データ部の初期化によるプロセス再起動機能の提案と評価

田端利宏† 谷口秀夫†

プログラムの実行単位であるプロセスの生成処理は，オペレーティングシステムの処理の中でも負荷の大きい処理である．このため，従来から，プロセスの生成処理を高速化できる技術がある．たとえば，UNIX では sticky bit や vfork システムコール，さらに Demand Paging や Copy on Write がある．一方，プログラムの中には，繰り返し実行されるものが少なくない．たとえば，UNIX の make プログラムを利用したコンパイル処理では，繰り返しコンパイラを実行するため，プロセスの生成と削除が頻発する．我々は，プロセスを繰り返し生成し削除する処理に着目し，プロセスを高速に再起動させる機能を提案する．プロセスを再起動させることにより，プロセスの生成と削除のオーバヘッドを削減でき，処理を効率化できる．本論文では，プロセス再起動機能について述べ，その基本性能の評価結果と効果予測の結果について報告する．

Proposal and Evaluation of Process Restart Function by Initializing Data Segments

Toshihiro Tabata† and Hideo Taniguchi†

An operating system controls many processes to execute programs. The processing of process creation and process termination has a heavy load in an operating system. Therefore there are many techniques that can speed up process creation; for example, sticky bit and the vfork system call are realized in UNIX. Furthermore, demand paging and copy-on-write are realized. Generally, specific programs are often executed repeatedly. For example, in the "make" command of UNIX, process creation and process termination are repeated, because a compiler is executed many times. We propose the function for restarting a process. The function is effective where specific programs are executed repeatedly, because the function can reduce the overhead of process creation and process termination. This paper describes the structure of a process and the function for restarting a process. This paper also reports the performance of the function.

1. はじめに

オペレーティングシステム（以降，OS と略す）は，プロセスを単位としてプログラムの実行を制御する．OS はプログラムを実行するために，プロセスを生成し，その実行状態を制御する．そして，プログラムの実行が終了するとプロセスを削除する．このように，プロセスの生成と削除の処理は，プログラムの実行に関わる必須な処理である．

プロセスの生成処理では，プロセスのメモリ空間の作成，メモリ空間へのプログラムの複写処理，およびプロセス管理表のエントリの確保と初期化を行う．このとき，メモリ間のデータ複写やディスクからメモリへのデータ複写が発生する．また，プロセスの削除処理では，プロセスの利用していた資源の解放，プロセスのメモリ空間の削除，およびプロセス管理表のエントリの解放を行う．このため，プロセスの生成と削除の処理は，処理のオーバヘッドが大きい．したがって，プロセスの生成と削除の処理を高速化する必要がある．

従来から，プロセスの生成を高速化する研究が行われている．たとえば，UNIX[1)] では，sticky bit や vfork システムコールを実現し，プロセスの生成処理を高速化している．さらに，Demand Paging（ODP）や Copy on Write（CoW）の実現により，プロセスの生成処理を高速化している．sticky bit では，プロセスの削除時にテキスト部をメモリやスワップ領域に残しておき，次に同じプログラムからプロセスを生成するときに，残しておいたテキスト部の内容を利用することでプロセスの生成処理を高速化する．さらに，vfork システムコールでは，親プロセスとメモリ空間を共有することで，プロセスの生成処理を高速化して

† 九州大学大学院システム情報科学研究院
Faculty of Information Science and Electrical Engineering, Kyushu University

いる．また，プロセスを軽量化したスレッドを単位として，プログラムの実行を制御する研究も行われている[2)~5)]．

一方，既存の OS，たとえば UNIX では，数多くのプログラムがファイルシステム上に存在するものの，実際に頻繁に利用されるプログラムはそのうちの一部のプログラムだけである．具体的には，ls や ps などの利便性の高いプログラムが頻繁に実行される．また，make プログラムを利用したコンパイル処理では，コンパイラを繰り返し実行するためにプロセスの生成と削除が繰り返される．さらに，Web サーバの Apache では，クライアントからのアクセス集中時に多くの接続要求を受付けるため，プロセスを繰り返し生成する．たとえば，Web サーバが CGI プログラムを実行する際には，要求があるごとにプロセスを繰り返し生成し実行することが多い．したがって，アクセスが集中する場合には，計算機の負荷が高くなるため，プロセス生成処理の負荷が Web サーバのスループット低下の要因となることが考えられる．このように同じプログラムを繰り返し実行する場合には，1 つのプログラムから繰り返しプロセスを生成する．このとき，OS 内部では，同じプログラムによるプロセスを生成し削除する処理が繰り返される．

本論文では，特定のプログラムが繰り返し実行されることが多いことに着目し，一度実行したプロセスを再起動させ，高速にプロセスの実行を開始できる機能を提案する．プロセスの再起動処理では，プロセスの処理終了時に，プロセスを非スケジューリング対象とし，メモリ空間上に残しておく．それから，残しておいたプロセスと同じプロセスを生成するときに，プロセスのテキスト部をそのまま利用し，データ部とプロセス管理表のエントリを初期化し，プロセスを最初から再実行できる環境を構築する．なお，本論文で対象とするプログラムは，動的リンクライブラリを利用しない静的リンクの応用プログラム（以降，AP と略す）である．

本論文で提案するプロセス再起動機能と類似した機能を持つシステムとして Amoeba[6)] がある．Amoeba は，プロセスを一時停止させる機能を持ち，この機能をプログラムのデバッグやプロセスマイグレーションに利用している．このため，停止位置からの再開ができる．一方，本論文で提案するプロセス再起動機能を実現したシステムでは，プロセスを停止させ，要求に応じて再起動させる．つまり，Amoeba と同様にプロセスを停止させるが，プロセスの再起動に重点をおいている．

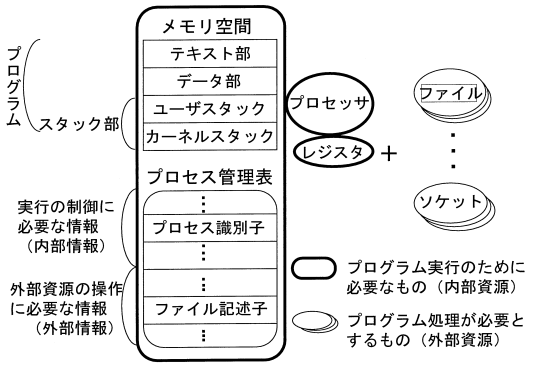

図 1 プロセスの構成要素
Fig. 1 Elements of process.

