情報処理学会論文誌 Vol.62 No.2 443–454 (Feb. 2021)

プロセス生成を高速化する資源プール機能の実現と評価

谷口 秀夫[1,a)] 山内 利宏[1,b)] 田村 大[1]

受付日 2020年4月30日, 採録日 2020年11月5日

概要：最近のクラウドコンピューティング環境においては，短命のプロセスを多数起動し，それらのプロセスの相互連携によってサービスを実現する手法が注目されている．このため，プロセス生成を高速化することは非常に重要である．そこで，分散指向永続オペレーティングシステム ***Tender*** では，プロセス削除時にプロセスを構成する資源を削除せずに保持し，プロセス生成処理時に再利用することでプロセスの生成処理と削除処理を高速化する手法が提案されている．しかし，この手法は，再利用する資源がない場合，プロセス生成処理を高速化できない．また，再利用する資源として保持されている量は再利用されないことにより単調増加するため，未使用メモリの枯渇を招き，新たな資源を生成できなくなる恐れがある．そこで，本論文では，これらの問題に対処する資源プール機能を提案する．資源プール機能は，プロセスを構成する資源を再利用する処理に加え，資源追加処理と資源削減処理を持つ．評価として，プロセス生成処理における各資源の再利用効果，基本評価と Web サーバの応答時間による資源プール機能の有効性，および Linux との比較を述べる．

キーワード：オペレーティングシステム，プロセス生成，プロセス構成資源，資源プール，メモリ使用量

Implementation and Evaluation of Resource Pooling Function for High-speed Process Creation

Hideo Taniguchi[1,a)] Toshihiro Yamauchi[1,b)] Dai Tamura[1]

Received: April 30, 2020, Accepted: November 5, 2020

Abstract: In the recent cloud computing environment, many short-lived processes are created, a method of realizing a service by mutual cooperation of these processes has been attracting attention. Therefore, speeding up the process creation is very important. ***Tender*** OS, thus, proposes a mechanism for fast process creation and deletion. The proposed mechanism involves the recycling of process resources. However, the proposed mechanism cannot recycle process resources during process creation if the stored process resources are not adequate. Stored process resources may increase monotonically if they are not recycled for process creation, which in turn can cause memory starvation. This paper, therefore, proposes a resource pooling function for addressing the these problems. In addition to the function for resource recycling, the resource pooling function incorporates the resource creation function and the resource reduction function. Furthermore, this paper reports the effectiveness of the resource pooling function in terms of processing time efficiency and memory usage.

Keywords: operating system, process creation, process resource, resource pool, memory usage

1. まえがき

オペレーティングシステム（以降，OS）は，プログラムを実行するためにプロセスを生成し制御する．このため，プロセス生成を高速化することは非常に重要である．

OS のプロセス生成の処理は，プロセスを管理制御するための管理表の作成，仮想空間を実現するマッピング表などの作成，および当該のプログラムについて外部記憶装置からメモリへの読み込み，に大きく分類できる．前者 2 つの処理は，プロセッサ処理でありプロセッサの高性能化に

[1] 岡山大学大学院自然科学研究科
Graduate School of Natural Science and Technology, Okayama University, Okayama, 700–8530 Japan
[a)] tani@cs.okayama-u.ac.jp
[b)] yamauchi@cs.okayama-u.ac.jp

より処理時間を短縮できるが，3つ目の処理は入力処理をともなうため時間を要する．

プロセス生成の処理開始からプログラムの最初の命令を実行するまでの時間（以降，プログラム実行開始時間と名付ける）を短くする機能として，オンデマンドページング（以降，ODP）機能やコピーオンライト（以降，CoW）機能がある．ODP機能は，プロセス生成時に前者2つの処理を行い，3つ目の処理を行わないでプログラムの最初の命令を実行しようとする．つまり，PC（プログラムカウンタ）にプログラムの最初の命令アドレスを設定する．このため，すぐにページ例外が発生し，1ページ（多くの場合4KB）の読み込みによりプログラム実行を開始でき，プログラム実行開始時間を短くできる．また，CoW機能は，fork()&exec()におけるプログラム実行開始時間を短くできる機能である．しかし，両機能とも，プログラム実行が進行するにつれページ例外が何度も発生してしまう．つまり，両機能は，プログラム実行開始時間を短くできるものの，処理オーバヘッドが大きいページ例外処理を何度も実行する．このため，プログラムが局所的に実行され，ページ例外発生回数が少ない場合に有効な機能である．多くの場合，デスクトップ計算機環境は，サーバ計算機環境に比べ，プロセッサ負荷や入出力負荷は少ないため，両機能は有効である．

