter(struct ast_mutex *m, int mask)`	排他領域に入る
`void ast_mutex_exit(struct ast_mutex *m, int state)`	排他領域を出る

2.2 スレッドライブラリへの抽象状態同期の導入

前節では抽象状態がオブジェクトに付随するものとして述べたが，オブジェクト（ないしデータ構造）を保護するロックに抽象状態が付随すると考えてもよい．筆者らはこのような考えに基づき，スレッドライブラリから使う抽象状態同期つきロックの API を設計した[11]．**表 1** に API を示すが，その動作は次のとおり．

- 抽象状態は 1 語のうち 1 ビットだけが “1” の値で表し，ロック内部に保持する．その初期値は `ast_mutex_init()` で初期化時に指定する．
- スレッドがロックを獲得しようとする際には `ast_mutex_enter()` を呼び出すが，このとき「どの抽象状態の場合に獲得するか」を表すマスク値（指定する抽象状態値すべての論理和）を指定する．マスク値とロックの抽象状態値の論理積が 0 のスレッドは待たされ，0 でないスレッドのみがロックを獲得できる．
- ロックを獲得したスレッドは排他領域を実行後に `ast_mutex_exit()` でロックを解放するとともに次の抽象状態値を設定する．このとき，待たされているスレッドの中でマスク値と新しい抽象状態値の論理積が 0 でないスレッドがあれば，そのようなスレッドのうち 1 つが任意に選ばれロックを獲得する．

筆者らは抽象状態同期機構を持つロックを，POSIX スレッド API を用いて実装し，有限バッファの例題を記述してみて，従来の条件変数を用いたものより簡潔であること，ウォーターマークつき有限バッファのように複数の状態を待つ必要があり，条件変数では書けない条件同期も問題なく記述できることを確認した[11]．この実装は POSIX スレッドをサポートするプラットフォームに対する可搬性があるが，条件変数を用いた条件同期より実行オーバヘッドは大きい．ただし，今日広く用いられている，繰返し条件式を評価するスタイルの条件同期（CCR に相当）と比べた場合の実行性能は同程度である．

抽象状態同期をより効率良く実装するために，筆者らはさらに，FreeBSD 6.0/6.2-

© 2008 Information Processing Society of Japan

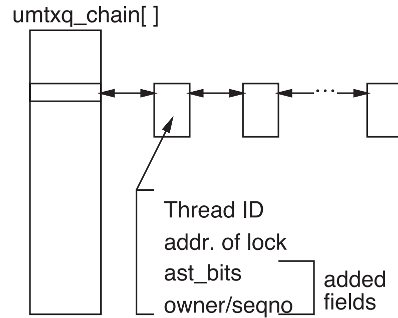

図 1　カーネル内データ構造の変更
Fig. 1　Data structures in kernel land.

RELEASE に付属する `libthr` スレッドライブラリ（1 ユーザスレッド=1 カーネルスレッドの実装）を前提にカーネルに機能追加を行った．その追加内容は次のとおり．

- FreeBSD のユーザレベルから呼び出す待ち合わせ機構が使用する待ち合わせキューは，決まった個数ぶんのデータ構造が配列 `umtxq_chain[]` にあらかじめ用意されていて，その下に待ち合わせるスレッドが連結リストとしてつながるようになっている（**図 1**）[⋆1]．この連結リストのデータ構造を変更して，抽象状態マスクと，待っているスレッドを起こすときにデータ構造を保護するロックを獲得するために使用する値を格納するように拡張した．
- FreeBSD ではユーザレベルから呼び出す同期機構は，`_umtx_op()` という 1 つのシステムコールが入口となっており，その呼び出し時にどのような操作を行うかをパラメータとして渡す．その操作の種類を増やして，抽象状態同期ロックの待ち/再開の 2 つの操作を追加した．具体的には待ちの操作ではこのスレッドが待ち合わせる抽象状態マスクをデータ構造に格納し，再開ではキューの先頭から順に，待ちスレッドで (1) 当該抽象状態ロックで待ち合わせていて，(2) 次の状態とその待ちスレッドに付属する抽象状態マスクとの論理積が 0 でないスレッドを探し，見つかった最初の 1 個を起こすようにした．

これらのコード修正の総行数は約 100 行であった．

⋆1 ウェイトチャネル値（カーネル内で待ち合わせ対象を識別する 32 ビットの値）と `umtxq_chain[]` のエントリの対応は決まったハッシュ関数によっている．

このカーネルに追加された機能を利用した抽象状態同期ライブラリの実装では，条件同期により待っているスレッドを再開するとき，条件を満たすものちょうど 1 個を選んで再開でき，反復して条件を評価する方式より効率が優っている．実際，有限バッファの出し入れを行うベンチマークを行った結果，反復判定を行う方法と比較してオーバヘッドが 1/3 程度であり，条件変数を用いた実装と比べても 1～10%程度高速であることが確認できた[11),12)]．

3. 読み書きロック

3.1 読み書きロックとその実装

読み書きロックは実アプリケーションで共有データをガードするために頻繁に使われる形態のロックであり，次のように動作する．

- 読み手は何人でも同時にデータにアクセスできる．
- 書き手は 1 時に 1 人だけデータにアクセスできる．
- スケジューリングが公平（fair）であれば，書き手が無限に待たされ続けること（飢餓）はない．

最後の項はデータ保護の点からは必須でないが，多くの読み手が入れ替わりつつ読み続けると書き手が飢餓状態となり実用上不都合なため，POSIX スレッドを含む多くの実装がこの仕様を入れている．

抽象状態同期ではロックの状態を明示的に扱うため，上記のような仕様をもとに「区別可能な状態」と「それら相互の遷移」に着目することで仕様に合致するロックを設計することができる．以下にその過程を説明する．

- 仕様から「読み手がアクセスしている状態」「書き手がアクセスしている状態」「どちらもアクセスしていない状態」が存在することが読み取れる．これらを READ，WRITE，FREE と名付ける（**図 2** (a)）．
- 次に，状態間の遷移を検討する．読み手/書き手のロック獲得/解放により，少なくとも FREE と READ，FREE と WRITE の間で相互に遷移があることは分かる．ロックの 1 つの状態遷移は 1 つのスレッドによって引き起こされる（複数のスレッドが同時にロックを操作することはない）ため，READ と WRITE の間で直接遷移することは（読み手のスレッドと書き手のスレッドが同時に関与する必要があるため）ありえない（図 2 (b)）．
- 「書き手が無限に待たされることはない」という要件から，書き手が待っているときは新たな読み手がロックを獲得することはないと分かる．このため，「読み手が 1 人以上

© 2008 Information Processing Society of Japan

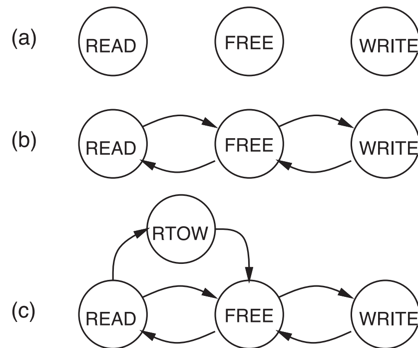

図 2　読み書きロックの状態遷移
Fig. 2　State transition of read-write locks.

