Vol. 47 No. SIG 11(PRO 30) 情報処理学会論文誌：プログラミング July 2006

スレッドライブラリへの抽象状態同期の導入

大木敦雄† 久野靖†

スレッドライブラリは共有メモリ型マルチプロセサシステムにおいて並行プログラミングを行う標準的な手法の1つである．現在世の中に普及しているスレッドAPIであるPOSIX Thread準拠のものをはじめとする多くのライブラリでは，スレッド間の条件同期の方法として，条件変数（condition variables）を採用している．しかし，条件変数を用いたコード記述は読解しにくく誤りの原因となりやすい．このため，条件変数を1つだけ用意し，すべての条件同期をこの1つの条件変数で待ち合わせる方法が使われることが多い．Java言語における標準の待合せ機構も実質的にはこれに相当する．しかし，この方法ではすべてのスレッドが条件イベントごとに実行を再開し，自分が続行できる条件でない場合は再度待合せに入るため，オーバヘッドが大きい．本論文では，筆者らが考案した抽象状態同期と呼ばれる同期機構をスレッドライブラリAPIに組み込むことを提案する．抽象状態同期では，排他領域により守られているデータ構造の状態を数個の状態に抽象化して考え，そのどれとどれに相当する場合に各操作が領域内に入れるかを指定することで読みやすく効率的なコードを作成できる．本論文では具体的なAPIの提案，その試験実装，性能評価，コードの書きやすさなどについても報告する．

Incorporating State Abstraction-based Synchronization into Thread Libraries

Atsuo Ohki† and Yasushi Kuno†

Thread libraries are widely used for concurrent programming on shared-memory multiprocessors. On POSIX Thread and other well-known thread libraries, condition variables are used for conditional inter-thread synchronization. However, code with multiple condition variables are difficult to understand and error-prone. Hence, programming style with single condition variable is widely used. Standard synchronization scheme on Java programming languages is effectively designed likewise. However in this scheme, all threads are woken up on each event and have to check for their continuing conditions repeatedly, leading to large overhead. In this paper we propose to incorporate state abstraction-based synchronization into thread libraries. In our scheme, state of the data structure guarded by an mutex is mapped to small number of "abstract states." Every mutex acquisition are guarded by a set of abstract states on which the processing can continue. We also report our API proposal, along with experimental implementation, performance evaluation and some discussion on code readability.

1. はじめに

マルチプロセッサシステムや分散システムの普及とともに，プログラミングにおいて並行性を扱うことの必要性も高まっている．しかし，今日主流である手続き型言語/オブジェクト指向言語の範囲内では，並行性を扱うための機構，特に排他制御と同期を扱うための機構は1970年代からほとんど変化していない．

具体的には，共有メモリ環境の場合，POSIX Thread API[5]に代表される，マルチスレッドに排他領域を構成するためのロック（ないしそれを言語の構文と合体させたモニタ）と，条件に基づく待合せを行うための条件変数とを組み合わせる方式[4]が，実験的な言語や特殊な環境を除けば，圧倒的に広く使われている．

これは，CやC++のような最初から並行性を前提とした設計を行っていない言語に対し，後から最小限の変更で並行性を導入する場合，ライブラリの呼び出しだけでほぼ実装可能なロック+条件変数方式が好都合だったためと予想される．また，この方式が一定の普及を見たため，最初から並行性を言語設計に取り込んでいる言語でも，Java[6]のようにロック（モニタ）+条件変数方式を前提としているものがある．

しかし，条件変数を用いたスレッドの待合せには，(1) 構造化されていないため繁雑で間違いを犯しやすい，(2) 性能上のペナルティを強いられる場合がある，

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

(3) データ構造が複雑な場合，最大限の並行性を得るのは容易でない，という弱点がある．

また，CCR (conditional critical region，条件つき排他領域) などを用いて上記の弱点を解消する方式も探求されているが[1]，言語の大幅な変更が必要だったり，実装が難しく採用に制約があったりするなど，別の弱点をかかえることになっている．

筆者らは，複数の実行単位間の並行制御のための新たな機構として抽象状態同期 (abstract state synchronization，AST) を提唱し，その並行オブジェクト指向言語への組み込みについて研究してきた[7),8)]．本論文では，抽象状態同期をマルチスレッドライブラリの同期機構として組み込むことで，上述の問題点を大幅に改善できることを示す．

まとめると，スレッドライブラリに抽象状態同期を組み込むことの動機ないし必要性は，次のとおりである．

(a) 条件同期は並行プログラミングにおける基本的な機能であるにもかかわらず，既存の機構は性能面/記述性/実装しやすさのいずれか 1 つ以上において弱点があった．これを克服したい．

(b) 現実の並行プログラミングはその大部分が，特殊な言語ではなく，既存言語とスレッドライブラリの組合せで行われている．このような実用の場面において上記 (a) を実現したい．