一方，プロセッサ負荷や入出力負荷が高いサーバ計算機では，ページ例外の発生を抑制する必要がある．クラウドコンピューティング環境においては，短命のプロセスを多数起動し，それらプロセスの相互連携によってサービスを実現するという手法が注目されている．この手法は，次の問題をかかえている．たとえば，AmazonのAWS lambdaにおいては，プロセスを要求ごとにゼロから起動していたのでは，パフォーマンスに大きな影響が出るため，一定時間プロセスのコンテキストを保存して利用するという方式がとられている．しかし，この方式ではすべてのプロセスを再び利用するとメモリなどの資源が枯渇するため，長時間使用されていないプロセスを終了しコールドスタート状態にする工夫がある．しかし，この工夫では，ユーザからの要求に対する長い遅延が発生し，連携するlambda functionの段数によっては，数秒単位の遅延が発生するという問題（lambda cold start problem）をかかえている．

著者らは，先行研究[1], [2]で，プロセスを構成する資源を再利用する機能（以降，資源再利用機能と略す）を提案し，実現した．具体的には，プロセス生成を高速化する手法として，分散指向永続オペレーティングシステム ***Tender*** [1]（The ENduring operating system for Distributed EnviRonment）では，資源再利用機能を実現している．この資源再利用機能をプロセスの生成と削除の処理に適用することで，これらの処理を高速化できることを示し，ベンチマークプログラムにより有効性を示した[1], [2]．また，プロセス構成資源の細分化を利用して，プロセスを構成している資源を変更するプロセス変身機能を提案した[3]．さらに，プロセス変身機能の高速化とともに，BSD/OS互換システムコールI/Fを実現し，Apache Webサーバを用いた評価で有効性を示した[4]．

しかしながら，資源再利用機能には依然として課題が存在する．資源再利用機能は，同じプログラムによるプロセスの生成と終了が繰り返し発生する環境において，有効である．しかし，最初のプロセス生成処理を高速化できないという問題がある．また，様々な異なったプログラムによるプロセスの生成や終了が発生する環境では，プロセス生成時に資源の再利用の程度が少ないという問題がある．具体的には，プロセスの終了つまりプロセス削除処理において，資源再利用機能により当該資源は再利用する資源として保持されているもの（以降，再利用資源と略す）として管理される．しかし，様々な異なったプログラムによるプロセス生成が行われる環境では，再利用の条件に整合する資源が少ないため，多くの資源を再利用したプロセス生成処理は行われず，その処理を高速化できない．また，この現象により，再利用資源として管理される資源量が増加し，新たな資源生成ができないという2次的な問題も発生する．

本論文では，上記の問題の解決として，資源再利用機能を拡張した資源プール機能を提案する．資源プール機能は，資源再利用の処理に加え，資源を生成して資源プールに追加する処理（以降，資源追加処理と略す）と資源プールから資源を削除して資源量を削減する処理（以降，資源削減処理と略す）からなる．資源追加処理は，起動されるプログラムを予測できれば，プロセス構成資源を事前に生成し，資源プールに追加する処理である．資源削減処理は，資源プールに保持され続ける資源を削除する処理である．また，評価として，プロセス生成処理における各資源の再利用効果，基本評価とWebサーバの応答時間による資源プール機能の有効性，およびLinuxとの比較を述べる．

2. *Tender* オペレーティングシステム

2.1 資源の分離と独立化

Tender では，OSが制御し管理する対象を資源として細分化し，資源の分離と独立化を行っている．これにより，プロセスは複数の資源から構成される．また，プロセス構成資源は，プロセスの存在に関係なく，プロセス構成資源の生成や存在が可能である．一方，既存OSでは，プロセスに識別子を与え，プロセスの状態やプログラム，メモリ領域といったプロセス構成資源をまとめて管理している．このため，プロセス構成資源は，プロセスが存在するときのみ存在する．

資源の分離と独立化により，各資源を操作するためのプログラムを部品化できる．このため，機能の追加や削除が容易になる．また，各資源を操作するためのプログラムの

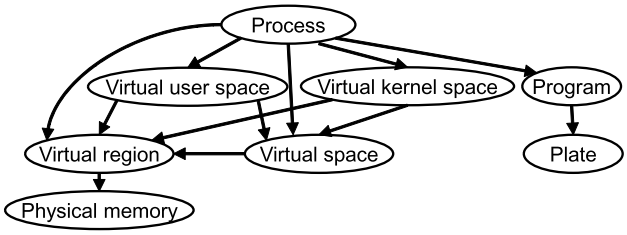

図 1 *Tender* におけるプロセスを構成する資源
Fig. 1 Process resources in *Tender*.