以降，プロセスの構造について述べ，プロセス再起動機能を提案する．また，プロセス再起動機能を実装した ***Tender*** オペレーティングシステム[7)] の特徴について述べる．さらに，プロセス再起動機能を実現するために追加した提供インタフェース，および変更点について述べ，プロセス再起動機能の基本性能の評価結果を報告する．

2. プロセスの生成と削除

2.1 プロセスの構造

プロセスとは，OS がプログラムを実行するとき，動作を制御する基本単位である．OS は，プロセスを生成し，その状態を制御し，終了させる．また，プロセスの状態を，プロセス管理表により管理する．

プロセスの構成要素を**図 1** に示す．プロセスの構成要素には，プログラム，プロセス管理表，プログラムの実行のために必要なもの（これを内部資源と呼ぶ），プログラムの処理が必要とするもの（これを外部資源と呼ぶ）がある．プログラムは，テキスト部，データ部，スタック部（ユーザスタック，カーネルスタック）からなる．テキスト部は，プロセッサが実行可能な命令の列である．データ部は，初期値を持つ変数や文字列の集合部分と，初期値を持たない変数の集合部分からなる．後者は BSS 部と呼ばれている．スタック部には，ユーザスタックとカーネルスタックがある．ユーザスタックは，プロセスがユーザモードで走行するときに利用され，カーネルスタックは，プロセスがカーネルモードで走行するときに利用される．プロセス管理表が持つ情報は，実行の制御に必要な情報（内部情報）とプログラム処理が必要とする外部資源の操作に必要な情報（外部情報）に分類できる．内部資源には，メモリ空間，プロセッサ，レジスタなどがあり，外部

資源には，ファイルやソケットなどがある．

2.2 問 題 点

プロセスの生成処理には，処理のオーバヘッドが大きいという問題がある．これは，メモリ空間を生成し，プログラムをメモリ空間に読み込む際のメモリ間のデータ複写や，ディスクからメモリへのデータ複写に起因する．また，プロセスの削除処理においても，内部資源や外部資源の解放を行うため，処理のオーバヘッドが大きい．

3. プロセスの再起動機能

3.1 基 本 機 能

プロセス再起動機能とは，実行を停止したプロセスのデータ部，およびプロセスの管理表のエントリを初期化し，プロセスを最初から再実行させる機能である．

プロセス再起動機能は，大きく分けると 2 つの処理を行う．1 つは，プロセスの実行を停止する処理である．もう 1 つは，プロセスのデータ部，およびプロセス管理表のエントリを初期化し，プロセスを最初から再実行できる環境を構築する処理である．ここでは，プロセスの実行を停止し，非スケジューリング対象とした状態を，再起動可能状態と名付ける．

図 2 に，プロセスの再起動処理と，プロセス再起動機能と同等の処理をプロセスの生成と削除で実現した処理を示し，プロセスの再起動処理の流れを説明する．プロセスの再起動処理は，プロセスの実行を停止する処理，およびデータ部を読み込み，プロセス管理表のエントリを初期化し，プロセスを最初から走行させる処理からなる．一方，プロセス再起動機能と同等の処理は，プロセスの削除と生成の処理で実現できる．プロセスの削除と生成の処理では，プロセスを削除した後，新たに生成するプロセス用にプロセス管理表のエントリを確保する．次に，メモリ空間を作成し，テキスト部とデータ部を読み込み，プロセス管理表のエントリを初期化する．

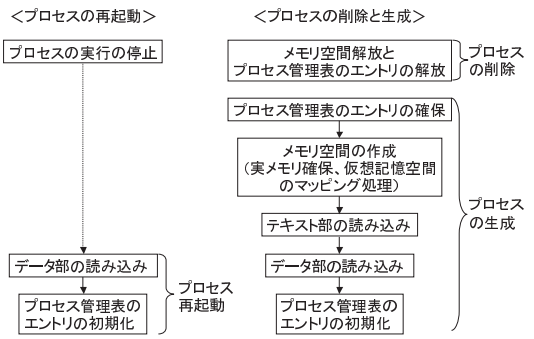

図 2 同一プログラムを再実行する処理の流れ
Fig. 2 Processing flow of program re-execution.

なお，プロセスの生成処理として大きく 2 つの方式がある．1 つは，プロセスの生成時に，メモリ空間にプログラムを読み込む方式（メモリ常駐方式）である．もう 1 つは，プロセスの生成時に，メモリ空間にプログラムを読み込まず，プロセスの実行時にページ例外により必要とするページを読み込む方式（ODP 方式）である．プロセスの再起動処理は，プロセスの生成の両方式に比べ，プロセスの削除，プロセス管理表のエントリの確保，メモリ空間の作成を行わないため高速である．プロセス再起動処理がメモリ空間の作成を行わないのは，再起動前に利用していたアドレス変換表を再利用できるためである．また，プロセスの再起動処理は，メモリ常駐方式に比べ，テキスト部の読み込み処理を行わないため高速である．さらに，ODP 方式ではプロセスの生成時に実メモリを確保しないため，ページ例外処理による実メモリの確保が必須である．プロセス再起動処理は，再起動前に確保していた実メモリをそのまま再利用できることが期待できるため，ページ例外の回数は ODP 方式より少ないことが期待できる．ただし，ODP 方式には必要最小限のメモリしか必要としないという長所がある．もちろん，プログラムが保有するデータ部全体の大きさに対する利用割合の多少により，影響を受ける．

