オペレーティング・システム 48－4
（1990. 9. 7）

TOP-1オペレーティング・システムの構造

河内谷 清久仁　森山 孝男　山崎 秘砂　白鳥 敏幸
日本アイ・ビー・エム株式会社 東京基礎研究所

密結合型マルチプロセッサ・ワークステーションTOP-1上のオペレーティング・システムTOP-1 OSの構造とその開発過程について述べる。TOP-1 OSは、AIX PS/2をベースにマスタ・スレーブ型の並列処理機能を追加したOSである。本稿ではまずTOP-1ハードウェアとTOP-1 OSの基本構造を示し、続いてTOP-1 OS開発の詳細について説明する。TOP-1 OS開発にあたって特に問題となったのは、プロセス管理機構やメモリ管理機構などのMP対応化であった。これらを中心に、MP化にあたっての変更点を示す。さらに、開発及びデバッグに用いた環境やTOP-1 OS上での並列プログラミング法, 残された問題点についてもふれる。

Structure of the TOP-1 Operating System

Kiyokuni Kawachiya　Takao Moriyama　Hisa Yamasaki　Toshiyuki Shiratori

IBM Research, Tokyo Research Laboratory
5-19, Sanbancho, Chiyoda-ku, Tokyo, 102 Japan

TOP-1 (TOkyo research Parallel processor-1) is a tightly-coupled multiprocessor workstation, developed at IBM Tokyo Research Laboratory. This paper describes the structure and development process of TOP-1's MP operating system, which is called "TOP-1 OS." TOP-1 OS is based on AIX PS/2, and supports multiple processors through its master-slave structure. First, the TOP-1 hardware and the basic concepts of TOP-1 OS are described, and then development details such as process and memory managements. The debugging environment of TOP-1 OS is also mentioned, along with other topics related to TOP-1 OS.

1 はじめに

TOP-1 (TOkyo research Parallel processor-1) は日本アイ・ビー・エム (株) 東京基礎研究所で開発された、密結合共有メモリ型ワークステーションである。その主たる開発目的は、マルチプロセッサ・アーキテクチャの設計項目の評価、及び実際の並列ハードウェアを用いての各種並列処理ソフトウェアの研究である。現在TOP-1 上では、AIX PS/2[1]をベースとし、それに並列処理機能を追加したオペレーティング・システムが稼働しており、その上で様々な並列処理の研究が行なわれている。本稿では、このオペレーティング・システムであるTOP-1 OSについて、その開発方法と詳細な構造について説明する。

2 TOP-1 ハードウェアの概要

TOP-1 OS の説明に先だって、TOP-1 のハードウェアについて概要を説明する [1, 2]。図1はTOP-1 の標準システム構成を示したものである。TOP-1 本体は、10 枚のプロセッサ・カード, 2 枚の共有メモリ・カード, 1 枚のハードディスク制御 (HDC) カード, 1 枚のI/O インターフェース・カードから構成される。各プロセッサ・カードはIntel 社の 80386 プロセッサと、浮動小数点プロセッサとして Weitek 社の WTL1167 を装備している。また、128K バイトのスヌープキャッシュを持ち、共有バスを介して共有メモリに結合されている。共有メモリ・カードは 1 枚あたり 16M バイトの容量を持ち、標準構成では32M バイトの共有メモリ空間を提供する。I/O に関しては、300M バイトのSCSI ハードディスクを 4 台装備している。これらは、1 枚のプロセッサ・カードの拡張ローカルバスに接続されたハードディスク制御カードにより制御される。その他のI/O に関しては、I/O インターフェース・カードを介して接続されたIBM PS/55[2]上のものを利用する。

TOP-1 ハードウェアをオペレーティング・システム開発という点からみた場合、以下のような特徴がみられる。

メモリ構成： TOP-1 は基本的には密結合共有メモリ型のマシンであり、各プロセッサから共有メモリへのアクセスの一貫性は、スヌープキャッシュにより保証される。一方、システム・テストや OS サポートのために、各プロセッサ・カードごとに32K バイトのローカルメモリが実装されており、各々のプロセッサからのみアクセスが可能である。

[1] AIX 及び PS/2 は IBM Corp. の商標です。
[2] PS/55 は IBM Corp. の商標です。

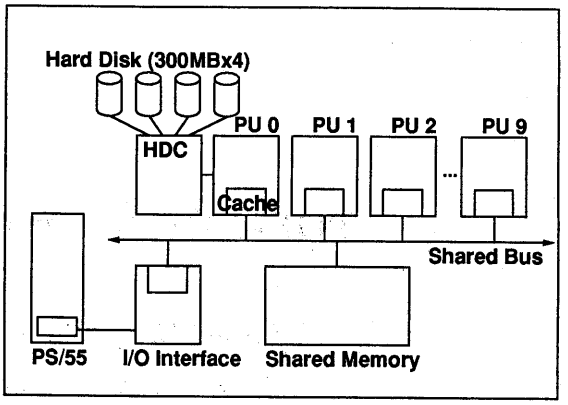

図 1: TOP-1 のシステム構成

同期制御の機構： TOP-1 ハードウェアでは、プロセッサ間の同期制御機構として、バス・ロックによる不可分命令と、プロセッサ間メッセージ機構の二種類が用意されている。バス・ロックは 80386 の LOCK 信号により自動的に行なわれる他、I/O ポートを通じて明示的に行なうこともできる。プロセッサ間メッセージ機構では、任意の複数のプロセッサを指定して 32 ビットのメッセージを送ることができる。

デバイスの接続形態： TOP-1 が直接制御できるデバイスはハードディスクのみであり、その他のデバイス (ディスケット・ドライブ, ストリーマなど) は PS/55 上のものが使用される。このハードディスクは特定のプロセッサからのみ制御可能である。また、主にデバッグ用の目的で各プロセッサ・カードに 2 つずつ非同期通信ポートが用意されている。