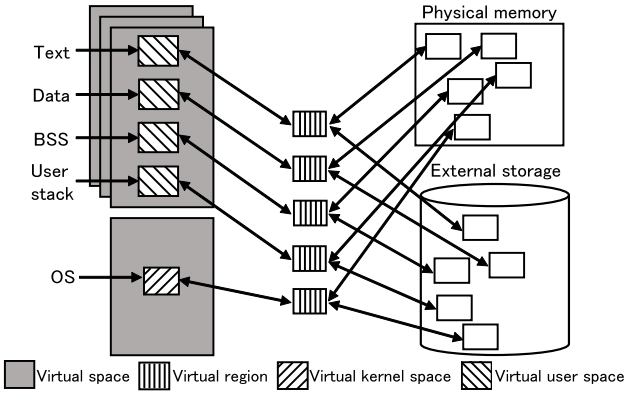

図 2 *Tender* におけるメモリ関連資源
Fig. 2 Resources of memory management in *Tender*.

呼び出しは，資源インタフェース制御と呼ばれるプログラムを介する．このプログラムが各資源を管理するプログラムの呼び出しを管理するため，OS の動作や内部状態の把握が可能になる．

2.2 プロセス構成資源

Tender におけるプロセスを構成する資源を**図 1** に示す．「プロセス」は，応用プログラム（以降，AP）の実行単位であり，プロセス識別子とプロセス管理表の情報を持つ．「プログラム」は，AP のテキスト部，データ部，BSS 部の大きさと先頭アドレス，および実行開始アドレスの情報を持つ．AP のプログラム実体などの内容は，永続的な記憶をメモリ上に提供する資源である「プレート」が保持する．

また，***Tender*** におけるメモリ関連資源の関係を**図 2** に示す．「仮想空間」は，仮想アドレスの空間であり，仮想アドレスを実アドレスに変換する変換表に相当する．「仮想領域」は，メモリイメージを仮想化した領域であり，その実体は，実メモリまたは外部記憶装置上に存在する．そのサイズはページサイズ（4 KB）の整数倍である．「仮想カーネル空間」と「仮想ユーザ空間」は，「仮想領域」を「仮想空間」の持つ仮想アドレスと対応付けたものである．「仮想カーネル空間」と「仮想ユーザ空間」の差異は，ユーザモードで走行するプログラムがアクセス可能か否かの違いである．「仮想カーネル空間」は，カーネルモードで走行するプログラムのみアクセス可能であるのに対し，「仮想ユーザ空間」は，ユーザモードで走行するプログラムもアクセス可能である．プロセスのテキスト部，データ部，BSS 部，およびユーザスタック部は，「仮想ユーザ空間」として「仮想空間」上に存在する．ここで，データ部と BSS 部はそれぞれ，初期値を持つ変数の集合と持たない変数の集合である．

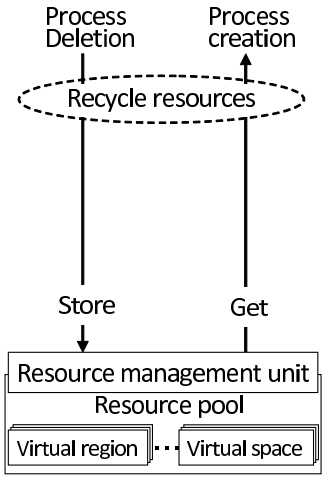

図 3 資源再利用機構
Fig. 3 Resource recycle mechanism.

2.3 資源再利用機能

2.3.1 基本方式

プロセスとして利用された資源の再利用機能を実現する機構を**図 3** に示す．AP の実行により，OS はプロセスを生成し，実行し，削除する．プロセス削除の際，プロセス構成資源を削除せずに，資源管理部に資源を保存する．これにより，プロセス生成の際，再利用の条件に整合した場合は，資源管理部が保持する資源を取得（再利用）することで，プロセス構成資源の生成処理を省略し高速化する．なお，再利用の条件に整合しない場合は，資源を再利用できないため，プロセス構成資源の生成処理が必要であり，高速化できない．なお，再利用資源の種類，および各資源の再利用の条件については，次項で述べる．

2.3.2 再利用資源

再利用資源は，再利用する資源として保持されているものであり，実行中の「プロセス」を構成する資源として利用されていない．このため，たとえば，プログラム領域に相当する部分は，「内容を利用できるテキスト部用仮想領域」として，他に同じプログラムを利用する実行中のプロセスがない場合（テキスト部共有がされていない場合）のみに，再利用資源として保存される．逆にいえば，同じプログラムを利用する複数プロセス間では共有され，その 1 つのプロセスが終了しても再利用資源に保存されない．一方，データ領域に相当する部分は，内容を再利用できないため，「仮想領域」として，プロセスが終了すると再利用資源として保存される．

表 1 再利用資源
Table 1 Recycle resources.