3.2 特徴，制約，および利用するか否かの判断方法

プロセスの再起動機能には，以下の特徴がある．

(1) プロセスを作り直さずに再起動することで，高速に処理を開始することができる．
(2) プロセス識別子は，プロセス再起動の前後で不変である．これにより，プロセス識別子を宛先としてプロセス間通信を行う協調処理では，一部のプロセスの障害による再起動を他のプロセスから隠蔽することが可能である．

また，再起動の前後でプロセス識別子は不変であることから，プロセスの再起動機能を利用するには，応用プログラムに制約が生じる場合がある．プロセス識別子を一時ファイルなどの名前に利用する AP の場合，再起動後に一時ファイルなどの名前が衝突することがある．AP がこれらのファイルをそのまま利用すると処理を正常に行えないことがある．このため，プロセス識別子を一時ファイルなどの名前に利用しないといった対処が AP に必要となる．

上記の特徴を生かしてプロセス再起動機能を利用する場合，コンパイラ，AP，および OS の各々において，プロセス再起動機能を利用するか否かを判断する方法が考えられる．以下に，利用するか否かを判断す

る方法について簡単に述べる．

（方法 1） コンパイラが，AP のコンパイル時にプロセスの生成頻度を解析し，プロセスの生成頻度の高い部分については，プロセス再起動機能を利用するように書き換える．ただし，実行頻度を詳しく解析するのは難しいという問題がある．

（方法 2） AP が，プロセスの生成頻度が高いプロセスについて，明示的に再起動を指定する．

（方法 3） OS が，実行プログラム名と実行回数のログをとり，プロセスの生成回数が多いプログラムに関しては，実行終了時に再起動可能状態に移行させる．再起動可能状態にあるプロセスと同じプログラムをプロセスとして生成する場合には，OS 内部でプロセスの生成処理をプロセスの再起動処理に置き換える．

なお，いずれの方法においても，再起動可能状態のプロセスを残すことは，計算機資源を余計に消費することを意味する．したがって，資源の消費を最小限に抑えるために，効率良くプロセスを再起動させる必要がある．たとえば,（方法 3）の場合，再起動可能状態にあるプロセスの数や量を LRU などの管理法により管理し，必要に応じて，再起動可能状態のプロセスを消去することも考えられる．しかし，プロセッサ性能の高性能化，およびメモリ価格の低価格化により，再起動可能なプロセスを残すためのプロセッサ処理時間は短くなり，計算機に搭載されるメモリの量も増大する傾向にある．このため，プロセス再起動のために余剰なメモリを確保できることが期待できる．また，余剰なメモリを活用することで，プロセス再起動を有効に利用できることが期待できる．さらに，メモリが不足しても，再起動可能状態のプロセスを解放することでメモリ資源を転用できる．

4. 実装と評価

4.1 実 装 内 容

4.1.1 項では，既存の ***Tender*** の特徴について述べる．次に，4.1.2 項では，プロセス再起動機能を ***Tender*** オペレーティングシステム[7] に実装するための変更点，および追加した提供インタフェースについて述べる．

4.1.1 *Tender* オペレーティングシステム

＜資源の分離と独立化＞

Tender では OS の操作する対象を資源として，分離し独立化した．資源には，資源名と資源識別子を付与し，資源操作のインタフェースを統一している．さらに，各資源を操作するプログラム部品を資源ごとに分離し，共有プログラムを排除した．また，各資源の管理情報も資源ごとに分離し，各資源の管理表の間のポインタを禁止した．このように，資源の分離と独立化を行うことで，資源の事前生成や保留により，資源の作成や削除をともなう処理を高速化している．さらに，プログラムを部品化できるため，機能の追加や変更が容易である．

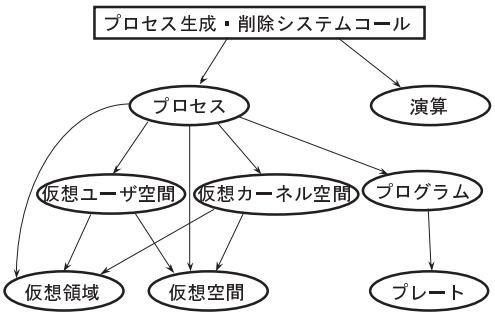

図 3 プロセスの生成と削除時の資源の呼び出し関係
Fig. 3 Relation of resources on process creation and termination.

OS の資源の分離と独立化の例として，プロセスの生成と削除時の資源の処理の呼び出し関係を**図 3** に示す．図 3 の矢印は，資源間の処理の呼び出し関係を表している．プロセスの資源を分離し独立化した結果，プロセスとプロセスの構成要素の関係が明確になり，プロセスの構成要素は，プロセスの存在に関係なく存在することが可能となっている．たとえば，UNIX の場合，プロセスを削除すると，利用していたメモリ空間を解放しなければならないため，メモリ空間の再利用はできない．***Tender*** では，プロセスを削除するときに指定したメモリ空間を残すことができる．さらに，プロセスを生成するときに，そのメモリ空間を再利用し，プロセスの生成処理を高速化できるようにした．

＜資源「プログラム」＞

分離し独立化した資源の例として，プロセスの生成と削除処理に関連の深い資源「プログラム」について説明する．資源「プログラム」は，プログラムの「テキスト部の大きさ」,「テキスト部の先頭番地」,「データ部の大きさ」,「データ部の先頭番地」，および「プログラムの開始番地」の情報を持ち，プログラムデータの実行形式を隠蔽し，プログラムをメモリ上に存在させる．したがって，資源「プロセス」の管理部は，プログラムデータの内容を意識することなく処理を実行できる．さらに，プログラムを資源化したことで，プロセスの存在に関係なくプログラムがメモリ上に存在できる．このため，すでにメモリ上に存在するプログラ

表 1 プロセス再起動機能のために追加したインタフェース
Table 1 Additional interface for process restarting.