PS/55 の接続形態： TOP-1 側から見ると、PS/55 上の 80386 は 11 番目のプロセッサのように見える。若干の制約はあるが、PS/55 から TOP-1 上の共有メモリにアクセスしたり、PS/55 と TOP-1 上のプロセッサ・カードとの間で前述のメッセージをやりとりすることが可能である。これらの制御はI/O インターフェース・カードにより行なわれる。

3 TOP-1 OS の基本構造

一般に、並列計算機用のオペレーティング・システムを作るには大別して二つのアプローチが考えられる。一つはマルチプロセッサ (以下 MP) を考慮した OS を一から作る方法で、CMU で開発された Mach の Core 部分などはこの手法で作られている [3]。もう一つは既存の OS に MP サポートの機構を追加する方法である。前者の方法では、よりきめ細かな MP サポートが可能となる反面、

開発に手間がかかり、既存のOSとの互換性を得ることも大変である。TOP-1 OSの開発では、並列処理研究のテストベッドとして使用するために、早期開発と既存OSとの基本的互換性の二点を重視することにした。このため、後者の方法を用いて開発を行なった。TOP-1 OSのベースとなるOSとしては、IBMのPS/2, PS/55用のUNIX[3]であるAIX PS/2を採用した[4]。

3.1 UNIXカーネルのMP対応化

AIX PS/2は一般のUNIXと同様ユニプロセッサ用のOSであり、当然そのままではマルチプロセッサ上では動かない。最も大きな問題はOSのカーネルがリエントラントになっていないことである。UNIXのカーネルは実行途中でプリエンプトされないことを前提として書かれている。そのため、複数のプロセッサが同時にカーネル・コードを実行すると、カーネル内のデータ構造が破壊されてしまう。この問題を解決するためには、これらのデータ構造に対して排他制御を行なう必要がある。

UNIXカーネルのMP化は、この排他制御のレベルによって、シンメトリ型とマスタ・スレーブ(アシンメトリ)型に大別できる[5]。シンメトリ型では、カーネル内のデータ構造それぞれにロックを設け排他制御を行なう。これにより、使用するデータ構造が競合しない限り、複数のプロセッサ(の上で動く複数のプロセス)が同時にカーネル・コードを実行することが可能となる。これに対し、マスタ・スレーブ型ではカーネル全体に対して排他制御を行なう。複数のプロセスがカーネル・コードを実行しようとした時、そのうちの1つだけがカーネルに入ることを許可され、残りはカーネルの入口で待たされる。

マスタ・スレーブ型は実現が比較的容易である反面、カーネルの実行がシリアライズされるため、それがシステム全体のボトルネックになる危険性がある。シンメトリ型ではその心配はないが、複雑なカーネル・コード全体からロックすべきデータ構造を切り出し、しかもデッドロックなどが生じないように排他制御を行なうには、かなり大規模な変更が必要と考えられる。TOP-1 OS開発にあたっては、短期間で、確実に動作するMPオペレーティング・システムを作成する必要があったため、マスタ・スレーブ型を採用した。さらにTOP-1 OSでは、カーネル・コードを実行するプロセッサを特定のプロセッサに固定している。理由の一つは、TOP-1のハードウェア構成上、ある特定のプロセッサからしかハードディスクにアクセスできないためで、このプロセッサでのみカーネル・コードが実行されるようになっている。

[3]UNIXはAT&T社の開発・許諾製品です。

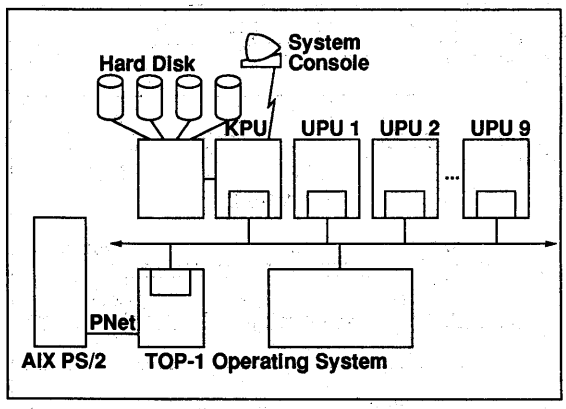

図2: TOP-1 OSの構成

3.2 TOP-1 OSの構成

図2にTOP-1 OSの構成を示す。OSのカーネル・コードは、特定のプロセッサ(KPU: Kernel Processing Unit)上でのみ実行され、これによりカーネル内のデータ構造の一貫性が保証される。残りのプロセッサ(UPUs: User Processing Units)ではプロセスのユーザ・コード部分が並列に実行される。I/Oに関しては、ハードディスクと、KPUの非同期通信ポートに接続されたシステム・コンソールのみがTOP-1 OSから直接制御され、その他のI/O装置はバス結合されたPS/55上のものが使用される。

PS/55上ではAIX PS/2を一部拡張したOSが動作しており、バス上に実現された疑似ネットワーク(PNet: Pseudo Network)を通じてTOP-1 OSとIPレベルで通信することが可能である。さらに、TOP-1 OSとPS/55上のAIX PS/2はIBMの透過ネットワーク環境であるTCF (Transparent Computing Facility)を用いてクラスタ化されており、全体で1つのシステムとして見えるようになっている。

4 TOP-1 OS開発の詳細

TOP-1 OSではマスタ・スレーブ型を採用することで、UNIXカーネル自体をリエントラント化する作業は必要でなくなった。MP対応化にあたって残された問題は以下のようなものであった。

- プロセスをどのようにKPU, UPU上で実行するか
- 各プロセッサの仮想メモリ管理をどのようにするか
- その他にMP対応が必要なものは何か
- 開発及びデバッグの環境をどうするか