通番	資源の種類	再利用の条件	保持できる資源	メモリ使用量（KB）
(a)	プログラム非依存資源	つねに利用可能	ワーク領域用仮想カーネル空間	4
(b)			「仮想空間」	S_{pd}+S_{pts}
(c)		サイズが同じ場合	「仮想領域」	S_{vr}
(d)	プログラム依存資源	プログラム内容が同じ場合	「プログラム」	0
(e)			内容を利用できるテキスト部用仮想領域	S_{text}
(f)			各部の仮想領域が対応付けられた仮想空間	S_{text}+S_{data}+S_{BSS}+S_{us}+S_{pd}+S_{pts}

各再利用資源の特徴を**表 1** に示す．再利用資源は，大きく 2 つに分類できる．1 つは，プログラムの内容に依存する情報を保持しない資源（以降，プログラム非依存資源と略す）である．もう 1 つは，プログラムの内容に依存する情報を保持する資源（以降，プログラム依存資源と略す）である．

プログラム非依存資源として，以下の 3 種類がある．

(a) ワーク領域用仮想カーネル空間は，新規プロセスに渡す引数を格納するための一時的な領域であり，プロセス生成時に必要となる領域のサイズはプログラムに関係なく毎回共通であるため，つねに再利用可能である．メモリ使用量は，ページサイズ（4 KB）である．

(b)「仮想空間」は，仮想アドレスの空間であり，ページディレクトリとページテーブルの情報を持ち，つねに再利用可能である．メモリ使用量は，ページディレクトリのサイズ（$S_{pd} = 4$ KB）とページテーブルのサイズ（S_{pts} KB）の合計である．

(c)「仮想領域」は，サイズが同じであれば，再利用可能である．メモリ使用量は，「仮想領域」のサイズ（S_{vr} KB）分である．

また，プログラム依存資源は，プログラム内容が同じ場合に再利用可能であり，以下の 3 種類がある．

(d)「プログラム」は，テキスト部などの先頭アドレスとサイズ，および実行開始アドレスといった AP の情報を持つ．「プログラム」の情報の格納場所は，***Tender*** 起動時に同時に存在できる最大数の分だけ確保されており，この領域を使用するため，この資源生成時に新たなメモリ確保は行わない．

(e) 内容を利用できるテキスト部用仮想領域は，プログラムのテキスト部のデータが読み込まれた「仮想領域」である．メモリ使用量は，プログラムのテキスト部のサイズ（S_{text} KB）分である．

(f) 各部の仮想領域が対応付けられた仮想空間は，プログラムに対応するテキスト部，データ部，BSS 部，およびユーザスタック部用の「仮想領域」が対応付けられた「仮想空間」である．メモリ使用量は，プログラムのテキスト部，データ部，および BSS 部のサイズ（S_{text} KB，S_{data} KB，および S_{BSS} KB），ユーザスタック部のサイズ（$S_{us} = 64$ KB），および「仮想空間」のサイズ（$S_{pd} + S_{pts}$ KB）分である．

3. 資源プール機能

3.1 資源再利用機能の問題点と対処

資源再利用機能には，大きく 2 つの問題がある．

1 つは，この機能が有効に働く環境，つまり同じプログラムによるプロセスの生成と終了が繰り返し発生する環境においても，最初のプロセス生成処理では再利用資源がないため，プロセス生成を高速化できないという問題である．また，このことは，様々な異なったプログラムによるプロセスの生成や終了が発生する環境では，さらに深刻である．プロセスの終了つまりプロセス削除処理において，資源再利用機能により当該資源は再利用資源として管理される．しかし，様々な異なったプログラムによるプロセス生成が行われる環境では，再利用の条件に整合する資源が少ないため，多くの資源を再利用したプロセス生成処理は行われず，その処理を高速化できない．つまり，多くのプロセス生成処理において，高速化できない．

もう 1 つは，様々な異なったプログラムによるプロセスの生成や終了が発生する環境では，処理が進むにつれて再利用資源として管理される資源量が増加する．この結果，未使用メモリの枯渇を招き，新たな資源を生成できないという問題である．つまり，この環境では資源再利用機能を利用できない，むしろ利用してはならないという問題である．

したがって，資源プール機能には，今後生成されるプロセスが必要とする資源の再利用確率を向上させるために，未使用メモリの枯渇を招かない範囲で，適切な資源量を保持することが要求される．