形式	機能
setrestart(path)	本システムコール以降，プログラム path から生成されたプロセスは，実行終了時に再起動可能状態に移行する．
unsetrestart(path)	本システムコール以降，プログラム path から生成されたプロセスは，実行終了時に削除する．また，本システムコールにより，以下に示す（条件 1）または（条件 2）に合うプログラム path から生成された再起動可能状態のプロセスをすべて削除する．（条件 1）資源の引き継ぎあり，かつ実行終了時の利用者が本システムコールを発行した利用者と同じ．（条件 2）資源の引き継ぎなし，かつ他の利用者が同一の path について再起動可能状態に移行させる設定を行っていない．
getrestartpid(path)	プログラム path から生成された再起動可能状態プロセスの pid を 1 つ返す．なお，pid は，procrestart(pid, argv) が発行されるまで，仮予約される．当該再起動可能状態のプロセスがない場合は 0 を返す．
procrestart(pid, argv)	再起動可能状態のプロセス pid を再起動させる．argv はプロセスに渡す引数へのポインタである．

ムを利用して，プロセスの生成処理を高速化できる．

4.1.2 提供インタフェース

プロセス再起動機能を実現するために，**表 1** に示すインタフェースを追加した．また，これらのインタフェースを実現するための機能をプロセス管理に追加し，プロセスの状態に再起動可能状態を追加した．さらに，プロセスの処理終了時に再起動可能状態に移行するために，プロセスを削除するシステムコールの先頭に再起動可能状態に移行するか否かを判断する条件文を追加し，条件が真のときには，プロセスを再起動可能状態に移行させる処理を追加した．複数の個所に機能を追加する必要があったため，追加個所は散在しているけれども，追加した主要な個所の C 言語の行数は，約 600 行であった．

Tender では，資源の分離と独立化機構を持つため，再起動の前後での外部資源の継続利用を実現できる．具体的には，プロセス終了時にプロセスが確保していた外部資源に関する情報を，再起動後のプロセスに引数として渡すことで実現する．これにより，再起動前と再起動後の利用者が同一である必要がある AP は，外部資源を引き継いだか否かを判断でき，渡された情報を利用して外部資源をそのまま利用できる．ただし，外部資源の不正な利用を防ぐためには，再起動前と再起動後のプロセスの所有者が同一である必要がある．また，AP は再起動後に継続利用できる資源の有無を確認し，外部資源を利用するような対処が必要である．

表 1 に示した提供インタフェースの機能について説明する．setrestart(path) は，再起動可能状態に移行させるプログラムを登録する．unsetrestart(path) は，再起動可能状態に移行させるプログラムの登録を解除する．同時に，資源の引き継ぎがあり，かつ実行終了時の利用者が本システムコールを発行した利用者と同じである再起動可能状態のプロセスをすべて削除する．資源の引き継ぎなしの再起動可能状態のプロセスについては，他の利用者が再起動させてもよいため，他の利用者が同一プログラム path に対し再起動可能状態に移行させる設定を行っている場合のみ残すこととする．これにより，他の利用者と資源の引き継ぎなしの再起動可能状態のプロセスを共有できる．また，他の利用者が当該プログラムについて再起動機能を利用しない場合には，再起動可能状態のプロセスを削除し，資源の消費を抑制できる．getrestartpid(path) は，プログラム path から生成された再起動可能状態のプロセスの識別子を 1 つ返す．procrestart(pid, argv) は，プロセス pid を再起動させる．また，setrestart と unsetrestart による設定内容は，利用者ごとに保持することとする．これは，先に述べたように，プロセスの再起動の前後で資源の引き継ぎを行う場合には，再起動前と再起動後の利用者が同一である必要があるため，getrestartpid の処理において再起動前後で利用者が同一であるかを確認する必要があることに起因する．

表 1 のインタフェースを利用したプロセスの再起動処理の流れの例として，AP が利用を判断する場合について，**図 4** に示し，以下に説明する．

(1) 繰り返し再起動させるプログラム（path）を実行する前に，setrestart(path) を発行する．
(2) getrestartpid(path) を発行し，プログラム path から生成された再起動可能状態のプロセスが存在するか否かを判断する．
(3) 再起動可能状態のプロセス pid が存在するとき（pid $\neq$ 0），procrestart(pid, argv) を発行する．再起動可能状態のプロセスが存在しないとき（pid==0），プロセスを生成する（***Tender*** では，proccreate(path) を発行する）．
(4) 必要に応じて，(2) と (3) の処理を繰り返す．

```
                    ≀
(1)  setrestart(path);
                    ≀
(2)  if ((pid = getrestartpid(path)) == 0){
       pid = proccreate(path, argv);
(3)  } else procrestart(pid, argv);
                    ≀
(4)  unsetrestart(path);
                    ≀
```

図 4 プロセスの再起動処理の流れ（AP が利用を判断する場合）
Fig. 4 Processing flow of restarting process on AP.

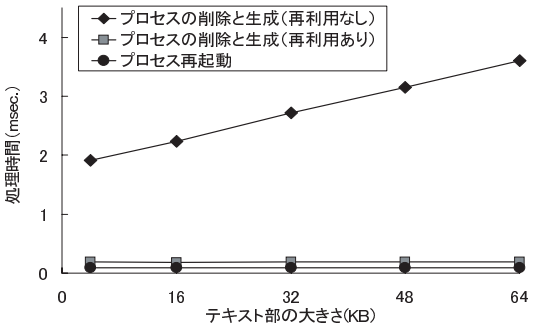

図 5 テキスト部の大きさと処理時間の関係
Fig. 5 Relation of text segment size and processing time.