本章ではこれらの問題点について、TOP-1 OSでの解決法の詳細を順に説明する。

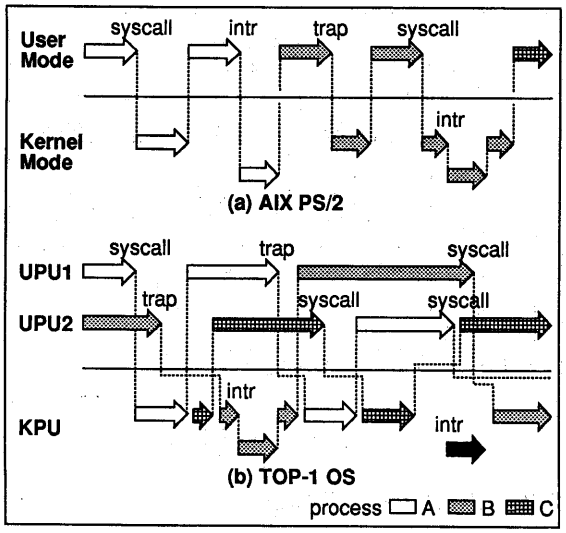

図 3: プロセス及びプロセッサの動作軌跡

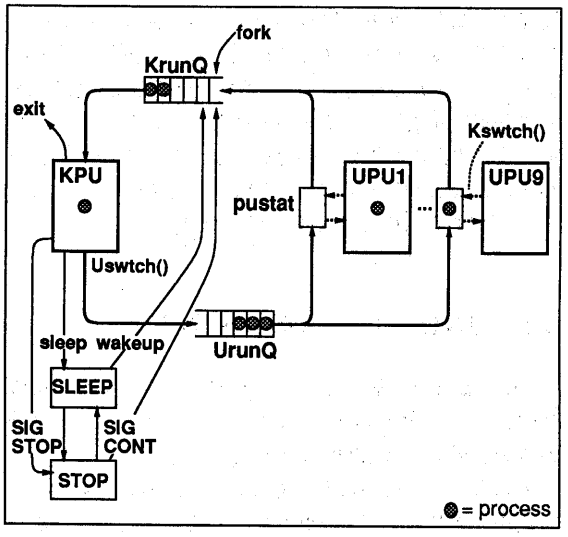

図 4: TOP-1 OS におけるプロセスの状態遷移

4.1 プロセス管理機構の MP 対応化

TOP-1 OS では、複数の UPU に複数のユーザ・モード・プロセスをディスパッチすることで、システムの並列性が引き出される。つまり、TOP-1 OS の MP サポートの中心部はプロセス管理機構にあるといえる。そこでまず最初に、プロセス管理機構の MP サポートについて説明する。

4.1.1 プロセス及びプロセッサの動作軌跡

図 3(a) は、ユニプロセッサ上での UNIX のプロセス及びプロセッサの動作軌跡を示したものである。UNIX プロセスはユーザ・コードを実行している状態 (ユーザ・モード) とカーネル・コードを実行している状態 (カーネル・モード) の 2 状態を行き来しながら動作している。ユーザ・モードからカーネル・モードへの移行はシステムコールやページ・フォールトなどのトラップ, 各種デバイスからの割り込みなどによって引き起こされる。カーネル・モードで必要な処理が終るとプロセスは再びユーザ・モードへ戻り、ユーザ・コードが続行される。この際に、優先度の高い別のプロセスが存在すると、プロセス・スイッチが行なわれることもある。

TOP-1 OS では、プロセスのカーネル・モード部分は特定のカーネル専用プロセッサ (KPU) で実行され、ユーザ・モード部分はそれ以外のプロセッサ (UPU) で実行される。このため、図 3(a) で 2 つのモードが切り替わる部分にプロセッサを切り替えるための機能が追加されている。この機能はそれぞれ Kswtch(), Uswtch() と名付けられた関数によって実現されている。図 3(b) は、TOP-1 OS 上でのプロセス及びプロセッサの動作軌跡を示したものである。プロセスはモードの移行にともない、KPU と UPU の間を行き来しながら実行されていく。前述のように、UNIX においてプロセスがカーネル・モードに移行するのはシステムコール, トラップ, 割り込みの 3 つの場合であり、TOP-1 OS ではこれらの処理ルーチンの入口で Kswtch(), 出口で Uswtch() を実行してプロセッサ切り替えを行なっている。

4.1.2 TOP-1 OS のプロセス管理

通常、システム内には多数のプロセスが実行可能な状態で存在しており、Kswtch(), Uswtch() を行なった時に必ず KPU, UPU が空いているとは限らない。そのため、TOP-1 OS では KrunQ, UrunQ という 2 つのプロセス・キューを用意してプロセスのスケジューリングを行なっている。これは一般の UNIX の runQ (runnable process queue) を拡張したもので、KrunQ には KPU での実行を待っているプロセスが、UrunQ には UPU での実行を待っているプロセスが入れられる。図 4は、これらのキューの扱いも含めた TOP-1 OS におけるプロセスの状態遷移の様子を示したものである。図中の pustat (PU status) は、KPU と UPU の間でプロセスを受け渡しするために UPU ごとに設けられた構造体である。この構造体は KPU, UPU の双方からアクセスされるため、スピン・ロックにより排他制御を行なっている。

これらのプロセス・キューはカーネル・モードで排他的に操作されるべきものであり、TOP-1 OS では KPU によってのみアクセスされる。プロセッサ間でのプロセスの受け渡しには前述の pustat 構造体と、表 1に示したプロセッサ間メッセージのうち STOP, GIVEUP の二種類

KPU の動作

```
1.プロセス管理ループ
  master_sched()
  {loop:
    if (タイム・スライスが来た)
      UPU に STOP メッセージを送る
    KrunQ ← STOP している UPU の pustat
     (プロセス回収)
    UrunQ → IDLE な UPU の pustat
     (プロセス・ディスパッチ)
    KrunQ から最高プライオリティの
      プロセス p を選ぶ
    if (p が存在)
      p を KrunQ から取り出し resume
    else goto loop
  }
2.UPU からの GIVEUP メッセージ処理
  giveup_message()
  {
    KrunQ ← STOP している UPU の pustat
    UrunQ → IDLE な UPU の pustat
  }
```

```
3.スイッチング・コード
  Uswtch()
  {
    if (save(セーブエリア))
      return … UPU で resume された時
    ローカルメモリにスタックを移す
    必要なら UPU に STOP メッセージを送る
    プロセスを UrunQ に入れる
    master_sched() を呼ぶ
  }

  swtch()
  {
    if (save(セーブエリア))
      return
    ローカルメモリにスタックを移す
    master_sched() を呼ぶ
  }
```