また，その適用可能な場面は「並行プログラミングにおいて条件に基づく同期を必要とするすべての場面」であるといえる．

本論文における評価では C 言語を使用しているが，他の言語でも原理的には同一のライブラリ API が利用できる (Java のように旧来の API が標準オブジェクトに組み込まれている場合にはこれを変更する必要がある)．

以下，2 章では，既存の同期機構が持つ問題点とその背景について分析し，その克服に向けた他の研究についても紹介する．3 章では，抽象状態同期の概念について解説し，それをスレッドライブラリの同期機構として組み込む方式について検討する．4 章では，3 章で述べた設計の実装および性能評価について述べ，5 章でまとめを行う．

2. 既存の同期機構とその問題点

2.1 ロック+条件変数同期

共有メモリ環境における並行プログラミングでは，共有データへの排他的なアクセスを保証する機構が不可欠である (実行主体が排他的なアクセスを確立している範囲を排他領域と呼ぶ)．これは，複数の並行実行主体が競合的にデータをアクセスした場合，その整合性が保証されないことを考えれば当然である．その基本的なモデルは，排他領域の入口でロックを獲得し，出口でロックを解放する，というものである☆．

具体的なロック獲得/解放のコード上での表現方法としては，明示的にロックの獲得/解放を記述する方法と，言語上での特定の構文範囲を排他領域に対応させる方法とがある．前者は特に並行記述を前提としない言語でもライブラリ呼び出しによって実装できるが，コードの記述性の観点からは後者が優れている．後者のうちでも，構文範囲として競合的アクセスから保護されるデータとそれを操作する手続きを合わせたものを用いる場合をモニタ[4)] と呼ぶ．現在では，オブジェクト指向言語のクラス記述をこれに対応させることが多い．

競合アクセスから保護されるデータに対し，それぞれの実行主体が必要とする操作が行えるか否かは，一般的にはデータの状態によって変わってくる．たとえば有限バッファの場合，put() はバッファが満杯であれば行えないし，get() はバッファが空であれば行えない．ロックを素直な形で用いる場合，これらの状態のいずれであるかは，ロック獲得後にデータを参照してはじめて分かる．そしてその結果，「行えない」状態だった場合はいったんロックを解放し，他の実行主体が状態を変化させるのを待ち，「行える」状態になった後で再度ロックを獲得して操作を行う必要がある．

Hoare のモニタ[4)] では，このような場合には条件変数 (condition variable) による待合せを行う．有限バッファの例題で関係する部分だけを示す．

```
type buffer(T) = monitor;
  ...
    nonfull, nonempty : condition;
prodecure put(p : T);
  begin
    if size = N then nonfull.wait;
    {有限バッファにデータを格納}
    nonempty.signal;
  end;
procedure get(var it : T);
  begin
    if size = 0 then nonempty.wait;
    {有限バッファからデータを取り出し}
```

☆ 単純なロックでは競合時に待ちが生じるが，データ構造の工夫による待ちなし (wait-free) の実装に関する研究もなされている[2)]．

```
    nonfull.signal;
  end;
begin {初期設定} end;
```

条件変数の実体は実行主体が待合せを行うためのキューであり，「条件変数.wait」によって待ち状態となった実行主体は，対応する「条件変数.signal」によって再開させられる（1つの signal はたかだか1個の実行主体を再開する．待ち合わせている実行主体がない場合は signal は何の効果ももたらさない）．個々の条件変数はひらたくいえば「個々の待ち合わせたい条件」ごとに用意することになる．上の例は Hoare のモニタの構文を用いたが，C/C++と POSIX Thread を用いる場合も，たとえば次のようにライブラリ呼び出しを用いて，同等の働きを実現できる☆：

```
void put(T p) {
  pthread_mutex_lock(&mutex);
  while(size == N)                    /* ☆ */
    pthread_cond_wait(&nonfull, &mutex);
  /* 有限バッファにデータを格納 */
  pthread_cond_signal(&nonempty);
  pthread_mutex_unlock(&mutex);
}
```

この方式は効率良く実装できるが，以下の弱点がある：

- どの状態でどの条件変数を用いて待機/再開を行うかの指定は繁雑であり，間違いを犯しやすい．
- 複数の待合せ条件にオーバラップがある場合に対応できない．たとえば「条件 A を待つ」「条件 A または B を待つ」の両方が必要な場合，これらを別個の条件変数にすることはできず，次節で述べる方法を併用することになる☆☆．