(5) 繰り返し再起動させるプログラムの実行が不要になると，unsetrestart(path) を発行する．

なお，図 4 の例では，再起動させたプロセスの処理が終了する前に，同一のプログラム path に対し，プロセス生成の要求がある場合，同一プログラム path から生成された再起動可能状態のプロセスがあれば，そのプロセスを再起動させる．そうでなければ，新たにプロセスを生成する．

4.2 基本性能

プロセス再起動機能の性能を明らかにするため，プロセス再起動の処理時間と，プロセスの削除と生成の処理時間を比較する．なお，プロセスを起動する前後における資源の引き継ぎは，プロセス再起動の場合は可能であるが，プロセスの削除と生成の場合には不可能である．このため，プロセス再起動の処理時間は，資源を引き継がない場合の時間とした．また，***Tender*** では，プロセスの削除と生成において，プロセスを構成する資源を再利用することが可能である[7]．このように資源を再利用するとプロセスの削除と生成の処理時間を短縮できる．そこで，以降では，プロセスの削除と生成の処理時間として，プロセスの構成資源をすべて再利用する場合（以降，再利用ありと呼ぶ）とそうでない場合（以降，再利用なしと呼ぶ）の処理時間を示し，プロセス再起動の処理時間と比較する．また，処理時間の比較を明確にするため，測定では，資源「プログラム」を利用してプログラムをメモリ上に常駐させ，処理時間にディスクとの実入出力時間を含まないようにした．***Tender*** のプロセスの生成処理は，メモリ常駐方式で実現している．なお，システムコールの呼び出しによるオーバヘッドは約 1 μsec であった．

プロセス再起動の処理時間として，procrestart システムコールの処理時間を測定した．また，プロセスの削除と生成の処理時間として，procdelete システムコール（***Tender*** でのプロセスを削除するシステムコール）と proccreate システムコールの処理時間を測定した．測定に使用した計算機は，Pentium II 450 MHz の計算機で，ハードウェアクロックのカウンタを利用して処理時間を測定した．

テキスト部の大きさを可変にし，データ部と BSS 部の大きさを 4KB に固定した場合の測定結果を**図 5** に示す．プロセスの再起動処理と，プロセスの削除と生成処理（再利用あり）は，テキスト部をそのまま利用できるため，テキスト部の大きさにかかわらずほぼ一定の処理時間（約 90 μsec，約 190 μsec）である．また，プロセスの再起動処理は，プロセスの削除と生成処理（再利用あり）に比べ，約 100 μsec 高速である．これは，プロセスの再起動処理では，プロセス状態を再起動可能状態に変更するだけでプロセスをそのまま残すことができ，データ部の初期化とプロセス管理表のエントリの初期化でプロセスを再起動できるからである．これに対し，プロセスの削除と生成（再利用あり）の場合は，プロセスの構成資源を再利用できるものの，プロセスの削除時には再利用可能な資源の登録処理およびプロセス管理表のエントリの解放処理が必要である．また，プロセスの生成時には再利用可能なプロセスの構成資源の検索処理を必要とする．さらに，プロセスの削除と生成（再利用あり）の場合は，システムコール呼び出し回数が 1 回多い．一方，プロセスの削除と生成処理（再利用なし）は，プロセスの生成時にテキスト部の内容を読み込むためのメモリ間データ複写が必要である．このため，テキスト部の大きさに比例した処理時間がかかる．

次に，データ部の大きさを可変にし，テキスト部と BSS 部の大きさを 4KB に固定した場合の測定結果を**図 6** に示す．すべての場合で，データ部の内容を読み込むためのメモリ間データ複写が必要であるため，データ部の大きさに比例した処理時間がかかる．ただ

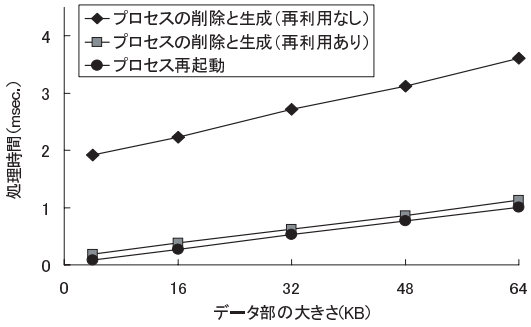

図 6 データ部の大きさと処理時間の関係
Fig. 6 Relation of data segment size and processing time.

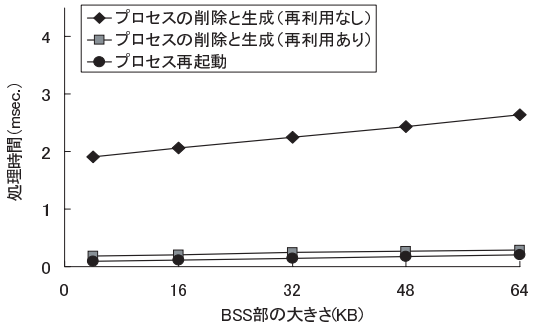

図 7 BSS 部の大きさと処理時間の関係
Fig. 7 Relation of BSS segment size and processing time.