UPU の動作

```
1.プロセス待ち合わせループ
  slave_sched()
  {loop:
    自分用の pustat を監視
    if (KPU からプロセス p が与えられた)
      pustat を RUN にして resume
    else goto loop
  }
2.KPU からの STOP メッセージ処理
  stop_message()
  {
    Kswtch()
    (シグナル関係の処理など)
    Uswtch()
  }
3.スイッチング・コード
  Kswtch()
  {
    if (save(セーブエリア))
      return … KPU で resume された時
    ローカルメモリにスタックを移す
    pustat の状態を STOP にする
    KPU に GIVEUP メッセージを送る
    slave_sched() を呼ぶ
  }
```

図 5: プロセス管理コードの詳細

が用いられている。これらのプロセス管理用メッセージの割り込みレベルは、プロセス・キューの一貫性を保証するためクロック割り込みと同じレベルに設定されている。

最後に、プロセス管理に関する各プロセッサの具体的動作を図5に示す。ただし、ここに示したアルゴリズムは各プロセッサの動作を説明するために整理し簡略化したものであり、必ずしも実際のコードと一対一に対応しているわけではない。次節で説明するが、全てのプロセッサは1つのカーネル空間を共有しており、実際のカーネル・コードは、特に KPU 用, UPU 用に分けて書かれているわけではない。またいくつかのショートカット・パスも用意され、実行効率を高める工夫が行なわれている。

4.2 メモリ管理機構の MP 対応化

TOP-1 OS では、プロセスは KPU と UPU の間を行き来しながら実行されていく。この時問題となるのはプロセスの実行環境、特に仮想メモリ空間である。KPU と UPU の間で同じ仮想メモリ空間をマップするための機構が提供されていないと、プロセスをうまく受け渡しすることができない。本節では、まず TOP-1 OS のメモリ管理構造を説明し、つづいてそれを実現するにあたっての問題点と解決法を説明する。

メッセージ	方向	効能
STOP	KPU⇒UPU	UPU 上のプロセスを中断
GIVEUP	KPU⇐UPU	KPU へのスイッチを要求
TLBFLUSH	KPU⇒UPU	TLB フラッシュを要求
PROFILE	KPU⇒UPU	UPU 上の統計情報収集
PNET_TX	PS/55⇔KPU	PNet の送信要求
PNET_RX	PS/55⇔KPU	PNet の受信要求

表 1: TOP-1 OS で使用する主なプロセッサ間メッセージ

4.2.1 TOP-1 OS のメモリ管理構造

TOP-1 OS のベースである AIX PS/2 は、80386 プロセッサのページ変換機構を用いてメモリ管理を行なっている。80386 のページ変換は PD (Page Directory), PT (Page Table) の2段階のテーブル参照によって行なわれ、これにより 80386 では 4G バイトの仮想メモリ空間を利用することができる [6]。図 6(a) は、AIX PS/2 のメモリ管理構造を示したものである。プロセスはそれぞれ自分自身のアドレス空間、すなわち PD を持っている。この PD の物理アドレスを 80386 内の制御レジスタ (CR3: Control Register 3) に登録することで仮想メモリ空間の切り替えが行なわれる。ユーザのコード, データ, スタック用のメモリ空間はそれぞれいくつかの PT を用いて実現されており、プロセスの PD から指されている。カーネル・メモリ空間のマップ用には専用の PT が用意され、全てのプロセスがこの PT を共用している。ここで注意が必要なのが U-area の扱いである。UNIX (AIX PS/2) では、各プロセスごとに U-area と呼ばれる領域が割り当てられており、そこにはそのプロセスに関する様々な情報が納められている。U-area はカーネルによってのみアクセスされる領域であり、カーネル・メモリ空間中に置かれている。さらに、U-area へのアクセスを効率良く行なうために、全てのプロセスの U-area は同じ仮想アドレ

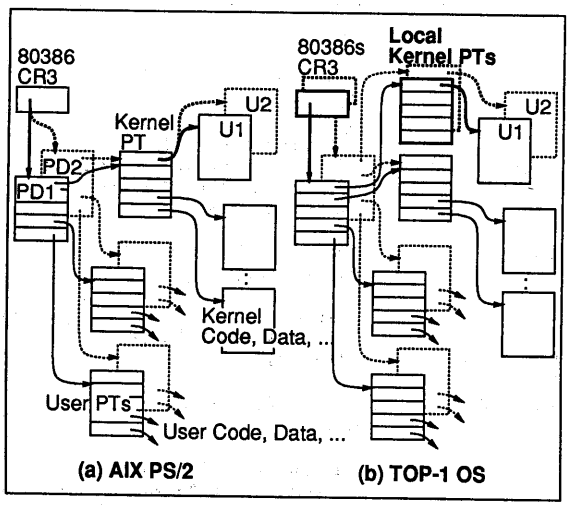

図 6: メモリ管理構造

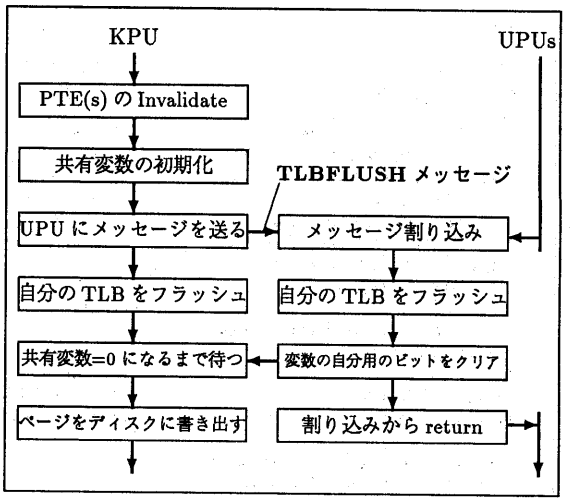

図 7: TLB フラッシュのアルゴリズム

ス上にマップされている。このため、AIX PS/2 で仮想メモリ空間を切り替える際には、前述の CR3 への PD の登録に加えてカーネル用 PT 内の U-area マップ用エントリの書き換えも行なわれる (図 6(a) の太い矢印)。