2.2 Java のモニタと反復判定

Java[6] では1つのオブジェクトが1つのロックを持ち，synchronized 指定のメソッドは入口/出口でロックの獲得/解放を行うという形で，モニタを言語に組み込んでいる．ただし，Java では条件変数が1つのオブジェクト（=ロック）につき1つしか用意されていないため，複数の異なる条件を待ち合わせる実行主体が1つのキューに混在することになる．このため，モニタ内でデータの状態を変更したときは notifyAll() によりすべての実行主体の待ちを解除し，各実行主体はそれぞれが行いたい操作に必要な条件のが成立したかどうかを次のように繰り返し調べる必要がある：

```
class Buffer {
  ...
  synchronized void put(T p) {
    while(size == N) wait();
    // 有限バッファにデータを格納
    notifyAll();
  }
  synchronized T get() {
    while(size == 0) wait();
    // 有限バッファからデータを取り出し
    notifyAll();
    return data;
  }
}
```

このような方式を反復判定と呼ぶことにする．反復判定は任意の条件式に基づき待合せ条件をチェックできるという一般性があるが，その反面，次の問題点がある：

- 待機中の実行主体を再開させる際に，すべてを起こしてそれぞれに条件の再検査を行わせることになり，性能面で不利である．
- ループによる条件のチェック，notifyAll() の使用などの慣習を守らないことによるバグが混入しやすい．
- 次にロックを獲得する実行主体はスケジューリングのタイミングによって決まるため，fairness を保証できない．

2.3 条件つき排他領域（CCR）

ここまでに見てきたように，条件変数は各実行主体ごとにそれぞれ特定の条件が成り立つまで排他領域内の実行開始を遅延させるために使われるので，そのような条件と排他領域を構文として一体化させることは自然である．これを条件つき排他領域（conditional critical region，CCR）[3] と呼ぶ．排他領域に入るための条件式部分は「ガード」と呼ばれることが多い．

CCR では，実行主体はまず排他的に条件式を評価し，その結果が真である場合に排他領域内の操作に進む．結果が真でない場合は待ち状態となり，他の実行主体が排他領域を出た後で再度条件式の評価に戻る．したがって，その動作は反復判定と同様のものになり，

☆ Hoare の条件変数では，signal により起こされた実行主体は優先的に（他のモニタに入ろうとしている実行主体より先に）モニタの実行権を獲得するのに対し，POSIX Thread API ではそのような保証がない．このため，入口での条件チェックを if 文の代わりに while 文にして，起きた後で再度条件をチェックする必要がある．

☆☆ ライブラリ API に「複数の条件変数を and/or などで組み合わせて待つ」機能が含まれていれば，A と B に対応する2つの条件変数で扱えるが，このような機能を持つスレッドライブラリは一般的でない．

その弱点は（専用の構文により書き方を守らないという問題は解決されるとしても）性能面での不利ということになる．

文献 1) では，トランザクションメモリのソフトウェアによる実装（software transactional memory, STM）を排他制御に用いた待ちなしの CCR の実装を Java に組み込み，その性能評価において，注意深く実装された既存の Java による実装と同等以上の性能を得たとしている．ただし，STM を用いる場合，排他領域内での変数の読み書きはすべて STM の操作を通じて行う必要があるため，コンパイラの変更が必要であり，通常のライブラリとして実現することは難しい☆1．

3. 抽象状態と抽象状態同期

3.1 抽象状態

オブジェクト指向言語では，1 つのオブジェクトがさまざまな状態をとることができる☆2．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，すべての状態の違いを区別することは必ずしも必要でない．

たとえば，空のスタックに “a” を積んだ状態と “b” を積んだ状態は異なる状態だが，スタックを扱う側にとっては，値を 1 つ取り降ろしてきて使えればよいのなら，両者を区別する必要はない．また，“a” と “b” の 2 つを積んだ状態を考えると，取り降ろせる個数が先とは違うが，スタックを用いる多くのコードは「空でなければ取り降ろしてくる」ため，“a” だけを積んだ状態と区別する必要がない☆3．したがって，スタックの状態を外部から参照する場合，興味があるのは {`empty`, `mid`, `full`} のいずれの状態にあるかの区別だといえる．このように，オブジェクト内部の状態を抽象化した少数の状態に対応づけたものを**抽象状態**と呼ぶ．

通常のオブジェクト指向言語では，プログラマが ad hoc に `isEmpty()`, `isFull()` などの述語メソッドを用意することで実質的に抽象状態の情報を提供しているが，(1) ad hoc なので何が状態であるか分かりにくく情報の系統的な活用が難しい，(2) メソッド呼び出しのオーバヘッドがある☆4，(3) 抽象状態が変化していなくてもそのつどメソッド内で条件式を評価する無駄が生じる，という弱点がある．