し，プロセスの再起動処理と，プロセスの削除と生成処理（再利用あり）では，アドレス変換表を再利用できるため，データ部の仮想アドレスに対応するアドレス変換表のエントリに，仮想アドレスに対応する物理アドレスをマッピングする処理が必要ない[7]．このため，プロセスの削除と生成処理（再利用なし）よりも，データ部の大きさの増加に対する処理時間増加の割合が小さい．プロセスの再起動処理は，プロセスの実体を削除せず，システムコールの呼び出し回数が 1 回少ない．このため，プロセスの削除と生成処理（再利用あり）に比べ，約 100 μsec 高速である．

最後に，BSS 部の大きさを可変にし，テキスト部とデータ部の大きさを 4KB に固定した場合の測定結果を**図 7** に示す．BSS 部は，初期値を持たない変数の集合部であるため，値 0 で内容を初期化する．このため，メモリ間データ複写の必要なデータ部の初期化よりも処理が高速である．また，すべての場合で BSS 部の大きさに比例した処理時間がかかる．プロセスの再起動処理とプロセスの削除と生成処理（再利用あり）は，BSS 部用のメモリ空間へのマッピング処理が必要ないため，プロセスの生成と消滅処理（再利用なし）よりも，BSS 部の大きさの増加に対する処理時間増加の割合が小さい．プロセスの再起動処理は，プロセスの実体を削除せず，システムコールの呼び出し回数が 1 回少ないため，プロセスの消滅と生成処理（再利用あり）に比べ，約 100 μsec 高速である．

以上の評価をまとめると，プロセス再起動の処理時間は，プロセス削除と生成（再利用あり）の処理時間に比べ，約 100 μsec だけ短い．また，プロセスの再起動処理は，プロセスの実体を削除せず利用するので，特定のプログラムに対する処理時間はつねに一定である．資源を再利用したプロセス生成は，プロセス削除時に残したすべてのプロセスを構成する資源をプロセス生成時に必ず再利用できるとは限らないため，その処理時間はつねに一定ではない．これは，図 3 に示すように，再利用できる資源の単位が，仮想空間，仮想領域などのプロセスを構成する資源であるためである．たとえば，資源再利用のために残したプロセスを構成する資源のうち，仮想空間が他プロセスの生成に再利用され，他に再利用可能な仮想空間がない場合，目的とするプロセス生成には，仮想空間以外の資源が再利用されることとなり，すべての資源を再利用できたときに比べ，処理時間は長くなる．したがって，特定のプログラムのプロセスの生成処理において，必ず最大限の効果をあげるためには，プロセス再起動機能を利用する必要がある．なお，プロセスを構成する各資源の再利用によるプロセスの削除と生成処理の高速化の効果は，文献 7) で報告している．

プロセスの再起動の処理時間 (y_1)，プロセスの削除と生成（再利用あり）の処理時間 (y_2)，およびプロセスの削除と生成（再利用なし）の処理時間 (y_3) を測定結果から定式化した結果を以下に示す．テキスト部，データ部，および BSS 部の大きさ (KB) を x_1，x_2，x_3 とする．

$$y_1(\mu\text{sec}) = 15.5x_2 + 2x_3 + 20 \quad (1)$$
$$y_2(\mu\text{sec}) = 15.5x_2 + 2x_3 + 120 \quad (2)$$
$$y_3(\mu\text{sec}) = 28x_1 + 28x_2 + 12x_3 + 1640 \quad (3)$$

4.3 既存 OS との比較

既存 OS の例として，UNIX（ここでは，BSD/OS[8] を取り上げて説明する）を例にあげ，プロセスの再起動処理と UNIX のプロセスの生成処理を比較し，UNIX におけるプロセス再起動の適用範囲について述べる．

プロセス再起動では，プロセス自身を初期化し，プロセスを新規に生成したときと同じ環境を作る．一方，UNIX では，fork/vfork システムコールで，現在のプロセスの複製を生成する．それから，必要に応じて exec システムコールで，新しいプログラムを読み込む．また，UNIX では，fork/vfork システムコール処理終

了後，exec システムコール発行するまでの間に，他の処理（たとえば，ファイル記述子に関する処理）を実行できる．しかし，プロセス再起動機能は，1 つのシステムコールで実現されているため，fork/vfork システムコールと exec システムコールの間で他の処理を実行する場合にプロセス再起動を適用できない．つまり，fork/vfork システムコールの直後に，exec システムコールを発行する場合にプロセス再起動を適用できる．また，このような場合は，後述するように make 処理に多く，プロセス再起動を多く適用できる．

そこで，fork/vfork システムコールの直後に exec システムコールが発行され，プロセスの再起動処理に置換可能な処理を仮定し，BSD/OS のプロセスの生成処理時間を測定する．4.2 節の測定と同じ計算機で，BSD/OS 3.1 のプロセスの生成と消滅の処理時間を測定した．なお，UNIX のプロセスの生成処理は，CoW と ODP により遅延書き込みを実現しているため，プロセスの大きさに関係なくほぼ一定の処理時間で終了する．そこで，測定では，親プロセスが vfork システムコールを発行し，wait4 システムコールで子プロセスの実行終了を待ち，制御が戻るまでの時間を測定した．子プロセスは，exec システムコールの発行後，すぐに終了する．処理を 11 回繰り返し，その処理時間の平均値を算出した．ただし，1 回目の処理時間は，ディスク I/O の影響があるため，平均値の算出からは除いた．vfork システムコールと exec システムコールの処理時間の平均値は，1,432.7 μsec であった．

4.4 make プログラムでの効果予測

4.4.1 make プログラムの処理の流れ

プロセスの生成と削除が頻発する処理の例として，make プログラムを利用した BSD/OS 3.1 カーネルのコンパイル処理について，プロセス再起動機能の効果を予測する．**表 2** に，プロセス生成システムコールを発行するプログラムの名前，およびプロセスの生成回数を示す．また，**表 3** に，BSD/OS 3.1 カーネルの make 処理で実行されるプログラムの名前，大きさ，および実行回数を示す．表 2 より，make プログラムと gcc2 プログラムにおいてプロセスの生成回数が多いことが分かる．具体的には，make プログラムは，gcc2 プロセスを 272 回生成し，gcc2 プログラムは，cpp，as，cc1 プロセスを各々272 回生成する．プロセス再起動機能は，繰り返し実行されるプログラムに対して有効であるため，gcc2，cpp，as，cc1 プロセスを各々272 回生成する処理について，プロセス再起動機能を利用したときの効果を予測する．

表 2 BSD/OS カーネルの make 処理でのプロセスの生成回数
Table 2 Number of process creation in making BSD/OS kernel.