ロックをとっているが，新たな読み手は入れない」という新たな状態が必要であることが分かる．これを RTOW と名付ける．

- RTOW への遷移としては少なくとも，READ からのものがある．また，RTOW からの遷移は，前記と同様の理由により WRITE へ行く場合はないので，状態をこれ以上増やさないなら，FREE へ遷移することになる（図 2 (c)）．

抽象状態の図 2 (c) の遷移に基づく読み書きロックの実装を**図 3** に示す．各操作の概要は次のとおり．

- `rdlock` — 読み手のロック．FREE か READ のときに通過し，読み手のカウントを増やし，状態を READ にするが，ただし書き手が待っている場合は RTOW にする．
- `rdfree` — 読み手のアンロック．状態は READ か RTOW のはず．カウントを減らし，まだ 0 でない（残っている読み手がいる）なら状態を READ にするが，ただし書き手が待っている場合は RTOW にする．0 である（自分が最後の読み手）なら，状態を FREE にする．
- `wrlock` — 書き手のロック．WRITE 以外のすべての状態でロックをとり，読み手がいるなら待ちカウントを増やし，RTOW にして，再度 FREE になるまで待ち，FREE でロックがとれたら待ちカウントを減らして WRITE にする．
- `wrfree` — 書き手のアンロック．状態は WRITE のはず．状態を FREE にするだけ．

なお，書き手は 1 人しか入れないので，書き手が入っている間ロックを保持していていいなら`/**/`を付したロックの解放と再獲得は不要となる．これは読み書きロックの使われ方によるが，メモリ内の値を更新するなどの用途であればロックの解放と再獲得を省略しても問題ない．

```
#define S_FREE  0x00000001  /* 空き */
#define S_READ  0x00000002  /* 読み */
#define S_WRITE 0x00000004  /* 書き */
#define S_RTOW  0x00000008  /* 書きへ移行 */
ast_mutex_t mutex;
volatile int rcnt = 0, wcnt = 0;
void rwinit() { ast_mutex_init(&mutex, S_FREE); }
void rwclose() { ast_mutex_destroy(&mutex); }
void rdlock() {
  ast_mutex_enter(&mutex, S_FREE|S_READ);
  ++rcnt;
  ast_mutex_exit(&mutex, wcnt?S_RTOW:S_READ);
}
void rdfree() {
  ast_mutex_enter(&mutex, S_READ|S_RTOW);
  if(--rcnt)
    ast_mutex_exit(&mutex, wcnt?S_RTOW:S_READ);
  else
    ast_mutex_exit(&mutex, S_FREE);
}
void wrlock() {
  ast_mutex_enter(&mutex, S_FREE|S_READ|S_RTOW);
  if(rcnt) {
    ++wcnt;
    ast_mutex_exit(&mutex, S_RTOW);
    ast_mutex_enter(&mutex, S_FREE);
    --wcnt;
  }
  ast_mutex_exit(&mutex, S_WRITE); /**/
}
void wrfree() {
  ast_mutex_enter(&mutex, S_WRITE); /**/
  ast_mutex_exit(&mutex, S_FREE);
}
```

図 3　抽象状態同期による読み書きロック
Fig. 3　Read-write locks with AST-sync.

図 3 の実装では待っている書き手がいても最後の読み手が状態を FREE にするので，そ

© 2008 Information Processing Society of Japan

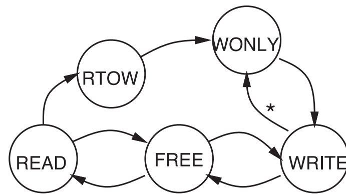

図 4　拡張した読み書きロックの状態遷移
Fig. 4　State transition of augmented read-write locks.

の時点で読み手と書き手がいた場合どちらが次にロックを獲得するかは定まらない（ただし読み手が獲得した場合，次の状態は RTOW になる）．これは前掲の仕様に合致しているが，さらに次のいずれかの要件を追加することも考えられる．

(a) 待っている書き手がいるときは，最後の読み手の次は必ず書き手の番になる．

(b) 待っている書き手がいる間は，読み手が新たにアクセスすることはない．

(a) では読み手の次は必ず書き手だが，その書き手が終わった後は読み手と書き手のどちらかは定めない．(b) では書き手がいる限り必ず書き手が優先となる．

このどちらの要件も「次は書き手のみがロックをとれる」という新たな状態を必要とする．これは，前述のように RTOW から直接 WRITE へ遷移することはできないので，RTOW から読み手（のロック解放）によってその新しい状態に遷移し，続いて書き手（のロック獲得）によって WRITE へ遷移するためである．この状態を WONLY と名付ける（**図 4**）．したがって遷移は RTOW から WONLY，WONLY から WRITE へのものが基本となるが，さらに WRITE から WONLY への遷移を含める可能性がある（*の遷移）．これは「書き手がロックを解放するとき次もまた書き手のみ」を実現するものなので，前記 (b) の要件に対応する（*の遷移がない場合は (a) に対応する）．

このように抽象状態同期では，必要な状態を列挙し，「1 つの遷移に複数のスレッドが関与しない」という条件を守って遷移を設定してゆくことで，さまざまな仕様を持つ同期機構を比較的素直に組み立てることができる．

図 5 に，(b) の仕様の読み書きロックの条件変数を用いた実装を示す．こちらの方がコードはかなり複雑であり，また図 3 や (a) の仕様を実現するための変更も簡単ではない．これらの要件の違いはアプリケーションの要請に応じて選択可能であることが望ましい．読み書きロックのライブラリ実装ではそのような選択は不可能であるし，また既存の条件同期のための標準的な機構である条件変数では（図 5 の複雑さから考えて）これらの要件を正しく実装するのは簡単ではないといえる．

```
pthread_mutex_t mutex;
pthread_cond_t rsignal, wsignal;
volatile int state, wcnt;
void rwinit() {
  state = wcnt = 0;
  pthread_mutex_init(&mutex, NULL);
  pthread_cond_init(&rsignal, NULL);
  pthread_cond_init(&wsignal, NULL);
}
void rwclose() {
  pthread_mutex_destroy(&mutex);
  pthread_cond_destroy(&rsignal);
  pthread_cond_destroy(&wsignal);
}
void rdlock() {
  pthread_mutex_lock(&mutex);
  while(wcnt > 0 || state < 0)
    pthread_cond_wait(&rsignal, &mutex);
  ++state;
  pthread_mutex_unlock(&mutex);
}
void rdfree() {
  pthread_mutex_lock(&mutex);
  if(--state == 0) {
    if(wcnt > 0) pthread_cond_signal(&wsignal);
  }
  pthread_mutex_unlock(&mutex);
}
```

図 5　条件変数を用いた読み書きロック
Fig. 5　Read-write locks with condition variables.