そこで筆者らは，オブジェクトがとりうる抽象状態の集合を列挙値として明示的に宣言し，特定のインスタンス変数にその現在値を保持することで，抽象状態値を明示的に表現することを提案した．抽象状態値はオブジェクトの状態を変化させるメソッドが本来の動作と一緒に更新し，オブジェクト外部からは読み出しのみ可能とする．これにより，条件式の評価を必要最低限にとどめることができ，外部からも効率良く抽象状態を参照できる．

3.2 抽象状態同期

条件同期はオブジェクトの状態が特定のものになるまで実行を遅延することなので，この状態と抽象状態を対応させて扱うことは自然である．筆者らはそのような機構を提案し，抽象状態同期と名づけた．

抽象状態同期では，オブジェクトごとに抽象状態を保持することは前節と同様だが，加えて排他領域（いずれかのオブジェクトに付属）ごとにその入口に排他領域の実行を許す抽象状態の集合を指定し，排他領域に入ろうとする実行主体はオブジェクトの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．排他領域に入った実行主体はその出口において次の抽象状態を設定し，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，抽象状態の数がハードウェアの 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]．

3.3 スレッドライブラリへの抽象状態同期の導入

筆者らの過去の研究[7),8)] は，抽象状態同期を採り入れた新しいプログラミング言語を開発するというアプローチを探求した．本論文では別のアプローチとして，現実の並行プログラムで主流となっている，旧来の言語とスレッドライブラリの組合せに抽象状態同期を導入することを通じて，前節までに述べたスレッドライブラリ方式の同期機構に関わる弱点の解消を試みる．

具体的には，1 つのロックが 1 つのオブジェクトに対応するものと考え，ロックのデータ構造中にそのオ

☆1 厳密にいえば，すべての変数の読み書きにおいて STM の呼び出しをプログラマが挿入すれば実現可能だが，現実的でない．

☆2 書き換え不能なオブジェクトの場合，オブジェクトが作られた時点でさまざまな状態のうちの 1 つが選ばれ，以後変化しないものと考えることができる．

☆3 区別する必要があるような応用もありうるが，その場合は両者の違いを抽象状態としても区別するようなスタックの定義を行う．

☆4 インライン展開などの最適化は可能である．

ブジェクトの抽象状態を保持させる．そしてロック獲得時にライブラリ呼び出しの引数として，排他領域に入ることが許される抽象状態の集合を渡す（ここでは前節で述べた方法により，各状態を1ビットで表し，1語のマスクで抽象状態の集合を表すものとする）．また，排他領域を出るところで新しい抽象状態を設定する．抽象状態の初期値は，オブジェクトの初期状態に対応したものをロック初期化の際に指定する．

これらを組み合わせると，先の有限バッファに格納する操作は初期値を S_EMPTY に設定するものとして，次のように記述できる．

```
void put(T p) {
  ast_mutex_enter(&mutex, S_MID|S_EMPTY);
  /* 有限バッファにデータを格納 */
  if(バッファが満杯)
    ast_mutex_exit(&mutex, S_FULL);
  else
    ast_mutex_exit(&mutex, S_MID);
}
```

対応する取得操作は次のようになる．

```
void get(T *p) {
  ast_mutex_enter(&mutex, S_FULL|S_MID);
  /* *p に有限バッファからデータを取り出し */
  if(バッファが空)
    ast_mutex_exit(&mutex, S_EMPTY);
  else
    ast_mutex_exit(&mutex, S_MID);
}
```

この枠組みでは，ast_mutex_enter を通過した後では対応するデータ構造はただちに操作可能な状態になっているので，条件をチェックしてリトライする操作は不要になる．ロックは ast_mutex_exit により解放するが，そのときロックを操作後の新しい抽象状態に遷移させる．

まとめると，enter から exit までの間は排他領域であり，他の実行主体が enter しようとした場合は待たされる．そして，exit したときには待っている任意の実行主体が排他領域に入れるのではなく，待っている実行主体のうち，設定された抽象状態に適合するものが1つ選ばれ排他領域に入れるものとする．この場合，状態の遷移のさせ方によっては飢餓が起きることもありうるが，それはプログラマの責任と考える．

4. 実　　装

筆者らは FreeBSD 6.0-RELEASE に付属するスレッドライブラリを改造して，前章で述べた API による抽象状態同期機構を実装した．FreeBSD にはスレッドライブラリとして，(1) ユーザレベルのみでスレッドを実装するもの（libc_r），(2) カーネルスレッドとユーザスレッドが1：1に対応するもの（libthr），(3) 複数のカーネルレベルスレッドが複数のユーザレベルスレッドに M：N 対応するもの（libpthread），の3つが標準で備わっている．今回はカーネルの最小限の改造で抽象状態同期を導入することをめざしたため，(2) をもとに改造を行った．

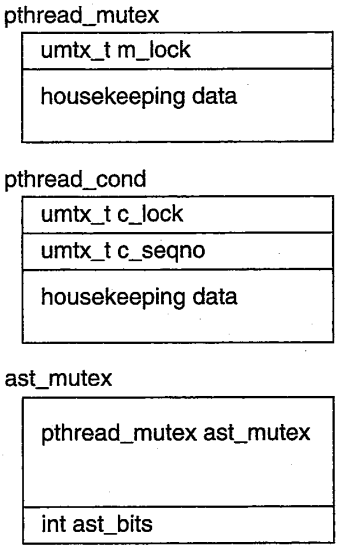

図1　ユーザ空間側のデータ構造
Fig. 1　Data structures in user-land.