TOP-1 OS のメモリ管理構造は基本的には AIX PS/2 のものと同様である。図 6(a) の構造のうち大部分は共有メモリ上に置かれ、各プロセッサのプロセス・ディスパッチに応じて使用される。ここで問題となるのが U-area のマップ法である。カーネル用 PT は共有メモリ上にあるため、その中のエントリを書き換えて U-area をマップすることができない。そこで、TOP-1 OS では図 6(b) に示すように、各プロセッサのローカルメモリ上に U-area マップ用の PT を用意している。

4.2.2 メモリ管理コードの MP 対応化

TOP-1 OS では、前項で示したメモリ管理構造にもとづき、デマンド・ページングとスワッピングを用いてメモリ管理を行なっている。ページングでは各ページフレームのイン/アウト、スワッピングでは PD, PT, U-area を含めたプロセス全体のイン/アウトが行なわれる。ただし、カーネル用のコードやデータは物理メモリ上に常駐しており、ページ・アウトやスワップ・アウトされることはない。本項では、このメモリ管理コードの MP 対応化について説明する。

PDE, PTE のロック： まず最初に問題となったのが、PDE (PD Entry), PTE (PT Entry) アクセスの排他制御である。TOP-1 OS ではカーネル・コードを実行するプロセッサを 1 つ (KPU) に固定することで、大部分のカーネル・データに関しては排他制御は不要になっている。しかし、PDE, PTE 内に設けられた Accessed ビットや Dirty ビットは UPU 側の 80386 からも書き換えられるため、これらのエントリに対するアクセスを排他制御してやる必要がある。幸い、80386 のメモリ管理機構自身がこれらのビットの書き換えを行なう場合には自動的にバス・ロック信号が出され、排他的に実行されるようになっているため、実際に問題となるのはカーネル・コード内でこれらのエントリを操作している場合のみである。TOP-1 OS では、これらの操作を全て 80386 の不可分命令で行なうことによって解決している。

PTE の Validate/Invalidate 処理： TOP-1 OS のメモリ管理構造では、カーネル・コード (KPU) が PTE の操作をしている間も、同時に UPU がその PTE を使って走っている可能性がある。そのため、PTE を Validate/Invalidate する際には特別の注意が必要となる。まず、Valid だったページをページ・アウトなどのために Invalidate する場合は、TLB (Translation Lookaside Buffer) 整合性の維持の問題が生じる。TOP-1 OS では表 1 に示した TLBFLUSH というプロセッサ間メッセージを用いて各プロセッサの TLB をフラッシュすることで TLB 整合性を保っている。そのアルゴリズムを図 7 に示す。KPU が TLB フラッシュ完了を知るためには、共有変数を利用している。この方法は、基本的には [7] で示されている方法と同じであるが、TOP-1 OS の場合、ページを Invalidate するプロセッサが KPU に固定されており、また PTE の変更が不可分に行なえるので、UPU 側は全員のフラッシュ完了を待つ必要はない。

次に、Invalid だったページを Validate する場合だが、80386 では Invalid なエントリの情報は TLB にキャッシングされないので TLB フラッシュ処理は必要ではない。

ただし、PTE内のValidビットを立てるタイミングには注意する必要がある。PTE内のデータを正しい状態にしてから、最後にValidビットを立てなければならない。

ページ・フォールト時のタイムラグ： メモリ管理コードの第三の問題点は、ページ・フォールト時の処理である。プロセスがUPU上で実行されている間にページ・フォールトのトラップが起こると、4.1.1で説明したようにそのプロセスはKPUへ移され、KPU上でページ・インの処理等が行なわれる。一般のユニプロセッサ用UNIXの場合この移行は連続的に行なわれるが、TOP-1 OSではその構造上どうしてもタイムラグが生じてしまう。このため、プロセスがページ・フォールトを起こしてからKPU上でページ・イン処理を行なうまでの間に、フォールトを起こしたページの状況が他のプロセスによって変えられてしまう場合がある。そのため、TOP-1 OSのページ・フォールト・ハンドラでは、まず最初にフォールトを起こしたページの状況を調べてから必要な処理を行なっている。

スワップ・アウト候補の決定： 最後の問題点はスワップ・アウト候補の決定法である。ユニプロセッサの場合、スワッパ(プロセス0)が動いている時は他のプロセスは全て停止しており、任意のプロセスを選んでスワップ・アウトすることができる。TOP-1 OSの場合、KPUでスワッパが動いている時もUPU上でユーザ・プロセスが同時実行されているので、これらのプロセスはスワップ・アウト候補からはずすようにしている。4.1.2で説明したように、UPUに対するプロセス・ディスパッチはKPUにより行なわれるので、スワッパがスワップ・アウト候補を選んでいる間にUPUで新たにプロセスが動き出すことはない。

4.3 その他の変更点

TOP-1 OS開発では、プロセス及びメモリ管理機構以外にも変更が必要となる部分がいくつか存在した。本節では、それらについて順に説明する。

4.3.1 クロック処理

TOP-1 OSでは、プロセッサ間メッセージ以外の割り込みは全てKPUに対して行なわれ、KPU上で処理されるようになっている。クロック割り込みも例外ではなく、インターバル・タイマにより20msecごとにKPUにのみ割り込みがかけられる。一般のUNIXでは、クロック割り込みハンドラ内で「割り込み時に動いていたプロセス」に対してリソース使用量のチェック, CPUタイムの更新, プライオリティの再計算などのカーネル・プロファイリング処理が行なわれる。また、profilシステムコールによるユーザ・プロファイリング処理もこの時行なわれる。

