2008－SE－162 (7)
2008／11／19

Universal Shell Programming(USP)向けシェル機能と実装

片山善夫[†] 當仲寛哲[††]
[†]タオベアーズ合同会社
[††]有限会社ユニバーサル・シェル・プログラミング研究所

近年の計算機の劇的な高性能化により、業務アプリケーションの開発をシェルのみで行う USP 手法が成功を収めている。USP 手法で開発されたシステムは、従来のシステムと比べ、遥かに低いシステム価格で遥かに高い性能を持っている。しかし、一方で開発言語としてのシェルの問題点も明らかになってきている。また、マルチコア CPU の普及によって、更なる高性能化の可能性も出てきた。これに対応するため、Unix で提供されているシェルに USP 手法向けの機能を実装した。実装した機能は、エラー処理機能と並列処理機能である。

The Functions of the Shell for Universal Shell Programming(USP) and it's Implementation

Yoshio Katayama[†] Nobuaki Tounaka[††]
[†]TAO Bears LLC.
[††]Universal Shell Programming Lab.

With the dramatic improvement of the computer of recent years, USP method with which business application is developped using only the shell has achieved successful result. The system which was developed with USP method has much high efficiency at much low system cost in comparison with the former system. But, on the one hand also the problematical point of the shell as a programming language for software development has become clear. In addition, also possibility of further improvement came out due to the spread of multiple core CPU. To cope with this situation, the function for USP method was implemented to the shell which is presented with UNIX. The implemented functions are error handling functions and parallel processing function.

1. はじめに

在庫管理、商品管理等を行う業務アプリケーションは、一般に COBOL 等のコンパイル言語と RDB(Relational Data Base)を用いて開発される。大規模スーパーマーケットのように多数の店舗を持つ大規模小売業では売上情報や発注情報が大量に発生するため、RDB は大規模となり、高速な処理が必要になる。また、業務アプリケーションがダウンすると、発注や配送などが滞り業務自体に悪影響が発生するので、業務アプリケーションが稼働するシステムは二重化等の対策が施される。その結果、システムは一層大規模化し、ハードウェアやソフトウェアの価格及び開発費を含めたシステム価格が数億～数百億円になる。更に、メンテナンス費用もシステム価格に応じて高額となる。

しかし、これと全く異なる業務アプリケーションの開発手法が USP(Universal Shell Programming)手法である。USP 手法では、名前のとおり Unix のシェルスクリプトでプログラミングを行う。USP 手法の基本となるアイデアは以下のとおりである。[1)]

(1)発生データは棄てない。すべて貯めておく
(2)効率よりも移植性。
(3)さまざまな技術を混在させない。
(4)柔軟に変化できることが大事。

発生データをすべて貯めておき、削除・変更を行わないので、データの変更履歴が不要になり、RDBにおけるロールバックも不要になる。発生データは1件1ファイルとし、ファイル命名方法を工夫することにより、ファイルシステム自体をデータベースとして扱えるようにすることで、RDB無しで業務アプリを構築している。

初心者でも開発に参加できるようするため、できるだけ少ない技術でシステムを構築する。そのためにシェルスクリプトを開発言語として採用している。コマンドラインインタプリタとしてシェルを修得すれば、開発言語の習得も同時に行えるのである。

システムの柔軟性を確保するため、USP手法ではプログラムの再利用をスクリプトのコピーで行っている。これはオブジェクト指向におけるクラスライブラリの利用と真っ向から対立している。ライブラリ化していると、ライブラリ化された部分の変更が必要になった時に、影響範囲の同定が非常に困難になる。その結果、変更を容易に行うことができなくなり、システムの柔軟性が失われる。

USP手法では、以下のUNIXの考え方[2]を踏襲している。

(1)スモール・イズ・ビューティフル
(2)一つのプログラムにはひとつのことだけをやらせる
(3)できるだけ早く試作する
(4)効率より移植性を優先する
(5)データはテキスト形式で保存する
(6)コマンドを梃子として使う
(7)シェルスクリプトでアプリケーションを記述する
(8)すべてのプログラムをフィルタとして設計する

プログラムを単機能化することは、プログラムを小さくすることにもなる。その結果、プログラムは使い捨てにすることができ、プログラムの保守は変更ではなく作り直しで行うことができる。変更では既存部分への予想外の副作用が出ることがあるが、作り直しならその心配は不要である。

データをテキスト形式で保存することで、プログラムの動作確認やデバッグが容易になる。更にシェルスクリプトではデータを変数で持つことが殆どないため、プログラムの内部状態が少ないことも、デバッグを容易にしている。