今回改造した部分に関係するユーザ空間側のデータ構造を図1に示す．pthread_mutex はオリジナルのライブラリの mutex データ構造であり，その中にある umtx_t 型のデータはカーネル側のロック機構が使うデータ構造であり，このユーザ側データ構造の排他アクセスに使用する．pthread_cond は同じく条件変数のデータ構造であり，こちらはこのデータ構造の排他アクセスと，条件変数での待合せのときにカーネル内部のキュー構造（後述）に使用するため，2つのカーネルロック（umtx_t）を持つ．ast_mutex は抽象状態同期 API が使う新たなデータ構造であるが，中身はオリジナルの pthread_mutex と現在の抽象状態値を表す1語とからなる．

カーネル空間側のデータ構造を図2に示す．umtx_chain はハッシュ表であり，双方向連結リストのヘッダとなっている．mutex や条件変数の数が多くならなければ，1つのスロットは1つの mutex や条件変数で待っているスレッドだけの連結リストと考えてよい．連結リストの各要素には，オリジナルのカーネルではスレッド ID とユーザ空間側のカーネルロック（umtx_t データ構造）のアドレスが格納されてい

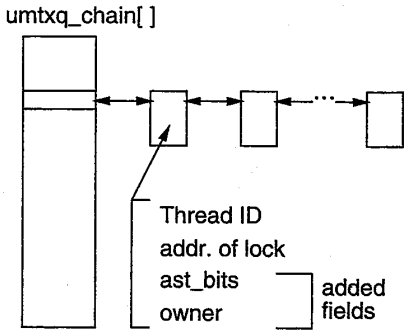

図 2 カーネル空間側のデータ構造
Fig. 2 Data structures in kernel land.

たが，今回の改造ではそれに次の 2 つのフィールドを追加した

- `ast_bits` — 待っている状態集合を表すマスク値.
- `owner` — 待っているスレッドが起こされるときに `umtx_t` のロックを獲得するのに使用する値.

これらのフィールドは抽象状態同期によって待ちに入った場合にのみ使用される.

FreeBSD 6.0 では，`umtx_t`（カーネル内ロック）の操作は `_umtx_op()` という 1 つのシステムコールのみを使用しており，そのパラメータでどのような操作をさせたいかを指定する．オリジナル版では操作として「ロックする」「解放する」「ロックを指定して寝る」「指定したロックで寝ているスレッドを指定個数起こす」の 4 つの操作が用意されていたが，これに次の 2 つを追加した.

- 状態マスクを指定してロックで寝る.
- ロックで寝ているスレッドの中から，状態値に適合するマスクを持つものを 1 つ起こす.

後者の実装は連結リストを先頭から順探索して見つかった最初のものを起こすという単純な実装だが，後の評価で示すように，スレッド数が数百程度では目立った性能上のペナルティは観察されなかった.

ユーザランドのライブラリでは，システムコール `_umtx_op()` に追加した機能の呼び出しとエラーチェックのみを行う.

変更したコード行数は約 100 行，改造に要した作業量は筆者の 1 人が作業して丸 3 日ほどであった（元のライブラリとカーネルの構造読解を含む)☆.

またその過程で，ポータブルな実装の検証用に POSIX Thread API を用いて同じ機能を実現したものも作成した（付録 2 に掲載）．こちらは内部で反復判定を行うので，性能的なメリットはない（記述性は良くなる）．両者を区別するため，以下では本来の実装を「改造版」，POSIX Thread API 上のものを「可搬版」と記すことにする.

なお，いずれの実装についても，既存のスレッド API との混在ないし共存は特に問題ない．すなわち，1 つのプログラム/スレッドにおいて，あるデータ構造をガードする部分では従来の mutex，条件変数などを利用し，別のデータ構造をガードする部分には本実装を用いたとしても，特に問題は起きない☆☆.

5. 評　　価

改造したスレッドライブラリを評価するため，複数の例題（ケース）を選び，改造したライブラリで抽象状態同期を使った実装と，それと同様の機能を従来の API で反復判定および（可能な場合）条件変数で実装したもとの比較を，性能面と書きやすさの両面から行った．具体的なケースおよびその要点は以下のとおり.