上記の 2 つの問題に対処するため，2 つの処理を提案する．1 つは資源を生成して再利用資源を追加する処理（資源追加処理）であり，もう 1 つは再利用資源を削除して資源量を削減する処理（資源削減処理）である．なお，以降では，再利用資源として管理されたものを資源プールと呼ぶ．つまり，資源追加処理は資源を生成して資源プールに資源を追加する処理であり，資源削減処理は資源プールから資源を削除して資源量を削減する処理である．

資源追加処理により，資源プールに存在しないものの今

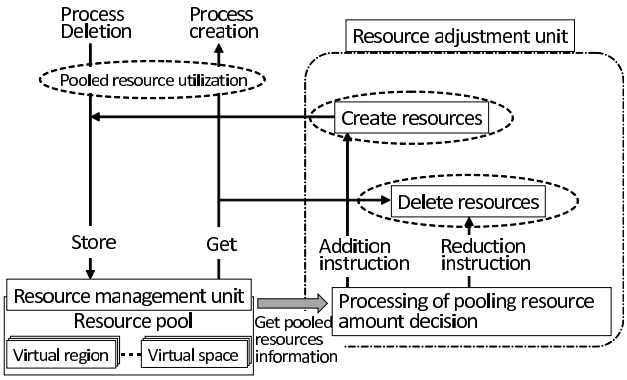

図 4 資源プール機構
Fig. 4 Resource pooling mechanism.

後プロセス生成で必要になる資源を追加することにより，同じプログラムによる最初のプロセス生成処理を高速化できる．さらに，様々な異なったプログラムによるプロセス生成処理も高速化できる．また，資源削減処理により，資源プールに保持され続ける資源を削除することで，新たな資源生成ができないという問題を解決できる．

3.2 資源プール機構

3.2.1 基本機構

資源プール機能を実現する資源プールの管理機構（以降，資源プール機構と略す）を図 4 に示す．資源プール機構は，再利用資源について，その資源量を調整する処理（以降，資源調整部と呼ぶ）を有する．

資源調整部は，以下の処理を行う．

(1) 資源管理部に格納されている資源プールの資源量を取得し，適切な資源量かどうかを判定する処理（資源量判定処理）を行う．
(2) 資源量判定処理により，資源管理部に格納されている資源プールの資源量が多いと判断された資源については，資源削減処理により再利用資源を削除する．
(3) 資源量判定処理により，資源管理部に格納されている資源プールの資源量が少ないと判断された資源については，資源追加処理により資源を生成し再利用資源として登録する．

なお，適切な資源量かどうかの判定内容については，次項で述べる．

3.2.2 資源量の調整法

〈削減法〉

適切な資源プールの資源量とするために資源を削減する量は，当該資源が再利用される可能性が高いか否かで決定する．つまり，表 1 に示した各資源の「再利用の条件」に基づき，削除する資源量を決定する．具体的には，「再利用の条件」に基づき以下のように判断し削除処理を行う．

(1) つねに利用可能：
この資源は，削除しない．これは，新たなプロセス生成処理時につねに利用できるためである．
(2) サイズが同じ場合：
この資源は，利用される確率で分類し，削除する．文献 [2] によると，サイズが 64 KB 以下の場合，プロセスを生成する際に利用されることが多く，利用される確率が高い．また，新しい FreeBSD（Ver.11）でも，/bin のプログラムの 90%以上，/sbin のプログラムの 80%以上が 64 KB 以下である．そこで，サイズが 64 KB 以下の場合，利用される確率が高いと判断し，削除しない．これに対し，サイズが 64 KB より大きい場合，プロセス削除処理で再利用資源として登録し，一定時間間隔（以降，資源量調整間隔 T1 と呼ぶ）で検査し削除する．もちろん，多くの機能を提供するプログラム（大きなプログラム）が多数存在する環境では，64 KB を大きく設定する必要がある．
(3) プログラム内容が同じ場合：
この資源は，同じプログラム内容のものを 1 個のみ再利用資源とする．つまり，プロセス削除処理で 2 個目のものは再利用資源としない．これは，2 個目以降のプロセス生成はテキスト部の共有機能が有効に働き高速化できるためである．また，一定時間間隔（以降，資源量調整間隔 T2 と呼ぶ）で検査し，再利用資源として登録された 1 個の資源についても，利用されない場合は削除する．