これに対し，抽象状態同期に基づく実装であれば，状態遷移を明示的に扱うことから，状態遷移として定式化できるものであれば，さまざまな要件に容易に適応可能であるという利点がある．

3.2　読み書きロックの性能評価

抽象状態同期を用いて記述した読み書きロックの性能を評価するため，マイクロベンチマークによる時間計測を行った．計測には Core 2 Extreme QX6700 2.66 GHz（4 コア）を

© 2008 Information Processing Society of Japan

```
void wrlock() {
  pthread_mutex_lock(&mutex);
  while(state != 0) {
    ++wcnt;
    pthread_cond_wait(&wsignal, &mutex);
    --wcnt;
  }
  state = -1;
  pthread_mutex_unlock(&mutex);
}
void wrfree() {
  pthread_mutex_lock(&mutex);
  if(wcnt) {
    pthread_cond_signal(&wsignal);
  } else {
    pthread_cond_broadcast(&rsignal);
  }
  state = 0;
  pthread_mutex_unlock(&mutex);
}
```

図 5　条件変数を用いた読み書きロック（続き）
Fig. 5　Read-write locks with condition variables (cont.).

表 2　読み書きロック計測のパラメータ
Table 2　Measuring parameters for read-write locks.

スレッド比 ($N_r : N_w$)	1 : 1	10 : 1	100 : 1	250 : 1	2000 : 1
読み書き比 ($N_r \times D_r : N_w \times D_w$)	1 : 1	1 : 1	1 : 1	10 : 1	100 : 1
リーダ数 (N_r)	20	200	2,000	2,000	2,000
リーダ反復数 (D_r)	10,000	1,000	100	100	100
ライタ数 (N_w)	20	20	20	8	1
ライタ反復数 (D_w)	10,000	10,000	10,000	2,500	2,000
総スレッド数	40	220	2,020	2,008	2,001

4 コア（論理 CPU 数 4）で使用し，OS は FreeBSD 6.2-RELEASE を使用した．

ベンチマークは N_r 個のリーダスレッドと N_w 個のライタスレッドを生成し，それぞれが D_r 回および D_w 回反復して読み書きロックを獲得/解放し，全体の所要時間を計測するものである．計測は，生成できるスレッド数に限界があったため，総スレッド数 2,000 程度の範囲内で N_r，N_w，D_r，D_w を**表 2** に示す 5 通りに変化させて実施した．

表 3　読み書きロックの性能（スレッド比 1 : 1，読み書き比 1 : 1）
Table 3　Performance of read-write locks (threads ratio 1 : 1, read-write ratio 1 : 1).

実装の種類	ユーザ時間	システム時間	経過時間
1 rwlock	1.165(0.095)	34.802(1.011)	13.341(0.364)
2 cond	1.168(0.088)	34.611(0.785)	13.292(0.280)
2' 〃(ロック保持)	0.467(0.063)	20.452(0.532)	8.680(0.168)
3 ast_mutex	1.144(0.117)	40.728(1.026)	15.847(0.318)
3' 〃(ロック保持)	0.572(0.060)	20.430(0.437)	9.307(0.143)
3b ast_mutex'	0.523(0.067)	28.569(0.386)	13.615(0.157)
3b' 〃(ロック保持)	0.331(0.054)	22.191(0.419)	9.899(0.156)

具体的にはこれらの値は，(1) 読み書き比（リーダとライタのロック総獲得数の比，$N_r \times D_r : N_w \times D_w$）が 1 : 1 でスレッド比 ($N_r : N_w$) を 1 : 1，10 : 1，100 : 1 と変化させた場合，および，(2) 読み書き比を 1:1，10:1，100:1 と変化させた場合の検討が行えるように選んだ．(2) については，スレッド比を調整することで，ライタだけが早期に終わってしまわないように配慮した．読み書き比，スレッド比ともライタの方が少ない場合だけ計測したのは，現実のアプリケーションでライタの方が多い状況はほとんどないと考えたためである．

計測対象とした読み書きロックの実装は次の 5 種類である（3 と 3' については，5 章で述べる改良を施した版である 3b と 3b' の計測も掲載してあるが，これについては後で述べる）．

1　システム標準の読み書きロック `pthread_rwlock`
2　システム標準の排他ロック `pthread_mutex` と条件変数 `pthread_cond` を用いて実装した読み書きロック（図 5）．
2'　2 を手直しして，書き手のロック獲得時には排他ロックを持ち続けるもの．
3　カーネル版抽象状態同期を用いて実装した読み書きロック（図 3 を書き手優先に手直ししたもの）．
3'　3 を手直しして，書き手のロック獲得時には排他ロックを持ち続けるもの．

書き手優先のセマンティクスは，1 と 2 と 3 が前節の (b)，2' と 3' が (a) である．この違いは，2' と 3' では書いている間に排他ロックを解放しないので，他の書き手がフラグを立てて待っていることを知らせることができないためである．

スレッドは作成するとただちに実行開始するため，バリアを用いて全スレッドが動き出してから読み書きロックの獲得/解放を開始するようにした．計測は各ケースとも同じものを 40 回ずつ実行し，平均と標準偏差を求めた．**表 3**，**表 4**，**表 5**，**表 6**，**表 7** に計測結果を

© 2008 Information Processing Society of Japan

表 4　読み書きロックの性能（スレッド比 10 : 1，読み書き比 1 : 1）
Table 4　Performance of read-write locks (threads ratio 10 : 1, read-write ratio 1 : 1).

実装の種類	ユーザ時間	システム時間	経過時間
1 rwlock	1.191(0.092)	36.187(0.759)	14.007(0.227)
2 cond	1.209(0.097)	36.371(0.621)	14.081(0.207)
2' 〃 (ロック保持)	0.473(0.076)	21.238(0.602)	9.102(0.182)
3 ast_mutex	1.104(0.097)	40.648(0.653)	15.818(0.192)
3' 〃 (ロック保持)	0.575(0.063)	20.654(0.355)	9.444(0.112)
3b ast_mutex'	0.527(0.061)	28.367(0.336)	13.480(0.114)
3b' 〃 (ロック保持)	0.311(0.053)	21.757(0.326)	9.653(0.118)

表 5　読み書きロックの性能（スレッド比 100 : 1，読み書き比 1 : 1）
Table 5　Performance of read-write locks (threads ratio 100 : 1, read-write ratio 1 : 1).

実装の種類	ユーザ時間	システム時間	経過時間
1 rwlock	1.210(0.112)	36.859(0.740)	14.267(0.240)
2 cond	1.249(0.092)	37.215(0.695)	14.375(0.223)
2' 〃 (ロック保持)	0.559(0.076)	22.928(1.107)	9.708(0.363)
3 ast_mutex	1.139(0.104)	40.137(2.123)	15.783(0.326)
3' 〃 (ロック保持)	0.630(0.078)	21.416(0.620)	9.691(0.180)
3b ast_mutex'	0.555(0.082)	28.668(0.377)	13.700(0.184)
3b' 〃 (ロック保持)	0.348(0.049)	21.948(0.357)	9.817(0.153)

表 6　読み書きロックの性能（スレッド比 250 : 1，読み書き比 10 : 1）
Table 6　Performance of read-write locks (threads ratio 250 : 1, read-write ratio 10 : 1).