ユニプロセッサの場合、1つのプロセスに対してプロファイリング処理を行なうだけであるが、TOP-1 OSではこれを拡張して「割り込み時にKPU/UPUで動いていたプロセス全部」に対して、KPU上のクロック割り込みハンドラがプロファイリング処理を行なっている。この際、UPUで動いているプロセスの状態を知るのにはpustat構造体を用いている。

カーネル・プロファイリング処理の結果の大部分はプロセスのU-area内に書き込まれる。ところが、4.2.1で示したように、プロセスのU-areaは同じ仮想アドレス空間に多重化してマップされており、アドレス空間を切り替えないと直接アクセスすることができない。その手間を避けるため、TOP-1 OSでは各プロセスのU-areaの物理アドレスを調べて、物理メモリがマップされている仮想アドレス空間を通じて間接的にU-areaにアクセスを行なっている。

さらに問題となるのはユーザ・プロファイリングである。ユーザ・プロファイリングの結果は、そのプロセスのユーザ・データ空間に書き込まれる。ところが、KPU上のクロック割り込みハンドラからは、このユーザ空間に直接アクセスすることができない。また、KPUからはUPUのInstruction Pointerの値を見ることができない。これらの問題点のため、現在のTOP-1 OSでは、ユーザ・プロファイリングの機能はサポートしていない。

これらの変更に加えて、TOP-1 OSのクロック割り込みハンドラにはTOP-1 キャッシュ上の統計情報収集ユニットをサポートする機構が追加されている。クロック割り込みごとに、表1に示したPROFILEメッセージが全UPUに送られる。UPU側で動くPROFILEメッセージ・ハンドラ内で、各プロセッサ・カードのキャッシュの統計情報が収集される。

4.3.2 シグナル処理

UNIXでは、プロセスへの非同期的なイベント通知にはシグナルが用いられる。シグナルを受信したプロセスでは、あらかじめ定めておいた方法でシグナルの処理が行なわれる。ここで問題となるのはシグナル受信の方法である。UNIXでは受信側のプロセスがカーネル・モードに入った時に自分自身でシグナルの有無をチェックし、その処理を行なうようになっている。一般のUNIXでは、数十msecごとのクロック割り込みによりプロセスは定期的にカーネル・モードに入り、最悪でもこのタイミングでシグナルのチェックが行なわれる。

TOP-1 OSでは、ユーザ・プロセスはトラップやシステムコールを起こさない限り、いつまでもKPUに移ら

ずに実行され続ける可能性があり、これによりシグナル受信が著しく遅れてしまう場合がある。この問題を解決するために、前項で説明したPROFILEメッセージを利用している。UPUはPROFILEメッセージ・ハンドラの中で、実行中のプロセスに対するシグナル到着の有無を調べ、シグナルが来ているようならKswtch()を実行してシグナル受信処理を行なう。この方法だと、プロセスは最悪の場合20msecの間、シグナルの到着に気付かずに走り続けることになるが、本来UNIXのシグナルは厳密な同期を目的としたものではなく、一般のユニプロセッサ用UNIXでもシグナルの受信に同様の遅れが生じることがあるため、これ以上の対応は行なっていない。

4.3.3 I/O処理

3.2でも述べたように、TOP-1 OSで直接サポートしているI/O装置は、ハードディスクとシステム・コンソールの2つだけである。これらの装置はKPUに直接接続され制御されている。TOP-1 OSはKPU固定のマスタ・スレーブ型の構成をとったため、これらの装置用のデバイス・ドライバに関して、特にMP対応化は必要にならなかった。

TOP-1 OS特有のI/O機能としては、疑似ネットワーク(PNet: Pseudo Network)があげられる。これはTOP-1とそれに接続されたPS/55の間に実現された疑似的なネットワークで、両マシンの間にIPレベルのPoint to Pointの通信路を提供している。この疑似ネットの実現のために、表1に示したPNET_TX, PNET_RXの2種類のプロセッサ間メッセージが用いられている。

4.4 開発・デバッグ環境

TOP-1 OSの開発は大きく二つの段階に分けて行なわれた。まず、第一段階ではTOP-1上に、KPUのみで動くユニプロセッサ版のOSを作成した。この段階では、ブート・コードやデバイス・ドライバなどを作成した。次の第二段階では、第一段階で作成したOSをマスタ・スレーブ型にする作業を行なった。開発・デバッグのための環境としては、PS/55上のAIX PS/2を主に使用した。

4.4.1 開発環境

図8に開発環境の概要を示す。全体がTCFによりクラスタ化されており、どのPS/55からも同じ環境が見えるようになっている。

OS開発の初期段階における問題の一つは、どのようにしてターゲットマシン上にOSの動作環境を作るかという点である。特に、どうやって初期ファイルシステムを作るかが問題となる。TOP-1 OS開発ではSCSIのマ

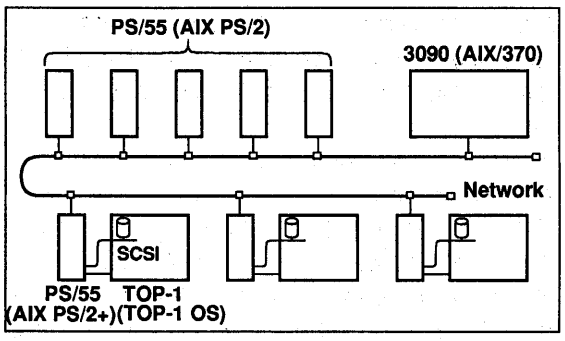

図8: TOP-1 OSの開発環境

ルチ・イニシエータ機能を利用してこの問題を解決した。図8に示した通り、TOP-1の1台目のハードディスクのSCSIバスを外部に引き出し、PS/55からもアクセスできるようにした。これにより、PS/55上のAIX PS/2から直接TOP-1のハードディスクにファイルシステムを作成することが可能となった。