すべてのプログラムをフィルタとして設計することによって、パイプラインの利用が可能となり、マルチCPU/コアシステムにおける並列処理が可能になる。USP手法では効率や性能を最優先としているわけではないが、やはり重要なファクターである。並列処理が容易に行えることは重要である。

USP手法による開発例として(株)良品計画の場合を図1-1に示す。

店舗規模：直営店181店、商品供給店147店，商品点数 23万点,売上個数 57万点/日
開発規模：260モジュール, 6人×6か月
ハードウェア構成：PC 80台
コスト*)：USP導入前8億円(ハード+ソフト)
→USP 導入後 3000 万円（ハード）
+4000万円（ソフト）
*) ハードウェア価格+ソフトウェア価格+ソフトウェア開発費

図1-1 (株)良品計画でのUSP適用例

2. シェルプログラミングの問題点

業務アプリケーションでは、何らかのエラーを検出した際は直ちにジョブを停止すること

が必要である。従来のDBを用いた業務アプリケーションでは、DBのロールバック等の後処理を行った後にジョブを停止しなければならないが、USPでは基本データは追記していくのみなので、単純にジョブを停止すればよい場合が殆どである。このため、エラー処理はエラー検出が中心となるのであるが、シェルの機能不足からエラー処理が煩雑になりやすく、プログラムの見通しが悪くなり、プログラムミスを誘発しやすくなる。したがって、エラー処理は可能な限り簡素に行える必要がある。この観点からUSPで用いられているシェルであるbashの問題点を考察する。また、USPおける性能上の問題点についても考察する。

2.1 エラー検出

シェルからコマンドが起動された正常終了したことの判定は、終了ステータスが0か否かで行われる。終了ステータスが0の場合が正常終了である。終了ステータスは特殊シェル変数$?に代入されるので、通常はこの変数を用いて判断される(図2-1)。

```
cmd1 args…
if [ $? -ne 0 ]; then
   エラー処理
fi
```

図2-1 コマンドのエラー判定

しかし、終了ステータスは必ずしも$?に代入されるわけではなく、以下のコマンドでは$?に代入されない。

(1)if文等の条件判定部のコマンド

(2)パイプラインの最後以外のコマンド

(3)バックグランドジョブのコマンド

条件判定部では実行したコマンドの終了ステータスで条件判断を行うので、終了ステータスが$?に代入されなくても不都合はない。しかし、他の場合は、コマンドの正常終了の判定を行うには終了ステータスを取得する必要がある。その為には他のコマンドを組み合わせる必要がある。

パイプラインではパイプラインの最後のコマンド終了ステータスが$?に代入され、前段のコマンドの終了ステータスは反映されない。しかし、bashではシェル配列変数PIPESTATUSにパイプラインの各段の終了ステータスが代入される。USP手法では、この変数と引数の総和を求めるツール(plus)を用いて、パイプラインのエラー検出を行っている(図2-2)。

```
cmd1 args… | cmd2 args… | …
STAT=`plus ${PIPESTATUS[@]}`
if [ $STAT -ne 0 ]; then
   エラー処理
fi
```

図2-2 パイプラインのエラー検出

バックグラウンドジョブの終了ステータスはwait組込コマンドの終了ステータスとして取得することができる。しかし、wait組込コマンドがシグナルで中断された場合は正しい終了コードを取得できない。このため、終了ステータスを取得するには、終了ステータス受渡し用のファイルを作り、そのファイルを経由して渡す必要がある(図2-3)。

```
(cmd args…; echo $? >/tmp/$$.1) &
wait
STAT=`cat /tmp/$$.1`
if [ $STAT -ne 0 ]; then
   エラー処理
fi
```

図2-3 バックグランドジョブのエラー検出

シェルを-eフラグ付きで起動すると、シェルから起動されたコマンドが正常終了しなかった場合(0以外の終了ステータスで終了した場合)は、直ちにシェルが終了する。しかし、

すべての場合でシェルが終了するわけではない。bash では以下のコマンドが正常終了しなかった場合でもシェルが終了しない。条件判定部以外でも終了しない場合があるので、-e フラグによるエラー自動検出に頼ることができない(図 2-4)。

(1)if 文等の条件判定部のコマンド
(2)パイプラインの最後以外のコマンド
(3)バックグランドジョブのコマンド
(4)サブシェル配下で実行されたコマンド

サブシェルとは、シェルが自身の複製を作り、その複製によって実行することである。サブシェルは( )による明示的に指定される場合だけでなく、暗黙的にサブシェルになる場合があり、サブシェルでの実行であることが分かり難く、プログラムミスを誘いやすい。

```
#!/bin/bash -e
if cmd args…; then  条件判定部
  …
fi
cmd args… | …  パイプライン前段
cmd args…&   バックグランドジョブ
(cmd args…)     サブシェル
```