- 通常の有限バッファ．状態として {`empty`, `mid`, `full`} の 3 つがあり，スレッドは「`full` である間遅延」(`put` するスレッド),「`empty` である間遅延」(`get` するスレッド）の 2 つがある．通常の条件変数で記述可能なので，条件変数版，反復判定版，抽象状態版（可搬版，改造版）の 4 つで比較を行った.
- ウォーターマークつき有限バッファ．バッファに半分以上データが入っているかどうかを区別するため，状態が {`empty`, `midlow`, `midhigh`, `full`} の 4 つとなる．`put` するスレッドに 2 番目の種類として「`empty|midlow`」である間遅延」を追加．このようにすると，`put` する側の条件が重なりのある 2 つになるため，通常の条件変数では記述できない．したがってこちらのケースでは，反復判定版，抽象状態版（可搬版，改造版）の 3 つで比較を行った.

いずれのケースについても，待ちスレッド数が少ない場合と多い場合を比較するため，`put()` するスレッドと `get()` するスレッドが同数の場合と，`get()` するスレッドを 1 個だけにした場合で計測を行った.

以下では通常の有限バッファで `put` する側のスレッド数 N_w 個，各スレッドが `put` する回数 C_w 回，`get` する側のスレッド数 N_r 個，各スレッドが `get` する回数 C_r 回のとき（N_wxC_w:N_rxC_r）のように記す．さらにウォーターマークつき有限バッファでは，バッ

☆ これとは別に，後述するオリジナルの FreeBSD 6.0 の条件変数にあったバグを修正する作業に 1 週間ほどを要した.

☆☆ 当然ながら，1 つの mutex において条件変数による同期と抽象状態同期を混在させることは，データ構造が異なるため行えない.

表 1 通常の有限バッファ/待ち数小
(500×100 : 500×100)
Table 1 Finite buffer/a few waiting threads.

実装の種類	ユーザ時間	システム時間	経過時間
条件変数	1.273	12.277	7.16
反復判定	3.133	34.181	20.26
可搬版	3.080	33.637	19.91
改造版	0.601	8.269	6.41

表 2 通常の有限バッファ/待ち数大
(500×100 : 1×50,000)
Table 2 Finite buffer/many waiting threads.

実装の種類	ユーザ時間	システム時間	経過時間
条件変数	1.081	9.872	5.80
反復判定	2.713	28.684	16.99
可搬版	2.785	29.366	17.40
改造版	0.530	7.340	5.75

表 3 印つき有限バッファ/待ち数小
(250×100 : 250×100 : 500×100)
Table 3 Water-marked buffer/a few waiting threads.

実装の種類	ユーザ時間	システム時間	経過時間
反復判定	4.161	46.892	27.83
可搬版	3.934	44.130	26.19
改造版	0.594	8.261	6.38

表 4 印つき有限バッファ/待ち数大
(250×100 : 250×100 : 1×50,000)
Table 4 Water-marked buffer/many waiting threads.

実装の種類	ユーザ時間	システム時間	経過時間
反復判定	3.158	33.392	19.82
可搬版	2.924	30.858	18.27
改造版	0.539	7.432	5.83

ファの使用量が半分未満の場合にのみ put するスレッド数 N_v 個，これらが put する回数 C_v 回のとき（N_wxC_w:N_vxC_v:N_rxC_r）のように記す．有限バッファの容量は 10 とした．

計測はすべて Pentium II 350 MHz 2CPU 構成で行い，使用したコンパイラは GCC 3.4.4，最適化は -O4 を指定した．計測はすべて 100 回ずつ行い，平均値を示す（単位は秒）．なお，実験の過程で，FreeBSD 6.0-RELEASE オリジナルの条件変数の実装にバグがあり希にハングすることが分かったため，そのバグを除去する修正を行ったところ，多数回条件変数を使用する反復判定版と可搬版において性能が向上した．バグのないものを比較対象にするのが適切と考えたため，以下ではこのバグ除去バージョンに基づく計測結果を掲載する（「条件変数」「反復判定」「可搬版」の計測値が該当）☆

これらの結果（表 1，表 2，表 3，表 4）を見ると，条件変数版と抽象状態同期を意図どおり実装した改造版では所要時間が同程度ないし，改造版がやや速いことが分かる．これは，次の要因によるものと考えられる．

- ユーザ時間については，抽象状態同期ではユーザ側コードの量が条件変数版より少ないぶんだけ，実行時間も少なくなる．
- システム時間については，条件変数版では mutex と条件変数という 2 つのサービスを利用しカーネル側ロックも 2 つになるのに対し，抽象状態同期では mutex の機能だけで済むのでコストが小さい．