実装の種類	ユーザ時間	システム時間	経過時間
1 rwlock	0.413(0.056)	16.602(0.417)	6.998(0.126)
2 cond	0.397(0.048)	16.581(0.424)	6.989(0.138)
2' 〃 (ロック保持)	0.333(0.047)	15.564(0.343)	6.655(0.090)
3 ast_mutex	0.476(0.063)	16.253(0.270)	7.245(0.076)
3' 〃 (ロック保持)	0.439(0.061)	14.806(0.357)	6.743(0.103)
3b ast_mutex'	0.273(0.043)	15.927(0.166)	7.188(0.065)
3b' 〃 (ロック保持)	0.242(0.052)	15.465(0.423)	6.889(0.182)

表 7　読み書きロックの性能（スレッド比 2000 : 1，読み書き比 100 : 1）
Table 7　Performance of read-write locks (threads ratio 2000 : 1, read-write ratio 100 : 1).

実装の種類	ユーザ時間	システム時間	経過時間
1 rwlock	0.357(0.048)	15.295(0.647)	6.495(0.193)
2 cond	0.320(0.051)	15.443(0.621)	6.532(0.187)
2' 〃 (ロック保持)	0.316(0.063)	14.742(0.300)	6.346(0.100)
3 ast_mutex	0.419(0.055)	14.173(0.297)	6.460(0.099)
3' 〃 (ロック保持)	0.432(0.061)	14.047(0.220)	6.424(0.068)
3b ast_mutex'	0.234(0.047)	14.611(0.200)	6.519(0.068)
3b' 〃 (ロック保持)	0.231(0.042)	14.583(0.249)	6.483(0.102)

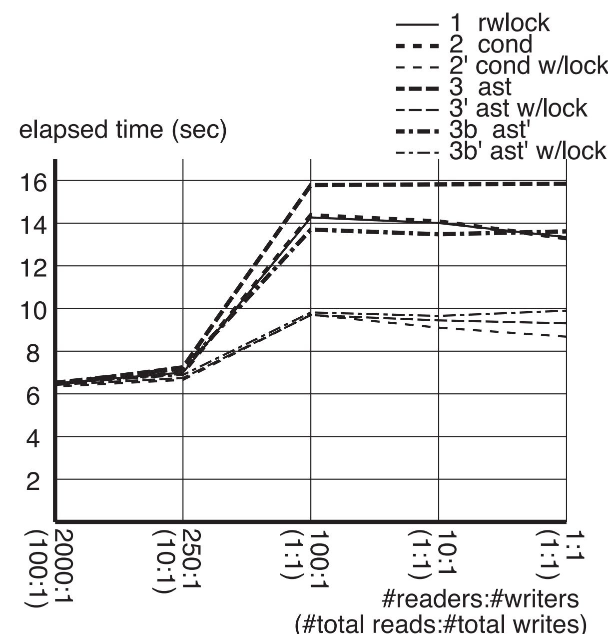

図 6　経過時間のグラフ
Fig. 6　Plot of elapsed time.

示す．単位はすべて秒で，かっこ内は標準偏差を表す．

また，計測値のうち経過時間をグラフ化したものを**図 6** に，1 操作あたりの所要時間（経過時間を $N_r \times D_r + N_w \times D_w$ で割ったもの）をグラフ化したものを**図 7** に示す（左からスレッド比が大きい順に並べた）．これらにおいて経過時間を用いたのは，CPU 数がすべて同一（4 個）であり，待ちによる遅延も含めて評価したいと考えたためである．

結果を見ると，読み書き比が 100 : 1 および 10 : 1 の場合（ライタの介在が少ない場合）は，どの方式でも所要時間はさほど変わらないことが分かる．これは，ライタの介在が少なければ条件による待ちそのものの発生頻度が小さくなるため，その実装の影響も観測されなくな

© 2008 Information Processing Society of Japan

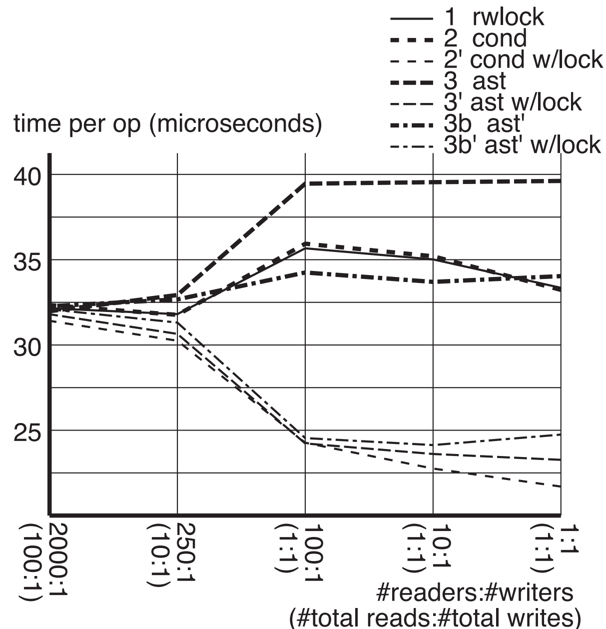

図 7　1 操作あたり所要時間のグラフ
Fig. 7　Plot of time required for one operation.

ることを考えれば当然の結果だといえる．

一方，読み書き比が 1:1（ライタの介在が多い）場合は，条件による待ちが多く発生するため，実装の差が現れやすくなっている．実アプリケーションにおける読み書きロックで読み書き比が 1:1 となることはまれだとしても，条件同期一般における競合が多い状況の振舞いの一例として，このような計測にも意味があると考える．以下ではこのケースを中心に検討する．

システム標準の読み書きロックと条件変数を用いた実装は性能がほぼ等しい．実際，ソースコードを見ると標準の読み書きロックは条件変数を用いて実装されており，アルゴリズムも（汎用性のための多くの処理を除けば）同等であった．この両者と比較して，抽象状態同期を用いた実装は経過時間で見てやや遅いことが分かる．

ただし，抽象状態同期による実装はリーダのスレッド数が 2,000 と多くなった場合（$N_r : N_w = 100 : 1$）でも所要時間が変化していないのに対し，読み書きロックと条件変数ではリータのスレッド数が大きくなった場合に所要時間が増大する傾向が見られる．このため，スレッド数がさらに大きくなった場合は抽象状態同期が有利になる可能性もある．

次に，ライタが書いている間は排他ロックを保持し続ける版の実装（2'，3'）について検討する．こちらも，読み書き比が 100:1，10:1 では（そもそも書き込みロックの使用数が少ないので）元の版（2，3）とほとんど差がない（10:1 ではいくらか速くなってはいる）．一方，読み書き比が 1:1 では元の版と比べて経過時間が 60%〜70%となっており，かなりの高速化が見られる．

ロックを保持する版相互の比較では，条件変数を用いた実装である 2' より，抽象状態同期を用いた実装である 3' の方がやや遅いが，こちらでも 2' はリーダスレッド数が増えたときに所要時間が増えるのに対し，3' は変化が少ないため，リーダスレッド数が 2,000 のときは両者は同程度の所要時間となっている．