図 2-4 -e フラグで終了しない場合

2.2 エラー処理

業務アプリケーションではエラー判定は非常に多くの個所で行わる。エラーと判定されたときに行われるエラー処理は、作業ファイルの消去等の後処理を行った後にシェルスクリプトの実行を終了するという処理になることが多い。このため、エラー処理は本来の制御フローとは異なる個所にまとめて記述すると可読性が向上する場合が多い。このような場合に goto 文は必ずしも必須ではないが、goto 文を用いた方が自然で分かりやすい記述となることが多い。goto 文の乱用はプログラムの可読性を著しく低下させる。このため、bash を含む Bourne shell 互換のシェルでは敢えて goto 文を持たない文法としている[3]。しかし、goto 文が有益な場合は少ないながらもあり、エラー処理はその一つである。

2.3 性能

USP 手法では RDB を用いないので、従来手法で RDB がボトルネックとなるクエリ処理(例えば在庫状況の問合せ等)では大幅な性能向上がある。しかし、ソートなどの CPU 負荷が高い処理では、スクリプト言語であるために性能的には不利である。それを補うには並列実行が有効である。マルチコア CPU の普及によって、低価格マシンでも並列実行が可能になってきた。より性能向上を図るには、CPU コアを有効に使い、OS のオーバーヘッドを減らすことが必要である。

パイプラインを用いることにより、コマンドの並列実行を行わせて性能向上を図ることができる。しかし、マルチコア化が進んだことよって 16 コアマシンも珍しくなくなってきている。このため、パイプライン長より CPU コア数のほうが上回り、CPU コアを活用しきれない場合が出てきている。このため、1 つのコマンドを複数の CPU コアで実行する並列化を考える必要がある。

Linux では CPU 負荷に応じてコマンドを実行する CPU を変更させることがある。これはマイグレーションと呼ばれている。通常はマイグレーションに伴うオーバーヘッドより負荷分散の効果が大きいので、全体の効率は向上している。しかし、特定のコマンドの性能を向上させたい場合では、そのコマンドのマイグレーションを抑止することによって性能向上を図ることができる。

Linux にはリアルタイム処理のためにプロセスの優先実行の機能がある。この機能を利用

することによってプロセススイッチのオーバーヘッドを抑えられるので、特定のコマンドの性能を向上させることができる。

3. USP 手法向けシェル機能と実装

USP 手法により大幅なコストダウンが達成されてきたが、前章で示したようにシェルには問題点がまだ残されている。本章ではこれらの問題点に対する解決方法を提示するとともに、bash における実装方法も提示する。

3.1 エラー検出

3.1.1 パイプラインの終了ステータス

パイプラインの終了ステータスとして $? に代入される値を全コマンドの終了ステータスの演算値に変更した。これにより、パイプラインの前段でエラーが発生する場合を含めて、パイプラインのエラー判定を $? で行うことが可能になる(図 3-1)。終了ステータスの演算は、総和（算術和）、論理和、最大値のいずれかを選択できる。指定はシェル変数 PIPESTATUS_OP で行う。この変数を未定義又は空文字列に定義した場合は、パイプラインの最後のコマンドの終了ステータスが $? にされるので、従来のシェルとの互換性を保っている。

```
PIPESTATUS_OP=OR
cmd1 args… | cmd2 args… | …
 (cmd1,md2,…の終了ステータスの論理
 和が $? に代入される)
if [ $? -ne 0 ]; then
   エラー処理
fi
```

図 3-1 パイプラインの終了ステータス

パイプラインの前段のコマンドより後段のコマンドが先に終了した場合は、前段のコマンドがシグナル SIGPIPE で終了することがある。アプリケーションからみた場合、SIGPIPE による終了は正常であることが多い。このため、終了ステータスが SIGPIPE による終了であることを示している場合は、終了ステータスを 0 として扱えると便利である。この指定はシェル変数 IGNORE_SIGPIPE_EXIT で行うことができる(図 3-2)。

```
PIPESTATUS_OP=OR
IGNORE_SIGPIPE_EXIT=y
cmd1 args… | cmd2 args… | …
```

SIGPIPE で終了しても
正常終了と扱われる

図 3-2 SIGPIPE の扱い

3.1.2 バックグラウンドジョブの終了ステータス

bash ではフォアグランドジョブの終了ステータスは $? に格納される。このシェル変数を配列に拡張し、添え字 0 の要素にフォアグランドジョブの終了ステータスを、添え字 1 以上の要素にバックグラウンドジョブの終了ステータスを格納するように変更した(図 3-3)。添え字はバックグランドジョブのジョブ番号が用いられる。bash では、配列変数を添え字なしで参照すると、添え字 0 の要素が参照されるので、この方法では変更前と互換性が保たれる。

```
cmd args… &
wait
if (( ${?[1]} != 0 )); then
   エラー処理
fi
```

図 3-3 配列に拡張された $?