また，反復判定版と抽象状態同期を POSIX Thread API 上で（反復判定を用いて）実装した可搬版はほぼ同じ性能であり，前 2 者と比較して数倍程度遅い．これは，冒頭でも述べたように，反復判定では本来起きる必要のないスレッドがすべて起きて条件検査を行うという無駄があることに起因すると考えられる．これらの傾向については，待ちスレッドの数が多いか少ないかによる違いはあまり見られない．

記述されたコード（付録 1）を見ると，条件変数版はどの場合にどの条件変数に対して signal を実行するかを注意深く見る必要があり，コード的にも複雑である．これに対し抽象状態同期を用いた場合は，排他領域の入口での状態集合と出口での新たな抽象状態を設定するだけであり，素直に記述し，また読むことができる．

さらに，条件変数版はウォーターマークつき有限バッファのように待ち合わせる複数の条件にオーバラップがある場合には適用できないが，抽象状態同期ではそのような制約はない．

6. 議論とまとめ

本論文では，スレッドライブラリに条件同期のための機構として抽象状態同期を組み込むことを提案し，そのようなライブラリを実装した．評価の結果，本方式は今日多く使われる反復判定型の同期よりも優れた性能を持ち，また従来の条件変数で効率良い実装が可能な場合と同程度の性能をより広い場面で得ることが

☆ バグ概要は次のとおり．条件変数内では 2 つのカーネル内ロックを使用し，1 つはユーザ側の mutex データ構造をガードするのに，もう 1 つはカーネル内のキューでの待合せに使用している．オリジナルの実装では，1 つ目のカーネル内ロックを解放してから 2 つ目のロックで待ちに入るため，その「すき間」において競合が発生するのがバグの原因であった．これを修正するため，1 つ目のロックを解放しないままカーネルに入り，カーネル内で 2 つ目の待ちに入る直前に 1 つ目を解放するようにした．

でき，記述性の面からも優っていることが分かった．

筆者らの過去の研究[7),8)]においては，抽象状態同期機構を内蔵した並列オブジェクト指向言語を設計/実装するという立場であったため，言語のランタイムシステムも抽象状態同期を前提に設計することで，OSに手を加えなくても効率の良い同期が実現できた．ただし，一般のプログラマに新しい言語への移行を促すのは簡単ではない．これに対し，本研究のアプローチでは，スレッドライブラリの同期APIの軽微な拡張により抽象状態同期を導入しているため，プログラマの移行性という問題は軽減されている．しかし，ランタイムシステムは既存の環境をそのまま使うため，その性能は，可搬版のようなユーザランドのみの実装では効率面では反復判定と同水準にとどまる．カーネルを手直しして実装を組み込むことができれば，効率良い実現が可能になる．その弱点は当然ながら，軽微であってもカーネルを修正する必要がある，という点である．今回のカーネル修正による実装はごく素直なものであり，変更したコード行数がさほど大きなものではなかったことから，その利点も考慮すると，将来的にはOSの標準のリリースにこの機能が組み込まれるようにすることも考えられる．

抽象状態動機は並行プログラミングにおいて「条件を必要とする同期」を行うほとんどの場面に適用可能である．たとえば，リーダ/ライタロックのような込み入った機能も，抽象状態同期を用いて実現できる[7)]．また別の例として，分配収集型の並列処理をPOSIX Thread APIで行おうとすると，マスタが全スレーブの終了を待ち合わせるのに1個ずつに対し繰り返し`pthread_join()`を呼び出して待つことになるが，本来ならこれは「N個のスレッド全部が終わるまで」待てば1度の同期で済むはずである．このようなものも，抽象状態同期を用いれば「カウントがNになるまで待つ」ためのオブジェクトを作ることで対応できる．すなわち，(この場合でいえばバリアや計数セマフォのような)「手持ちの問題に適した」同期機構がすぐ使える状態にない多くの場面において，抽象状態同期は「自前で効率良い同期用オブジェクトを作る」汎用的な土台として利用できると考える．

今後とも，抽象状態同期のより多様な環境における実装/性能評価を進めてゆくとともに，抽象状態同期の利点を活かすような並行処理のフレームワークについても検討してゆきたい．

謝辞 情報処理学会第57回プログラミング研究会において，有益なコメントをいただいた参加者の皆様に感謝します．また，情報処理学会論文誌：プログラミング編集委員の皆様，および査読者の皆様には論文の改良に関わる助言をいただきました．ここに感謝します．

参考文献

1) Harris, T. and Fraser, K.: Language Support for Lightweight Transactions, *Proc. OOPSLA'03*, pp.388–402 (2003).
2) Herlihy, M.: A Methodology for Implementing Highly Concurrent Data Objects, *TOPLAS*, Vol.15, No.6, pp.745–770 (1993).
3) 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).
4) Hoare, C.A.R.: Monitors: An operating system structuring concept, *CACM*, Vol.17, No.10, pp.549–557 (1974).
5) ISO/IEC 9945-1:1996 Information Technology — Portable Operating System Interface (POSIX) — Part 1, IEEE (1996).
6) Gosling, J., Joy, B., Steele, G. and Steele, G.L.: *The Java Language Specification*, Addison-Wesley (1996).
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., Yonzezawa, A. (Eds.), pp.167–186, Hermes (2000).
9) Lampson, B.W. and Redell, D.D.: Experience with processes and monitors in Mesa, *CACM*, Vol.23, No.2, pp.105–117 (1980).