先に述べたように，抽象状態同期では基本的な同期機構を組み合わせてアプリケーションソフトに合わせた仕様を持つ同期を構成しやすく，したがって 3' のようなロックを保持する仕様を用いた高速化も行いやすいと考える．

4. バ　リ　ア

4.1　バリアとその実装

バリアは SPMD（single program，multiple data）型の並列プログラムなどで複数スレッドが同期しつつ進行するために使われる形態の同期機構で，次のように動作する．

- 最初にスレッド数 N を指定してバリアを生成する．
- バリアを通過しようとするスレッドは，最初の $N-1$ 個は待たされる．最後の N 個目がバリアに到着すると，待っていたすべてのスレッドは進行可能となる．

以下に抽象状態同期によるバリア同期の設計過程を解説する．

- 区別可能な状態は明らかに「バリアがまだ閉じている」「開いている（全員が到着済み）」の 2 つであり，遷移はその 2 者間相互だけである（**図 8** (a)）．
- ただし，バリアの実装はこの 1 つの抽象状態ロックだけでは行えない．なぜなら「待たされている」スレッドは中のデータに触れないので自分が何番目かカウントできないからである．このため，もう 1 つのロックを追加してこのロックでカウンタをガードする（図 8 (b)）．この 2 番目のロックは状態が 1 つでよいので，抽象状態ロックである必要はないが，説明の都合上これも抽象状態ロックであるものとする．

© 2008 Information Processing Society of Japan

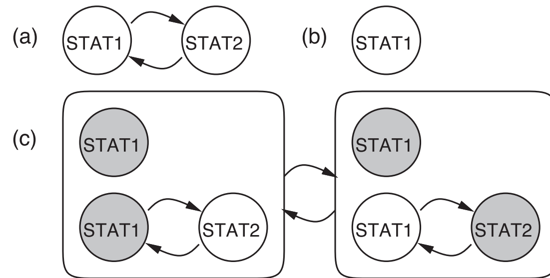

図 8　バリアの状態遷移
Fig. 8　State transition of barrier synchronization.

- バリア全体は 2 つの抽象状態ロックの組で実現される．これにより，バリア全体の状態は 2 つのロックの状態を組にしたものからなる（図 8 (c)，個々のロックが現在いる状態を網掛けで示す）．
- 2 状態のロックは，最初は「閉じている」状態で始まり，全員が到着したら「開いている」状態に遷移させる．素直に考えれば全員が通過し終わった後で再度「閉じている」状態に戻す必要がありそうだが，次にバリアを使用するときに状態の「開いている」「閉じている」を逆に読み替えることで戻す操作を省略できる．このため，状態は STAT1, STAT2 と番号で記述している．

このように，抽象状態同期ロックを複数組み合わせる場合も，状態の組合せにより見通し良く設計を行うことができる．一般に状態数の最大値は各ロックの状態数の積となるため，ロック数や各ロックの状態数が多くなると全体の状態が非常に多くなる可能性があるが，そのような複雑な同期構成については他の同期機構による記述も難しいものになると予想される．

抽象状態同期を用いたバリアの実装を**図 9** に示す．関数 `barrier()` のあらましは次のとおり．

- `count` はスレッド数 N で初期化してある．ロック 2 は最初 STAT1 にしてある．
- 1 つ目のロックをとり（これは単純な排他領域），`count` を減らし，結果が 0 か否かで「最後の 1 人」か「他の人」か分かれる．
- 「他の人」であれば，ロック 2 が「次の状態」（最初は STAT2）になるまで遅延し，とれたら同じ状態のままただちに解放する．
- 「最後の 1 人」であれば，`count` を次に備えて N に戻し，「現在の状態」と「次の状態」を交換し，ロック 2 を「次の状態」に切り替える（これによって待っていた「他の人」が全員通過する）．

```
#define S_STAT1 0x00000001
#define S_STAT2 0x00000002
ast_mutex_t mutex1, mutex2;
volatile int init, count, curr, next;
void barinit(int n) {
 init = count = n; curr = S_STAT1; next = S_STAT2;
 ast_mutex_init(&mutex1, S_STAT1);
 ast_mutex_init(&mutex2, S_STAT1);
}
void barclose() {
  ast_mutex_destroy(&mutex1);
  ast_mutex_destroy(&mutex2);
}
void barrier() {
  int c = curr, n = next;
  ast_mutex_enter(&mutex1, S_STAT1);
  if(--count) {
    ast_mutex_exit(&mutex1, S_STAT1);
    ast_mutex_enter(&mutex2, n); /* ☆ */
    ast_mutex_exit(&mutex2, n);  /* ☆ */
  } else {
    count = init; curr = n; next = c;
    ast_mutex_exit(&mutex1, STAT1);
    ast_mutex_enter(&mutex2, c);
    ast_mutex_exit(&mutex2, n);
  }
}
```

図 9　抽象状態同期を用いたバリア
Fig. 9　Barrier synchronization with AST-sync.

図 10 に，条件変数を用いたバリアの実装を示す．バリアの場合は，その仕様が条件変数の持つ機能と非常に近いため，「最後の 1 人以外は条件変数で寝て，最後の 1 人がそれを全員起こす」という方針で簡潔に実装できる．一般には条件変数を用いた実装は複雑なものとなりやすいので，この状況は例外的である．

次に，前述の設計を拡張して弾性バリア[10)]（elastic barrier）を実現してみる．弾性バリアとは，バリアに「幅」が設けられ（つまり入口と出口があり），すべてのスレッドが入口

© 2008 Information Processing Society of Japan

```
pthread_mutex_t mutex;
pthread_cond_t cond;
volatile int number, count;
void barinit(int n) {
  number = count = n;
  pthread_mutex_init(&mutex, NULL);
  pthread_cond_init(&cond, NULL);
}
void barclose() {
  pthread_mutex_destroy(&mutex);
  pthread_cond_destroy(&cond);
}
void barrier() {
  pthread_mutex_lock(&mutex);
  if(--count) {
    pthread_cond_wait(&cond, &mutex);
  } else {
    count = number; pthread_cond_broadcast(&cond);
  }
  pthread_mutex_unlock(&mutex);
}
```

図 10　条件変数を用いたバリア
Fig. 10　Barrier synchronization with condition variables.