プログラム名	プロセスの生成回数
sh	35
make	288
gcc2	816
cpp	15
合計	1,154

表 3 BSD/OS カーネルの make 処理で実行されるプログラム
Table 3 Programs executed in making BSD/OS kernel.

プログラム名	テキスト部	データ部	BSS 部	実行回数
cpp	36,864	4,096	13,092	282
as	53,248	16,384	34,964	277
cc1	970,752	40,960	67,324	272
gcc2	40,960	4,096	0	272
sh	118,784	8,192	3,348	14
[	3,472	184	72	11
rm	2,800	140	28	8
ln	912	140	16	4
ar	12,288	4,096	0	2
cat	1,956	132	36	1
chmod	1,348	140	0	1
date	3,204	336	12	1
expr	3,876	2,256	24	1
hostname	488	140	0	1
mv	1,500	136	8	1
ld	28,672	4,096	620	1
pwd	360	140	0	1
ranlib	5,136	288	380	1
size	3,516	180	360	1
strip	2,480	136	0	1
touch	2,588	128	0	1
合計	—	—	—	1,154

（テキスト部，データ部，BSS 部の大きさの単位はバイト）

4.4.2 効 果 予 測

プロセスの再起動処理とプロセスの生成と削除処理について，次に示す方法で処理時間を算出した．図 4 に示したプロセスの再起動処理を実現した場合のプロセス再起動の処理時間として，getrestartpid システムコールと procrestart システムコールの処理時間を求めた．具体的には，getrestartpid システムコールを ***Tender*** に実装し，1 回あたりの平均処理時間 (3.4 μsec) を測定した．また，procrestart システムコールの処理時間は，式 (1) と表 3 から算出した．***Tender*** のプロセスの生成処理（再利用ありとなし）については，処理時間を式 (2)，式 (3)，および表 3 から算出した．BSD/OS 3.1 のプロセスの生成については，処理時間を 4.3 節の実測値とプロセスの生成回数から算出した．

BSD/OS 3.1 カーネルの make 処理におけるプロセスの生成に関する処理時間を算出した結果を**表 4** に

表 4 BSD/OS 3.1 カーネルの make 処理におけるプロセスの生成に関する処理時間
Table 4 Processing time of process creation in making BSD/OS kernel.

測定方法	処理時間（秒）
プロセス再起動（***Tender***）	0.357
プロセス生成（***Tender***，再利用あり）	0.462
プロセス生成（***Tender***，再利用なし）	10.834
プロセス生成（BSD/OS，vfork & exec）	1.559
スレッド生成（BSD/OS）	0.656

示す．表 4 より，プロセスの生成（資源再利用なし）(10.834 sec) に比べ，プロセス再起動 (0.357 sec) は，約 30 倍高速であることが分かる．表 4 より，プロセス生成（資源再利用あり）(0.462 sec) に比べ，プロセス再起動 (0.357 sec) は，1.3 倍高速であることが分かる．これは，プロセスの実体を消さずにそのまま利用してプロセスを再起動させるためである．

次に，BSD/OS 3.1 でのプロセスの生成処理時間を測定し，BSD/OS 3.1 カーネルの make 処理におけるプロセスの生成に関する処理時間を算出した．4.3 節で測定した 1 回あたりの BSD/OS のプロセスの生成の処理時間から，cpp，as，cc1，gcc2 をそれぞれ 272 回生成する処理時間は $1{,}432.7\,\mu\text{sec} \times 272 \times 4 \simeq 1.559\,\text{sec}$ となる．

さらに，BSD/OS 3.1 のカーネルの make 処理において，プロセス生成処理をスレッドの生成処理に置き換えたと仮定した場合について，スレッドの生成に要する処理時間を算出した．BSD/OS では，遅延書き込みを実現しているため，プロセスと同様にスレッドの生成時間も，プロセスの大きさに関係なくほぼ一定である．先の測定と同じ条件で，BSD/OS でのスレッドの生成時間を測定したところ，602.5 μsec であった．スレッドを 272×4 回生成したときの処理時間は，$602.5 \times 272 \times 4 \simeq 0.656\,\text{sec}$ となる．

BSD/OS と ***Tender*** では，仮想記憶やプロセス生成の実現方式に違いがあるものの，参考として BSD/OS での測定結果を載せた．

4.5 ベンチマークプログラムによる実測評価

プロセス再起動機能による効果を明らかにするため，簡単なベンチマークプログラムを作成し，BSD/OS と ***Tender*** 上で測定した．ベンチマークプログラムは，親プロセスが，子プロセスを生成し，子プロセスに処理を行わせ，その終了を待つ処理を繰り返すものである．子プロセスの生成回数を 1,000 回とし，子プロセスの処理時間を約 5 msec，約 10 msec，約 20 msec と変化させ測定した．子プロセスの処理は，データ部のすべての領域に値を繰り返し書き込む処理とした．このときの子プロセスの大きさは，テキスト部 20 KB，データ部 20 KB，BSS 部 0 KB であった．なお，測定には，Pentium II 450 MHz の計算機を利用した．

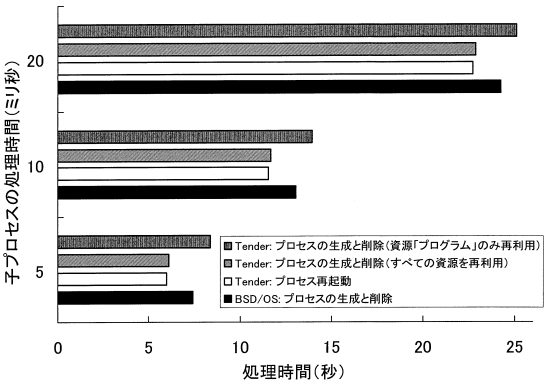

図 8 ベンチマークプログラムの実測結果
Fig. 8 Results of benchmark program.