なお，表 1 において，「(e) 内容を利用できるテキスト部用仮想領域」は，「(c) 仮想領域」と「(d) プログラム」から構成される．また，「(f) 各部の仮想領域が対応付けられた仮想空間」は，「(b) 仮想空間」，「(c) 仮想領域」，および「(e) 内容を利用できるテキスト部用仮想領域」から構成される．このように複数の資源から構成される資源を削除する場合，資源 A は資源 A の「再利用の条件」に基づき判断し削除し，資源 A が削除されるときは，資源 A を構成する資源 B は資源 B の「再利用の条件」に基づき判断し削除する．ここで，資源量調整間隔（T1）は大きなプログラムのプロセス生成間隔に依存し，資源量調整間隔（T2）は同じプログラムのプロセス生成間隔に依存する．いずれも，サービスの性質に合わせて設定する閾値である．

上記で示した削除可否の判断と削除処理により，資源プールに保持され続ける資源を削除することで，未使用メモリの枯渇を防ぐことができ，再利用の条件に整合しない場合に必要な新たな資源生成を可能にし，新たな資源生成ができないという問題を解決する．

〈追加法〉

適切な資源プールの資源量とするために資源を追加する量は，今後当該資源が利用される可能性が高いか否かで決定する必要がある．つまり，基本的には，今後のプロセス生成の流れを予測して，追加する資源量を決定することが好ましい．また，表 1 に示した各資源の「再利用の条件」

は，当該資源が再利用される可能性が高いか否かに大きく関係している．そこで，以下の判断で追加処理を行う．

(1) つねに利用可能：
この資源は，新たなプロセス生成処理時につねに利用できるため，その数が最小保有数（H1）を下回っている場合に追加処理を行う．

(2) サイズが同じ場合：
サイズが 64 KB 以下の場合について，利用される確率が高いため，その数が最小保有数（H2）を下回っている場合に追加処理を行う．

(3) プログラム内容が同じ場合：
同じプログラムのプロセス生成が予測される場合に追加処理を行う．

ここで，最小保有数（H1）は同時走行する最大プロセス数に依存し，最小保有数（H2）は同時走行最大プロセス数やプロセス利用メモリ量に依存する．いずれも，サービスの性質に合わせて設定する閾値である．また，「プログラム内容が同じ場合」の追加処理は，たとえば，同じプログラムが何度も実行される回数の情報，あるいは特定プログラムの実行が新たなプログラム実行を引き起こすといったプログラム実行の関係情報を基に行うことができ，サービスの内容に合わせて行う処理である．

上記で示した追加可否の判断と追加処理により，再利用可能な資源を事前生成し追加することで，当該プログラムを利用する最初のプロセス生成処理時間を高速化できる．

4. 評価

4.1 観点と評価環境

基礎データとして，プロセス生成処理における各資源の再利用効果を述べる．次に，基本評価と Web サーバの応答時間の評価により，資源プール機能は，プール資源が利用するメモリ量の増加を抑制できること，またプロセス生成時間を短縮できることにより全体処理時間を短縮できることを示す．なお，資源削減処理の処理オーバヘッドについても述べる．最後に，Web サーバの応答時間を用いて，Linux と比較し，プロセス生成をともなう場合（CGI プログラム実行）とプロセス生成をともなわない場合（html ファイルの表示）の得失を述べる．

評価環境を**表 2** に示す．クライアント計算機とサーバ計算機は，100Base-TX の Ethernet で接続した．資源プール機能の有効性を明らかにするために，サーバ計算機は 1 コアを使用した．また，測定を 2 回行い，1 回目の測定でデータが OS のファイルキャッシュに保存される状態とし，2 回目の測定値を用いることでディスク I/O は発生しない場合での測定値とした．

サーバプログラムは，Apache ver.1.3.33 を使用した．また，Web サーバへの要求は，Web サーバがプロセス生成をともなう場合として，perl で作成した CGI プログラムの実行をともなう処理を Web サーバに要求する．CGI プログラム実行時の処理の流れを**図 5** に示す．CGI プログラムは，Web サーバが sh を呼び出し，sh が perl（perl ver.5.005_03）を呼び出すことで実行される．評価で使用する CGI プログラムは，ファイルから値を読み出し，その値を表示する．なお，サーバ計算機が提供するサービスとしては，Node.js や Python VM のようにスクリプト言語を JIT コンパイルする環境や言語処理系固有の VM 環境，またそれらをさらにコンテナ内に構築した環境がある．いずれも，環境を実現するプログラムをプロセスとして起動し，その環境でサービスを実行してクライアントにサービスを提供する．したがって，応答時間は，通信時間に加え，環境を実現するプロセス起動時間とサービス実行時間の和となる．このため，ODP 機能や CoW 機能を有する Linux では，サービス提供のために多くの処理を行うとページ例外が多発し，応答時間の長大化を招く．ここでは，perl で作成した CGI プログラムによる簡単な処理を用い，プログラム実行時に発生するページ例外を抑制した処理で評価する．

表 2 評価環境
Table 2 Evaluation environment.