終了ステータスとして記録される値は、シェル変数 PIPESTATUS_OP で終了ステータスの演算方法が指定されてない場合はパイプラインの最後のコマンドの終了ステータス、演算方法が指定されている場合はその演算方法で

求められた値になる。

シェル変数 PIPESTATUS も配列に拡張し、バックグランドジョブのパイプラインの各コマンドの終了ステータスが記録するようにした。

バックグランドジョブの終了ステータスを確実に取得できるためには、バックグランドジョブに割り当てられるジョブ番号が重複しないことが必要である。バックグランドジョブに割り当てられるジョブ番号は、実行中ジョブまたは実行完了が報告されていないジョブのうち最大のジョブ番号に 1 を加えた値である。シェルスクリプトを実行している場合は、jobs 組込コマンドを実行しない限りジョブの実行完了が報告されないので、ジョブ番号が重複して割り当てられることはない。

3.1.3 -e フラグによる自動終了

前述した -e フラグを指定してもエラー発生時に自動終了しない場合のうち、パイプラインとサブシェルの場合は、コマンドが正常終了しなかった場合にシェルが直ちに終了するように変更した。バックグランドジョブのコマンドが正常終了しなかった場合は、wait 組込コマンドを実行した際にシェルが終了するように変更した。これにより、-e フラグによる自動終了をエラー検出として利用できるようになった。

3.2 エラー処理(goto 文)

エラー処理への分岐の為に goto 文を実装した。エラー処理への分岐が目的であるので、goto 文及び分岐先であるラベルには以下の制限を課している。

(1) goto 文及びラベルに対して変数展開、コマンド展開等は行わない
(2) eval、コマンド展開(``)内の goto 文及びラベル
(3) ループ文外からループ文中への分岐
(4) シェル関数外からシェル関数内への分岐及びその逆
(5) サブシェル外からサブシェル内への分岐及びその逆

bash では、シェルスクリプトを構文解析した結果を構文木で保持している。シェルスクリプトの読み込み、構文解析は文単位に行われる。読み込まれた文が関数定義以外の場合は、直ちに実行され、構文木は破棄される。このため、goto 文を実装するために、シェルスクリプトのトップレベル(シェル関数定義の外側)の実行部分に以下の処理を追加した。

(1) ラベルが現れたら、以降の構文木を保持する
(2) 未定義ラベルを参照する goto 文が現れたら、そのラベルが現れるまでスクリプトを読み込む

(1)は後方参照の為で、(2)は前方参照の為である。上記の処理はシェル関数内では不要である。シェル関数では、関数全体の構文木が保持されているので、新たな構文木の保持やスクリプトの追加読み込みの必要がないからである。

3.3 性能

3.3.1 並列実行

パイプラインで使われるコマンドには、行単位で処理を行い、前後の行の影響を受けないものが少なくない。例えば、grep や tr などは、その代表的な例である。このようなコマンドは、複数個同時に起動して、入力を分割して与えることにより、並列化を行うことができる。この並列化を para 文として実装した。para 文は、指定された並列度の個数のコマンドを起動し、前処理(入力分割)及び後処理(出力収集)とパイプで繋ぐ。前処理は入力を一定行数ごとに分割

し各コマンドへ分配する。後処理は各コマンドの出力を集めて1つの出力にまとめる(図3-4)。

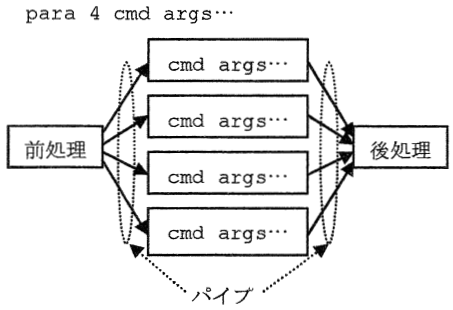


図 3-4 para 文による並列実行

並列実行できるコマンドは、(1)入力行と出力行が一対一対応するコマンド、(2)NUL 文字のみの行が保存されるコマンドのいずれかに制限される。(2)は、grep のように入力行と出力行が一対一対応しない場合に、入力行を分割する際に NUL 文字のみの行を分割のマーカーとして追加しておき、後処理ではマーカーを目印にして出力を組み立てる(図 3-5)。