実測した結果を**図 8** に示す．図 8 から，プロセス再起動機能を利用した場合の処理時間が最も高速であることが分かる．高速化された処理時間は，子プロセスの処理時間の長さに関係なくほぼ一定である．このことから，高速化された時間は，全体の処理におけるプロセスの生成回数に比例すると推察できる．つまり，プロセスの生成回数が多いほど，プロセス再起動機能の効果が大きくなるといえる．

さらに，プロセス再起動は，プロセス生成と削除（再利用あり）と比較すると，1,000 回あたり 0.13 秒高速である．この結果は，効果予測 (0.105 秒 $= 0.462 - 0.357$) と同等の結果となっている．また，プロセス再起動は，プロセス生成と削除（再利用なし）よりも，約 2.4 秒高速である．このとき，プロセス生成と削除（再利用なし）の処理時間は，効果予測 (10.477 秒 $= 10.834 - 0.357$) に比べ処理時間が短い．これは，子プロセスのテキスト部と BSS 部が小さいため，効果予測の場合と比べ，プロセス生成の処理時間が短くなったためである．

5. ま と め

本論文では，プロセスのデータ部を初期化することによる高速なプロセス再起動機能を提案した．プロセス再起動機能は，高速にプログラム処理を開始でき，再起動前に利用していた資源を継続して利用できる．プロセス再起動機能と既存 OS のプロセスの生成処理を比較し，プロセス再起動機能は，プロセスの削除，プロセス管理表のエントリの確保，およびメモリ空間作成の各処理を必要としないため高速であることを示

した．

Tender オペレーティングシステムにプロセス再起動機能を実装し，***Tender*** のプロセスの生成と削除処理に比べ，プロセス再起動機能が高速であることを明らかにした．さらに，BSD/OS 3.1 カーネルの make 処理について，プロセス再起動機能の効果を予測し，次のことを明らかにした．プロセスの生成に関する処理時間を比較すると，プロセス再起動機能は，***Tender*** 上でのプロセスの生成に比べ約 30 倍高速であることを明らかにした．また，プロセスの生成と削除を繰り返すベンチマークプログラムを利用した測定で，***Tender*** のプロセスの生成処理をプロセスの再起動処理に置き換えることで，プロセスの生成処理 1,000 回あたり約 2.4 秒高速化できることを明らかにした．

今後の課題としては，既存 AP の移植によるプロセス再起動機能の利用法に関する具体的な評価，動的リンクのプログラムの再起動に関する検討，およびプロセスが再起動時にプロセスが確保している外部資源の継続利用に関する検討がある．

参 考 文 献

1) Quarterman, J.S., Silberschatz, A. and Peterson, J.L.: 4.2BSD and 4.3BSD as Examples of the UNIX System, *ACM Computing Surveys*, Vol.17, No.4, pp.379–418 (1985).
2) Tevanian, Jr., A., Rashid, R.F., Golub, D.B., Black, D.L., Cooper, E. and Young, M.W.: Mach Threads and the Unix Kernel: The Battle for Control, Technical Report CMU-CS-87-149, School of Computer Science, Carnegie Mellon University (Aug. 1987).
3) 横山和俊，箱守 聰，谷口秀夫：処理形態に適したプロセスの軽量化，情報処理学会研究報告，Vol.91, No.63, pp.33–40 (1991).
4) Mukherjee, B., Eisenhauer G., and Ghosh, K.: A Machine Independent Interface for Lightweight Threads, *ACM Operating Systems Review*, Vol.28, No.1, pp.33–47 (1994).
5) Weissman, B.: Performance Counters and State Sharing Annotations: A unified Approach to Thread Locality, *ACM Operating Systems Review*, Vol.32, No.5, pp.127–138 (1998).
6) Mullender, S., Rossum, G., Tanenbaum, A., Renesse, R. and Staveren, H.: Amoeba: A Distributed Operating System for the 1990s, *IEEE Computer*, Vol.23, No.5, pp.44–53 (1990).
7) 谷口秀夫，青木義則，後藤真孝，村上大介，田端利宏：源の独立化機構による ***Tender*** オペレーティングシステム，情報処理学会論文誌，Vol.41, No.12, pp.3363–3374 (2000).
8) BSD/OS. http://www.bsdi.com/

(平成 14 年 9 月 12 日受付)
(平成 15 年 4 月 3 日採録)

田端 利宏（正会員）

平成 10 年九州大学工学部情報工学科卒業．平成 12 年同大学大学院システム情報科学研究科修士課程修了．平成 14 年同大学院システム情報科学府博士後期課程修了．平成 13 年日本学術振興会特別研究員．平成 14 年九州大学大学院システム情報科学研究院助手．博士（工学）．オペレーティングシステムに興味を持つ．電子情報通信学会会員．

谷口 秀夫（正会員）

昭和 53 年九州大学工学部電子工学科卒業．昭和 55 年同大学大学院修士課程修了．同年日本電信電話公社電気通信研究所入所．昭和 62 年同所主任研究員．昭和 63 年 NTT データ通信（株）開発本部移籍．平成 4 年同本部主幹技師．平成 5 年九州大学工学部助教授．平成 15 年岡山大学工学部教授．博士（工学）．オペレーティングシステム，実時間処理，分散処理に興味を持つ．著書「オペレーティングシステム」（昭晃堂）等．電子情報通信学会，日本ソフトウェア科学会，ACM 各会員．