	Client machine	Server machine
OS	FreeBSD 11.0-RELEASE	***Tender***, Linux 2.6.32
CPU	Intel Core i7-4770 (3.40 GHz) (4 cores)	Intel Core i3-2100 (3.10 GHz) (use 1 core)
RAM	4,096 MB	4,096 MB

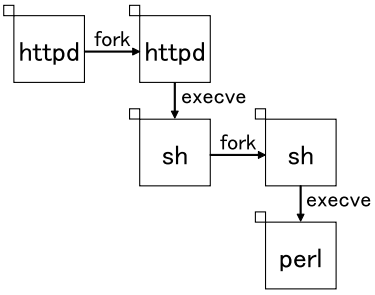

図 5 CGI プログラム実行時の処理の流れ
Fig. 5 Flow of running CGI programs.

4.2 基礎データ

評価の基礎データとして，プロセス生成処理における各資源の再利用効果を示す．

プロセス生成の処理の流れを**図 6** に示す．プロセス生成処理は，(A) から (N) に分類できる．(a) から (f) の処理は，表 1 の通番（(a) から (f)）に対応する資源の生成を行う処理である．つまり，資源プール機能を利用することにより処理の高速化を図れる処理である．(a) から (f) の処理について，資源を生成する場合（プール資源利用なし）と

表 3 各資源の確保にかかる処理時間（μs）
Table 3 Processing time of getting for pooled resources (μs).

通番	プール資源の種類	確保にかかる処理時間		プール資源利用の有無による処理時間の差分
		プール資源利用なし	プール資源利用あり	
(a)	ワーク領域用仮想カーネル空間	6.71	0.17	6.54
(b)	「仮想空間」	87.37	0.03	87.34
(c)	「仮想領域」	$0.09 \times S_{vr} + 1.28$	0.04	$0.09 \times S_{vr} + 1.24$
(d)	「プログラム」	1.31	0.02	1.29
(e)	内容を利用できるテキスト部用仮想領域	$0.12 \times S_{text} + 5.60$	0.08	$0.12 \times S_{text} + 5.52$
(f)	各部の仮想領域が対応付けられた仮想空間	$0.27 \times S_{text} + 0.25 \times S_{data} + 0.22 \times S_{BSS} + 115.92$	$0.07 \times S_{data} + 0.04 \times S_{BSS} + 4.64$	$0.27 \times S_{text} + 0.18 \times S_{data} + 0.18 \times S_{BSS} + 111.28$

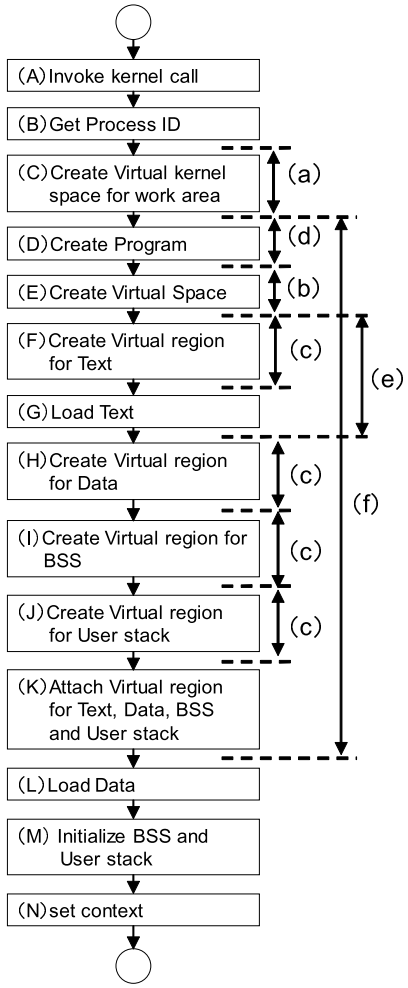

図 6 プロセス生成の処理の流れ
Fig. 6 Flow of process creation.