TOP-1に接続されたPS/55上のAIX PS/2には、開発及びデバッグ支援用にいくつかの機能が追加されている。まずデバイスとして、前述のSCSIハードディスク/dev/schd*の他、80386のI/O空間にアクセスするための/dev/io, TOP-1上の共有メモリにアクセスするための/dev/top1shmemなどが追加されている。さらに、TOP-1上の共有メモリ全体をあらかじめマップした共用メモリ・セグメントも用意されており、System V系UNIXのshmget/shmatシステムコールでPS/55上のプロセスのアドレス空間にアタッチして、直接アクセスすることもできるようになっている。

4.4.2 デバッグ環境

次にTOP-1 OSのデバッグに用いた環境について説明する。AIX PS/2のカーネルにはデバッガが組み込まれており、カーネル・コードの停止/再開, レジスタやメモリのダンプ, カーネル内のデータ構造やスタック・フレームの表示などを行なうことができる。TOP-1 OS開発においても、このカーネル・デバッガをかなり利用することができた。

ただし、このカーネル・デバッガでデバッグできるのはKPUのみで、UPU上で動くOSコードのデバッグや、システム全体の状況をトレースする目的には利用できない。このために、共有メモリ上にプロセッサごとに独立した仮想端末エリアを用意した。トレースのためにOSコード内に埋め込まれたprint文の出力は、それを実行したプロセッサに対応した仮想端末エリアに行なわれるようにした。この仮想端末エリアの内容はPS/55上の専用ブラウザによりリアルタイムで見ることができる。こ

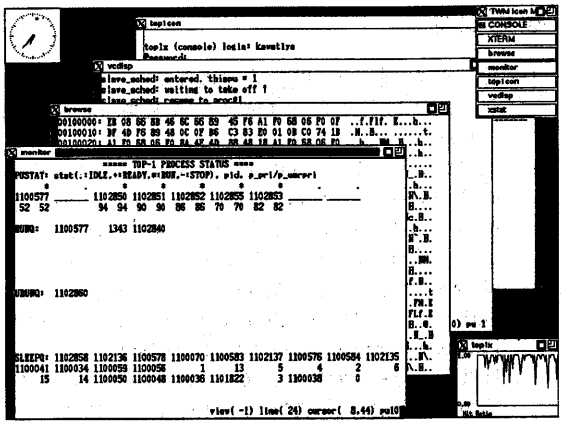

図 9: TOP-1 OS のデバッグ環境

の他 PS/55 上には、共有メモリの内容を見るためのメモリ・ブラウザも用意した。図9はそれらの表示例である。

カーネルのデバッグがある程度まで進むと、システム全体の挙動を把握する必要が生じてくる。このため TOP-1 OS では、必要に応じて1台のプロセッサをモニタリング専用に割り当て、仮想端末にプロセスの実行状況を表示させるようにしている。図9の一番手前のウィンドウがその表示例である。各プロセッサで実行中のプロセスの ID とプライオリティや、KrunQ, UrunQ 及び Sleep queue に入っているプロセスの ID などがモニタされている。

4.5 TOP-1 OS 開発のまとめ

本章では TOP-1 OS の開発について、特に MP 対応化の方法を中心に説明した。AIX PS/2 をもとに TOP-1 OS を開発するにあたって変更・追加したカーネル・コードの量は、行数で数えて全体の 5%弱であった。ただし、この中にはデバイス・ドライバなど MP 関係以外の変更も含まれている。また、ブート・コードや開発・デバッグ環境などのカーネル以外の部分は含まれていない。開発期間は、第一段階のユニプロセッサ版の開発に 2 カ月, 第二段階の MP 対応化にさらに 2 カ月程度を要した。これらの数字からもわかる通り、MP サポートの OS の早期開発という目標は達成されたといってよいだろう。

5 TOP-1 OS 上での並列プログラミング

TOP-1 OS における最もプリミティブな並列実行単位はプロセスである (Mach に見られるような、カーネル内で実現されたスレッドも一時試みたが [4] 、TOP-1 OS の現在の版には入っていない) 。一つのアプリケーションが多数のプロセスを fork し、それらが UPU 上で同時実行されることで並列性が生まれる。本章では、このプロセスを用いた並列プログラミングを行なう際に利用できる機能について簡単に説明する。

まず、プロセス間でデータを共有する方法であるが、TOP-1 OS では二種類の方法が提供されている。第一の方法は System V 系 UNIX で用意されている共用メモリ・セグメントの機能を使う方法である。共用メモリ・セグメントをアタッチした後 fork を行なうと、そのエリアをプロセス間で共有することができる。ただしこのやり方では、C などの言語から共有エリア内に置いた変数に名前を付けて直接アクセスすることができない。このため、TOP-1 OS では第二の方法として sharedfork という新しいシステムコールを用意している。sharedfork では、プロセスのデータ空間を共有したまま fork が行なわれる。スタック空間は共有されず、通常の fork と同様プロセスごとに別のものが用いられる。この方法では共有エリアに直接アクセスすることができるが、データ空間全体が共有されてしまうので、プログラミング時には特に注意が必要となる。

次にプロセス間の同期の方法であるが、これにも二種類の方法が利用できる。第一の方法は System V 系 UNIX のセマフォ機能を使う方法である。ただし、このセマフォ操作はシステムコールとして提供されているため、特に TOP-1 OS では KPU へのスイッチが起こってしまい効率が良くない場合がある。もっと高速に同期を行ないたい場合には第二の方法として、プロセスの共有メモリ上に同期用の変数を用意し、80386 の不可分命令を用いて Busy Wait 方式の同期をとる方法を使う必要がある。