を通過すると出口から出られるようになるようなバリアである．弾性バリアでは，入口に到達した人は待たなくてもただちに「中」の処理に進むことができ，出口に到達したときに最後の人が入口を通過済みならやはり待たずに出られるため，通常のバリアより同期オーバヘッドを軽減できるとされている．

弾性バリアの要点は，まだバリア中に残っているスレッドがいる場合は「次回のバリアの使用」に進んではならない（新たなスレッドが入口を通過することを許さない）ことである[*1]．これを実現するために，入口を N 番目のスレッドが通過したら入口を閉じて出口を開け，出口を N 番目のスレッドが通過したら出口を閉めて入口を開ける（**図 11**．今回はどのみち出口でも数を数えるため，状態の読み替えは行わず，状態も分かりやすいように OPEN と CLOSE にした）．先に述べたように，複数のスレッドが 1 つの遷移に関与する

*1　このことは通常の（幅のない）バリアでは問題にならなかった．なぜなら，カウンタは N 番目のスレッドが通過するときにリセットされ，2 番目の抽象状態ロックは N 番目のスレッドが通過するときに STAT1 と STAT2 の交換を行うというふうに，それぞれの排他領域内で閉じた動作で済んでいたためである．

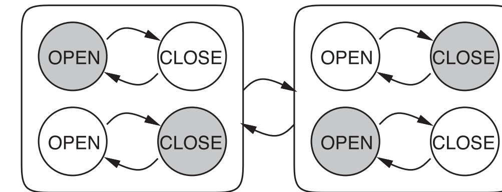

図 11　弾性バリアの状態遷移
Fig. 11　State transition of elastic barrier synchronization.

ことはできないが，逆に 1 つのスレッドが複数の遷移を起こすことは（他のスレッドとの干渉に注意したうえで）問題なく行えることに注意．

これを実装したものを**図 12** に示す（初期化などは略し，弾性バリアの入口と出口の関数のみを掲載した）．このコードの複雑さが図 9 と比べて大差ないことは，抽象状態同期に基づく同期記述の柔軟性を示すものだと考える．

4.2　バリアの性能評価

バリアについても前節と同じ環境でマイクロベンチマークによる時間計測を行った．ベンチマークは，N 個のスレッドを生成し，それが D 回反復してバリア同期を行い，全体の所要時間を計測するものであり，スレッド数を複数通り変え，バリア通過総数 $N \times D$ は一定になるようにして計測を行った．計測対象としたバリアの実装は次の 3 種類である（このほか，3 の改良版の試行である 3b と 4 も計測しているが，これらについては 5 章で述べる）．

1　標準のバリア `pthread_barrier`．
2　標準の排他ロック `pthread_mutex` と条件変数 `pthread_cond` を用いて実装したバリア（図 10）．
3　抽象状態同期を用いて実装したバリア（図 9）．

計測結果を**表 8**，**表 9**，**表 10** に示す．単位はすべて秒，かっこ内は標準偏差，表題中の表記（$N \times D$）は，スレッド数と各スレッドのバリア通過回数を表す．

結果を見ると，システム標準のバリアは条件変数版より高速であり，両者の実装が同一でないことが分かる．ソースを見ると，バリア変数の実装は条件変数を使わずにシステムの待ち合わせ機能を直接呼んでいる．具体的には，バリアは「何回目の同期か」を示すカウンタを持ち，最初の $N-1$ 個のスレッドはこのカウンタの番地をウェイトチャネル値として待ちに入り，最後のスレッドは $N-1$ 個のスレッドを直接起こす．通常の条件変数の broadcast では「何個起きたか」を管理するため，各スレッドが起きた後で排他ロックを獲

© 2008 Information Processing Society of Japan

```
void elastic_barrier_enter() {
  ast_mutex_enter(&mutex1, S_OPEN);
  if(--count1) {
    ast_mutex_exit(&mutex1, S_OPEN);
  } else {
    count1 = init;
    ast_mutex_exit(&mutex1, S_CLOSE);
    ast_mutex_enter(&mutex2, S_CLOSE);
    ast_mutex_exit(&mutex2, S_OPEN);
  }
}
void elastic_barrier_leave() {
  ast_mutex_enter(&mutex2, S_OPEN);
  if(--count2) {
    ast_mutex_exit(&mutex2, S_OPEN);
  } else {
    count2 = init;
    ast_mutex_exit(&mutex2, S_CLOSE);
    ast_mutex_enter(&mutex1, S_CLOSE);
    ast_mutex_exit(&mutex1, S_OPEN);
  }
}
```

図 12　抽象状態同期による弾性バリア
Fig. 12　Elastic barrier synchronization with AST-sync.

表 8　バリアの性能比較；スレッド数小（20×10000）
Table 8　Performance of barrier synchronization ; small number of threads (20×10000).

実装の種類	ユーザ時間	システム時間	経過時間
1 barrier	0.112(0.031)	4.976(0.384)	1.459(0.173)
2 cond	0.427(0.063)	6.805(0.908)	2.193(0.341)
3 ast_mutex	0.296(0.053)	5.720(0.119)	3.366(0.036)
3b ast_mutex'	0.217(0.049)	8.940(0.121)	4.466(0.056)
4 enter-exit	0.264(0.039)	13.800(0.346)	4.625(0.109)

得してデータをチェック/更新しているが，バリアは単に「カウンタを読み出して予期どおりの値かチェックする」だけなので排他ロックが省略でき，そのために効率が良くなっている．

これに対し，抽象状態同期版の実装では，起きた後はロックを獲得しているという基本設計であり，そのロックを改めて解放する必要もあるため，全般に条件変数版よりさらに遅くなっている．これは，バリアのような単純な同期方式の場合，抽象状態同期のような汎用性は性能の足枷になってしまうためだともいえる．ただし，前述の弾性バリアなど，アプリケーションの必要に応じて仕様を調整できる柔軟性を含めて考えれば，抽象状態同期版も一定の使途はあると考える．

表 9　バリアの性能比較；スレッド数中（200×1000）
Table 9　Performance of barrier synchronization ; moderate number of threads (200×1000).

実装の種類	ユーザ時間	システム時間	経過時間
1 barrier	0.116(0.035)	4.566(0.315)	1.294(0.152)
2 cond	0.433(0.056)	7.583(0.908)	2.426(0.306)
3 ast_mutex	0.355(0.057)	8.232(0.526)	4.122(0.207)
3b ast_mutex'	0.229(0.044)	9.149(0.192)	4.589(0.065)
4 enter-exit	0.291(0.039)	13.011(0.343)	4.210(0.070)

表 10　バリアの性能比較；スレッド数大（2000×100）
Table 10　Performance of barrier synchronization ; large number of threads (2000×100).