付 録

A.1 有限バッファのコード（put側）

計測に用いた各ケースの有限バッファについて，put側のコードを示す．通常の有限バッファでは対称性があるのでget側も同様．またウォーターマークつき有限バッファではput側のみ2種類に分かれている．

通常，条件変数版

```
void put(int data) {
  pthread_mutex_lock(&mutex);
  while(count >= BS)
    pthread_cond_wait(&full, &mutex);
  buf[ipt] = data; ipt = (ipt+1)%BS; ++count;
  pthread_cond_signal(&empty);
  pthread_mutex_unlock(&mutex);
}
```

通常，反復判定版

```
void put(int data) {
  pthread_mutex_lock(&mutex);
  while(count >= BS)
    pthread_cond_wait(&delay, &mutex);
  buf[ipt] = data; ipt = (ipt+1)%BS; ++count;
  pthread_cond_broadcast(&delay);
  pthread_mutex_unlock(&mutex);
}
```

通常，抽象状態版

```
void put(int data) {
  ast_mutex_enter(&mutex, S_EMPTY|S_MID);
  buf[ipt] = data; ipt = (ipt+1)%BS; ++count;
  ast_mutex_exit(&mutex, (count>=BS)?S_FULL:S_MID);
}
```

印つき，反復判定版

```
void put(int data) {
  pthread_mutex_lock(&mutex);
  while(count >= BS)
    pthread_cond_wait(&delay, &mutex);
  buf[ipt] = data; ipt = (ipt+1)%BS; ++count;
  pthread_cond_broadcast(&delay);
  pthread_mutex_unlock(&mutex);
}
```

印より上には put しない版は BS のかわりに WM(=BS/2) を条件として判定．

印つき，抽象状態版

```
void put(int data) {
  ast_mutex_enter(&mutex,S_EMPTY|S_MIDLO|S_MIDHI);
  buf[ipt] = data; ipt = (ipt+1)%BS; ++count;
  ast_mutex_exit(&mutex,
    (count>=BS)?S_FULL:(count>=WM)?S_MIDHI:S_MIDLO);
}
void putb(int data) {
  ast_mutex_enter(&mutex, S_EMPTY|S_MIDLO);
  buf[ipt] = data; ipt = (ipt+1)%BS; ++count;
  ast_mutex_exit(&mutex,
    (count>=WM)?S_MIDHI:S_MIDLO);
}
```

A.2 POSIX Threads 版の抽象状態同期ライブラリ

抽象状態同期ライブラリを POSIX Thread ライブラリを用いて実現したものを以下に示す．当然ながら，内部で反復判定を使用しているため，性能面でのメリットはない．

```
#include <pthread.h>

struct ast_mutex {
  pthread_mutex_t ast_pmutex;
  pthread_cond_t ast_pcond;
  int ast_bits;
};

typedef struct ast_mutex *ast_mutex_t;

int ast_mutex_init(ast_mutex_t *mutex, int state) {
  struct ast_mutex *t =
    (ast_mutex_t)malloc(sizeof(struct ast_mutex));
  int v;
  v = pthread_mutex_init(&t->ast_pmutex, NULL);
  v |= pthread_cond_init(&t->ast_pcond, NULL);
  t->ast_bits = state;
  *mutex = t;
  return v;
}

int ast_mutex_destroy(ast_mutex_t *m) {
  int v;
  v = pthread_mutex_destroy(&(*m)->ast_pmutex);
  v |= pthread_cond_destroy(&(*m)->ast_pcond);
  free(*m);
  *m = NULL;
  return v;
}

void ast_mutex_enter(ast_mutex_t *m, int mask) {
  pthread_mutex_lock(&(*m)->ast_pmutex);
  while(((*m)->ast_bits & mask) == 0) {
    pthread_cond_wait(&(*m)->ast_pcond,
                      &(*m)->ast_pmutex);
  }
}

void ast_mutex_exit(ast_mutex_t *m, int state) {
  (*m)->ast_bits = state;
  pthread_cond_broadcast(&(*m)->ast_pcond);
  pthread_mutex_unlock(&(*m)->ast_pmutex);
}
```

(平成 17 年 12 月 21 日受付)
(平成 18 年 3 月 22 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を経て，現在同講師．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．理学博士．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 改訂版』（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．