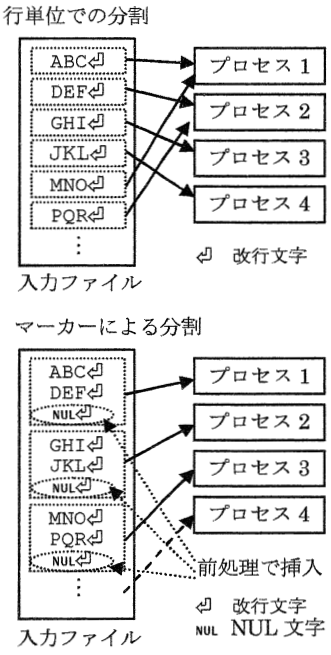


図 3-5 入力ファイルの分割方法

3.3.2 パイプラインの実行効率向上

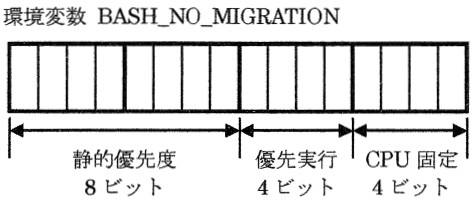
USP では実行時間の大半がパイプラインの実行に費やされている。したがって、パイプラインを実行する際に、各コマンドの実行効率を高めると性能が向上する。実行効率を落とす原因としてマイグレーションとプロセススイッチがある。パイプライン長より CPU コア数が多い場合は、パイプラインの各コマンドに CPU を固定的に割り当て、更に優先実行機能を用いてプロセススイッチが起きないようにすればよい。実行する CPU を固定するのはシステムコール **sched_setaffinity** で、優先実行はシステムコール **sched_setscheduler** で、それぞれ行うことができる。

sched_setaffinity は実行を許可する CPU をビットベクターで指定する。1 ビットのみ 1 で他のビットをすべて 0 のビットベクターを指定すると、1 のビットに対応した CPU のみで実行され、マイグレーションが起きなくなる。パイプラインを実行する場合、パイプラインの各コマンドに異なる CPU を順に割り当て、実行する CPU を **sched_setaffinity** でその CPU に固定する。複数のジョブが同時に起動された場合に特定の CPU が集中的に割り当てられることがないように、最初に割り当てる CPU は乱数で決めている。

sched_setscheduler はプロセススケジューリングポリシーを変更して優先実行を可能にする。Linux でサポートされている優先実行のスケジューリングポリシーは SCHED_FIFO と SCHED_RR の 2 種類がある。SCHED_FIFO は、I/O 要求による停止等の事象が発生するまでプロセスが実行され、タイムスライスが発生しないスケジューリングである。SCHED_RR は同一の優先度のプロセス間でタイムスライスが行われるスケジューリン

グである。プロセスの優先度は 1~99 が指定でき、99 が最も優先度が高い。パイプラインの各コマンドを起動する際に、スケジューリングポリシーと優先度を設定する。

CPU 固定と優先実行の指定は、シェル変数 BASH_NO_MIGRATION で行う(図 3-6)。



CPU 固定	0	CPU 固定しない
	1	CPU 固定する
優先実行	0	優先実行しない
	1	優先実行する (SCHED_FIFO)
	2	優先実行する (SCHED_RR)
静的優先度	1~99	範囲外の値は 1 または 99 として扱う

図 3-6　CPU 固定と優先実行の指定方法

4. 評価及び今後の課題

現在は USP 向け機能を実装したシェルを実際の業務アプリケーションに適用しているところで、今回実装した機能の評価はこれから行うところである。性能に関する部分の評価は注意を要する。パイプラインの実行効率向上は、対象のシェルスクリプト以外のプロセスがある場合に効果が発揮される。このため、実際の運用環境或いはそれをシミュレートする環境で効果を測定する必要がある。

今回の実装は、現在 USP 手法で用いているシェルである **bash** の上で行った。しかし、**bash** はインタラクティブ機能が非常に強力であり、スクリプト言語インタプリタとして見ると非常に重たく、実行性能を損なっている。このため、**ash** 等のより軽いシェルを利用したり、スクリプト実行専用のシェルを新規作成したりすることによって実行性能を向上させることが期待できる。

参 考 文 献

1)當仲寛哲：シェルスクリプトによるプロセス指向型開発手法, *SEA Forum April 2007*

2)Mike Gancarz: UNIX という考え方, オーム社

3)Brian W. Kernighan, John R. Mashey: The Unix Programming Environment, *Computer April 1981*, pp.12-24