実装の種類	ユーザ時間	システム時間	経過時間
1 barrier	0.190(0.037)	5.940(0.289)	1.743(0.091)
2 cond	0.657(0.086)	15.329(1.185)	5.275(0.421)
3 ast_mutex	0.623(0.061)	14.711(0.201)	6.600(0.049)
3b ast_mutex'	0.330(0.050)	9.938(0.395)	5.433(0.187)
4 enter-exit	0.375(0.059)	11.686(0.413)	4.517(0.114)

弾性バリアについては，FreeBSD を含め，広く使用されているオペレーティングシステムでこれをシステム標準として実装しているものはない．このため，今回は計測による定量評価は行わなかった．

定性的に検討すると，システム内部で弾性バリアを実装する場合，通常のバリアと比べて入口と出口の 2 回システムコールが必要となる分は所要時間増となると思われるが，それ以外にオーバヘッドが増加しそうな要因はないと考える．条件変数でも同様に，入口と出口の両方でロック獲得が必要になる分だけオーバヘッドが増加すると予想される．一方，抽象状態同期によるバリアではすでにカウンタの保護と待ちの 2 つのロックを使用しているので，それを弾性バリアに変更しても所要時間は同じであり，その分だけ両者との差は縮まるものと考える．ただし，次章で述べる enter-exit ペアの最適化は弾性バリアでは（出口でもカウンタをガードする必要があるため）適用できない．

© 2008 Information Processing Society of Japan

5. 抽象状態同期の性能改良

5.1 FreeBSD の同期機構

前章までで述べたように，カーネル版抽象状態同期の性能はおおむね条件変数を用いた実装と同程度である．条件変数を用いた同期コードが複雑で間違いやすく，このため性能を犠牲にしても条件式を反復評価する実装（CCR 相当）が広く使われていることを考えれば，この性能は悪いものではない．しかし，先に開発したカーネル版の実装は素朴なものであり，性能改良の余地がある．

この点について議論する前に，実装のプラットフォームである FreeBSD の同期機構について簡単に説明する．FreeBSD の同期機構はユーザモードで動く部分（以下ユーザランドと記す）とカーネル内コードの協調により実現されている．まず，mutex（排他ロック）はユーザランド内のフラグを compare & exchange 命令でアトミックにテストし，ロックが空いていればそのまま獲得するが，空いていなければカーネルコールにより実行を放棄する（待ち状態に入る）．ロックを解放するときも同じ命令でアトミックに状態を調べ，待っているスレッドがあればカーネルコールにより再開を依頼する．これにより，待ちをともなわないロックの獲得/解放はカーネルの介在なしに行える．

FreeBSD の同期機構の基本設計として，スレッドが待ち状態から再開してユーザランドに戻ったときに，待っていた条件（ロックが獲得可能になったなど）は必ずしも保証されない（条件が満たされないのに起きてしまうことを bogus wakeup と呼んでいる）．このため，再開したスレッドは必ず再度ロック獲得操作を行い，成功しない場合は再度待ちに入る必要がある．

カーネル内での待ち合わせは，配列 `umtx_chain[]` 配列中のどれかのキューに自スレッドをつないで実行を放棄する (スケジューリングキューから自スレッドを外す) ことで行われる．使用する配列エントリは前述のとおり，ウェイトチャネル値からハッシュ関数により計算される．

条件変数やバリアの実装も，カーネルコールによりこのキューにスレッドをつないで待ち，起こすときにこのキューからスレッドを取り出してスケジューリングキューに戻すことで行っている．カーネルコールの引数で戻すスレッドの数が指定でき，これにより「1 つ起こす」「全部起こす」のいずれもが行える．なお，これら全般において bogus wakeup があるため，起きた後ユーザランド側で再度チェックする必要がある．たとえば，バリアの場合は前述のように「何サイクル目の同期か」を示すカウンタを保持し，起きたときにこのカウンタが進んでいなければまだ全員揃っていないと分かるので，再度待ちに入るように実装されている．

5.2 性能改良のための試み

抽象状態同期の性能を向上させるための基本的な方向として，次の 2 つが考えられる．

- カーネルコールの回数やユーザランドでの操作を削減する．
- カーネル内での操作のオーバヘッドを軽減する．

前者を実現する方法の 1 つとして，抽象状態をガードする mutex の管理方法の改良を実現した．具体的には，抽象状態同期では，排他領域の出口で `ast_mutex_exit` を呼ぶことで新しい抽象状態が設定され，その状態で起きられるスレッドが（あれば）1 つ選ばれて起こされ，起きた状態では排他領域内にいる．従来はこの部分の実現は，起こす側が mutex を解放し，起きた側が改めて mutex を獲得していたが，この部分を手直しして起こす側から起きた側に mutex を受け渡せるようにした．

この計測結果を表 3〜7 と図 6〜7 (3b，3b') および表 8〜10 (3b) に記載してあるが，これにより読み書きロックでは読み書き比が 1 : 1 の場合には所要時間が 1 割以上減少し，そのうちスレッド比が 100 : 1，10 : 1 の場合は条件変数による実装より優る結果となった．ロックを保持したままの場合（3b'）は，リーダスレッド数が多いところでは改良前と同程度の性能だが，少ないところではいくぶん速くなっている．

バリア同期については，スレッド数が少ないところでは元の版よりも性能が低下していることが分かる．しかし，スレッド数が多くなってもオーバヘッド増加の度合いが小さいため，スレッド数 2000 では改良版の方が性能が高く，条件変数版と同程度になっている．

さらに別の視点として，図 3 や図 9 のコードを見ると，`ast_mutex_enter` や `ast_mutex_exit` の呼び出しが複数個連続している箇所がある．これは，排他領域に入った直後に内部状態を調べればその先どのように状態遷移してゆくべきか決ってしまうような場合に見られる．そのようなコードをユーザランドから 1 つずつ呼び出してゆかなくても，カーネル内で行えないか，ということが考えられる．ただし，bogus wakeup に対処して待ちを再試行するのにはユーザランド側の状態が必要であり，そのまま実現しても高速化につながらない面がある．

1 つの試みとして，バリア同期の実装（図 9）における`/*☆*/`の 2 行については，入った後状態を変えずに出てくるので実装上問題が少なく，また 1 回の状態設定で複数のスレッドが起こせるため，このような対（状態を変更しない enter/exit の対）を 1 回のシステムコールで行う実装を行ってみた．具体的な実装方法は次のとおり．

© 2008 Information Processing Society of Japan

- 「状態を変更しない enter-exit ペア」を表す待ち合わせを表すシステムコール操作を追加し，これが使える場面ではこれを呼び出す．
- そのようなものの 1 つがロックを獲得したときは，パラメータが同じ他のペアを一括して起こし，さらに起こされた側のシステムコールの返値として一括して起こされたことを示す特別な値を戻すようにする．
- ユーザランドに復帰したとき，その特別な値が返されていれば bogus wakeup ではありえないのでそのまま先へ進む．他の値であれば bogus wakeup と思われるのでこれまでどおり状態チェックに戻る．

この実装の計測結果は表 8〜10 (4) に記載している．これを見ると，スレッド数が 20 と 200 では条件変数による実装より遅いが，スレッド数が多くなっても経過時間が増加しないため，スレッド数が 2000 では条件変数による実装よりも良い性能を示している．それでも組み込みのバリア同期より遅いが，組み込みのバリア同期と部分的に異なる動作を必要とする場合には抽象状態同期から同期操作を組み立てることが有用であると考えられる．

6. 議論とまとめ

近年のプロセッサでは，単一 CPU コアの性能向上は頭打ちとなり，代わってマルチコアやハイパースレッドなどの技術に基づく複数スレッド動作によるスループット向上が指向されている．このため今後，1 つのプログラム内部での共有メモリモデルを用いたマルチスレッド実行がハードウェア資源の有効活用のために重要になっていくと思われる．このような背景から，さまざまな場面で記述しやすく，なおかつ効率良くスレッド間の同期を行える同期機構が求められている．