ここで説明した機能は非常にプリミティブな機能であり、アプリケーションを書く際に使いやすいとはいえない。そのため我々は、各種言語処理系やユーザレベル・スレッド機能などを用いた、より使いやすい並列プログラミング環境を整備中である [8, 16] 。

6 TOP-1 OS の問題点

最後に、現在の TOP-1 OS の問題点について述べる。まず明らかな問題点は、マスタ・スレーブ型の採用によるカーネル・ボトルネックである。TOP-1 OS ではユーザ・コードは UPU で並列実行されるが、カーネル・コードは KPU 上でシリアライズされてしまう。そのため、アプリケーションの種類によってはカーネルの実行がボトルネックとなり十分な並列性が出ない場合が予想される。また、システムコールなどのカーネル処理は全て KPU 上にスイッチして行なわれるため、一般のユニプロセッサ用 UNIX より時間がかかってしまうという問題点もある。

次に問題となるのが、プロセスがプロセッサ間を渡り歩

いてしまう点である。プロセッサ・カード上のキャッシュのことを考慮すると、一つのプロセスはなるべく同一のプロセッサで実行される方が望ましい[9]。TOP-1 OSのプロセス・スケジューラでもある程度の工夫は行なっているが、システム内のプロセス数が多くなると、必ずしもプロセスを同一プロセッサに割り当てることができていない。

第三の問題点は並列プログラミングに関するものである。TOP-1 OSは協調動作する複数のプロセスを認識する機能を持っておらず、これらのプロセスが同時にディスパッチされるとは限らない。プロセス間の同期をとる場合にはこの点に注意する必要がある。さらに、ライブラリがリエントラントでないという問題がある。ライブラリ関数の中にはデータ・セグメント内の変数を利用しているものがあり、この変数のアクセスには特に排他制御を行なっていない。これはsharedforkを用いて並列プログラムを作成する時に問題となる。

これらの問題点のうちいくつかは、OSに簡単な変更を加えることである程度解決可能である。しかし、完全に解決するためにはOSコードやライブラリのリエントラント化, 並列プログラミングのための新しいモデルとOS機能の提供といった根本的な対策が必要となると思われる。

7 おわりに

本稿では、TOP-1 OSの構造とその開発方法について説明した。TOP-1 OSはマスタ・スレーブ型の構造を用いたMPサポートOSであり、OSそのものの性能という点では検討すべき点がいくつか残されている。反面、マスタ・スレーブ型の採用により、短期間で開発することができ、早期に並列処理研究のためのテストベッドを実現できたと考えている。

現在TOP-1 OS上では、MP拡張したCommon Lisp[10, 11]やFortran処理系[12, 13]などが稼働しており、その上で動くアプリケーションも含めて研究が続けられている。OSそのものに関しても、OSの挙動の解析と評価[14]や各種スレッド機構の研究[15, 16], 新しいOS機構の研究[17]などを積極的に行なっている。また、試作したTOP-1をいくつかの大学での並列処理に関する研究に利用していただいており、今後の成果が期待される。

謝辞

本研究の機会を与えて下さった日本アイ・ビー・エム(株)東京基礎研究所の鈴木則久所長と山内長承氏, 穂積元一氏(現大和研究所)に感謝いたします。

参考文献

[1] 鈴木則久 他: "高性能並列処理ワークステーション(TOP-1)," 第37回情処全大論文集, pp.171–183 (1988)

[2] N. Oba, et al.: "TOP-1: A Snoop-Cache-Based Multiprocessor," Proc. of the 9th. Annual IEEE International Phoenix Conf. on Computers and Communications, pp.101–108 (1990)

[3] M. Accetta, et al.: "Mach: A New Kernel Foundation for UNIX Development," Proc. of USENIX 1986 Summer Conf., pp.93–112 (1986)

[4] 穂積元一 他: "TOP-1 オペレーティング・システム," 第39回情処全大論文集, pp.1199–1210 (1989)

[5] U. Sinkewicz: "A Strategy for SMP ULTRIX," Proc. of USENIX 1988 Summer Conf., pp.203–212 (1988)

[6] "80386 Programmer's Reference Manual," Intel Corporation (1986)

[7] D. L. Black, et al.: "Translation Lookaside Buffer Consistency: A Software Approach," Proc. of the 3rd. ACM International Conf. on Architectural Support for Programming Languages and Operating Systems (1989)

[8] 大庭信之 他: "並列処理ワークステーションTOP-1の性能評価環境," 情処研究報告90-ARC-83, pp.139–144 (1990)

[9] 上脇 正 他: "並列処理用OS SKY-1のスケジューリング方式," 第39回情処全大論文集, pp.1197–1198 (1989)

[10] T. Tanaka and S. Uzuhara: "Multiprocessor Common Lisp on TOP-1," To be presented at the 2nd. IEEE Symposium on Parallel and Distributed Processing (1990)

[11] 渦原 茂: "共有メモリ型マルチプロセッサにおける並列ガーベージコレクション," 情処研究報告90-ARC-83, pp.205–210 (1990)

[12] 大澤 暁: "TOP-1における流体問題解析," 情処研究報告90-ARC-83, pp.31–36 (1990)

[13] 大澤 暁: "TOP-1における流体問題解析," 第41回情処全大論文集, pp.6-149–6-150 (1990)

[14] N. Yamanouchi: "Performance Effects of Program Structures on a Snoop-cached Multiprocessor System," To be presented at the IPSJ InfoJapan'90 (1990)

[15] 渦原 茂, 森山孝男: "マルチプロセッサシステムにおけるライトウエイトプロセス機構," 日本ソフトウェア科学会第6回大会論文集, pp.353–356 (1989)

[16] 根岸 康: "密結合マルチプロセッサのためのスレッド機構," 第41回情処全大論文集, pp.4-125–4-126 (1990)

[17] 森山孝男 他: "粒度を考慮したマルチプロセッサの資源管理," 情処研究報告 90-ARC-83, pp.103–108 (1990)