共有メモリモデル向けの同期機構の最も初期のものは，Dekker のアルゴリズムに代表される，メモリアクセスの排他性に基づくものであるが，きわめて記述性が悪いことが知られていた．より記述性の良いプリミティブとしてはセマフォ[2] などが提案されたが，これはカウンタに基づく同期に限定されていた．

これに対し，排他領域の出入りをモジュールの形で記述し，待ち合わせに条件変数を使うことを提案したのが Hoare のモニタである[6]．条件変数による同期は汎用性があるうえに比較的効率も良く，現在でも POSIX スレッドや Java などで広く採用されている．

しかし，本論文でも繰返し述べたように，条件変数を用いた待ち合わせは依然として記述の難しさという弱点があった．このため，より記述性の良い同期機構にが探求され，条件つき排他領域（CCR）が考案された[1),5]．CCR は排他領域を任意の条件式でガードできるため汎用性があり記述しやすく，正しさが確認しやすいという利点がある．

もともとの CCR は専用の構文を持つ言語の一部として提案されたが，今日ではそのような専用言語ではなく，モニタと条件変数の上で以下のパターンによって実現され使用されることが多い（たとえば文献 9) など）．

```
pthred_mutex_lock(&mutex);
while(!(条件式))
  pthread_cond_wait(&cond, &mutex);
/* CCR 内部の動作 */
pthread_broadcast(&cond);
pthread_mutex_unlock(&mutex);
```

この場合，条件式の評価が排他領域内で行われること，待っているスレッドのうち条件が真になっていないものも毎回起こされて条件を再評価することから，どうしても性能は低くなる[11]．

別のアプローチとしてソフトウェアトランザクショナルメモリ（STM）を楽観的並行制御と組み合わせる方法がある[3),4]．この方法は記述性，性能ともに悪くないが，排他領域内のメモリアクセスすべてを STM 操作として行う必要があるため，既存言語のライブラリ API として用いるのには適していない．

性能上の問題が最も少ないのは排他領域と条件変数を組み合わせて使用する方式であるが，読み書きロックの記述例でも分かるように，具体的な各種の応用に合わせた条件同期においてコードが複雑になりやすく，誤りの原因になりやすい．このため，読み書きロックやバリア同期のような多く使われる同期のパターンに対しては専用のライブラリ実装が用意されている．

しかし，そのような専用実装はその特定の場面に限定されたものであり，アプリケーションの必要に応じた調整は行えない．もし，条件変数と同程度にプリミティブでオーバヘッドが少なく汎用性がありながら，記述性の高い条件同期機構があれば，専用のライブラリ実装の代わりにプリミティブから必要な同期を組み立てることで，各アプリケーションの必要に応じた調整が可能になる．

抽象状態同期は「待つ」「起こす」という制御を直接扱うのではなく，待ち合わせる「条件」を扱うという点で，条件変数よりも CCR に近い．ただし，CCR が「条件式」を記述するのに対し，抽象状態同期では条件を「有限個の抽象状態のどれにあるか」という外部化/抽象化された形で扱うため，条件式の評価という実装上難しい問題を避けて，OS カーネル内部などさまざまな箇所で条件の成否を効率良く判定でき，これによって効率の良い実

© 2008 Information Processing Society of Japan

装が可能になっている．

本論文では，抽象状態同期を用いて読み書きロックとバリア同期を実装し，性能評価を行った結果，これらの機構を抽象状態ロックによって記述するのは容易であり，性能も条件変数を用いた実装とあまり違わないことを示した．

今後の課題としては，抽象状態同期の API や実装をさらに改良してゆくことと，その有用な活用方法を探求してゆくことを考えている．

謝辞　情報処理学会論文誌：プログラミング編集委員会の皆様，および査読者様には，論文の内容を充実させ，記述を改良するための多くのご示唆をいただきました．ここに感謝します．

参考文献

1) Hansen, P.B.: A comparison of two synchronizing concepts, *Acta Informatica*, Vol.1, No.3, pp.190–199 (1972).
2) Dijkstra, E.W.: The structure of the "THE" — multiprogramming system, *CACM*, Vol.11, No.5, pp.341–346 (1968).
3) Harris, T. and Fraser, K.: Language Support for Lightweight Transactions, *Proc. OOPSLA'03*, pp.388–402 (2003).
4) Herlihy, M.: A Methodology for Implementing Highly Concurrent Data Objects, *TOPLAS*, Vol.15, No.6, pp.745–770 (1993).
5) Hoare, C.A.R.: Monitors: Toward a theory of parallel programming, *International Seminar on Operating System Techniques*, *A.P.I.C. Studies in Data Processing*, Vol.9, pp.61–71, Academic Press (1972).
6) Hoare, C.A.R.: Monitors: An operating system structuring concept, *CACM*, Vol.17, No.10, pp.549–557 (1974).
7) 久野　靖，大木敦雄：抽象状態同期に基づく並列オブジェクト指向言語 p6，情報処理学会論文誌，Vol.38, No.3, pp.563–573 (1997).
8) Kuno, Y.: Solving Inheritance Anomaly Problems by State Abstraction-Based Synchronization, *Object-Oriented Parallel and Distributed Programming*, Bahsoun, J.P., Baba, T., Briot, J.P. and Yonzezawa, A. (Eds.), pp.167–186, Hermes (2000).
9) Lea, D.: *Concurrent Programming in Java: Design Principles and Patterns*, Addison-Wesley (1999).
10) 松本　尚：一般化されたバリア型同期機構，情報処理学会論文誌，Vol.32, No.7, pp.886–896 (1991).
11) 大木敦雄，久野　靖：スレッドライブラリへの抽象状態同期の導入，情報処理学会論文誌：プログラミング，Vol.47, No.SIG 11(PRO 30), pp.28–37 (2006).
12) Ohki, A. and Kuno, Y.: State Abstraction-based Synchronization for Thread Libraries, *International Journal of Computer Sciences and Network Security*, Vol.7, No.6, pp.36–44 (2007).

(平成 20 年 2 月 17 日受付)
(平成 20 年 5 月 23 日採録)

大木　敦雄（正会員）

1957 年生．1983 年筑波大学大学院理工学研究科修士課程修了．同年静岡大学理工学部情報工学科助手．筑波大学大学院経営システム科学専攻助手，講師を経て，現在，同准教授．プログラミング言語，プログラミング環境，オペレーティングシステム，ユーザインタフェース等に興味を持つ．著書に『入門 FreeBSD』（アスキー）等がある．日本ソフトウェア科学会会員．

久野　　靖（正会員）

1956 年生．1984 年東京工業大学大学院理工学研究科情報科学専攻博士後期課程単位取得退学．同年東京工業大学理学部情報科学科助手．筑波大学大学院経営システム科学専攻講師，助教授を経て，現在，同教授．プログラミング言語，オペレーティングシステム，情報教育，ユーザインタフェース等に興味を持つ．著書に『Java によるプログラミング入門』（共立），『UNIX の基礎概念』（アスキー）等がある．日本ソフトウェア科学会，ACM，IEEE-CS 各会員．

© 2008 Information Processing Society of Japan