資源を再利用する場合（プール資源利用あり）の処理時間を実測した結果を**表 3** に示す．測定は，RDTSC 命令を利用した．

図 6 と表 3 より，表 1 に示した各資源の「資源の種類」に基づく資源プール機能の効果を**図 7** に示す．図 7 において，(1) は資源プール機能を利用しない場合の処理の流

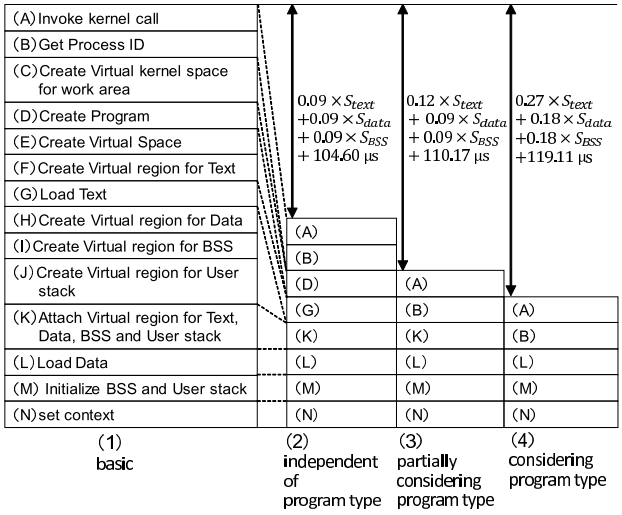

図 7 資源プール機能の効果
Fig. 7 Effect of using pooled resources.

れである．(2) は「プログラム非依存資源」を利用した場合，つまり表 1 の (a) から (c) の資源を利用した場合である．(3) は上記 (2) に加え表 1 の (a) から (e) の資源を利用した場合である．(4) は (a) から (f) の全資源を利用した場合である．当然のことながら，全資源を利用した場合 (4) は，最もプロセス生成処理を高速化できる．その際の短縮時間は，

$$0.27 \times S_{text} + 0.18 \times S_{data} + 0.18 \times S_{BSS} + 119.11\ \mu s$$

である．たとえば，Linux 2.6.32 の sh については，プログラムのテキスト部，データ部，および BSS 部の大きさが，それぞれ 836 KB，20 KB，および 20 KB である．したがって，この sh と同じサイズのプログラムのプロセス生成処理時間を 352.03 μs 短縮できる．

4.3 基本評価

プログラムサイズが異なる複数プログラムを次々に実行し終了する場合を取り上げ，資源プール機能の有効性を述

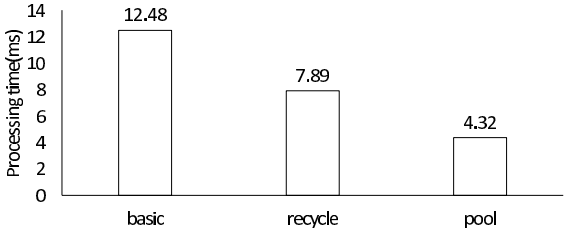

図 8 50 個のプログラム実行にかかる処理時間
Fig. 8 Processing time for 50 programs execution.

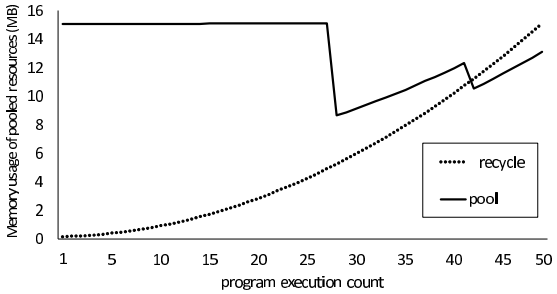

図 9 50 個のプログラム実行時のメモリ使用量
Fig. 9 Memory usage of pooled resources during 50 programs execution.

べる．

テキスト部，データ部，および BSS 部が各々 4 KB，8 KB，12 KB，…，200 KB と 4 KB ずつ増加する 50 個のプログラムを用意し，サイズが小さいものから順にプロセス生成処理を 1 回ずつ行う．なお，各プロセスは実行開始後，すぐに終了する．測定においてディスク I/O は発生しない場合であるため，プロセス生成時間は 1 ms 以下である．そこで，資源量調整間隔（T1，T2）は 2 ms と短くした．また，最小保有数（H1，H2）および「プログラム内容が同じ場合」については，50 個のプロセス生成処理のすべてにおいて，資源再利用が行われるように資源追加処理により事前に資源を作成し資源プールに追加した．

処理時間を**図 8** に示す．図 8 より，以下のことが分かる．

(1) 資源再利用をまったくしない場合（basic）に比べ，資源再利用機能（recycle）により処理時間は短縮されており，さらに資源プール機能（pool）により処理時間は大きく短縮されている．具体的には，資源プール機能（pool）により，資源再利用をまったくしない場合（basic）に比べ 8.16 ms（= 12.48 − 4.32），資源再利用機能（recycle）に比べ 3.57 ms（= 7.89 − 4.32）短縮している．つまり，65%もしくは 45%の大幅な短縮である．

なお，資源プール機能において，資源削減処理を動作させなかった場合（T1 = T2 = ∞）の処理時間は 2.84 ms となり，資源削減処理を行わないため処理時間は最も短くなった．したがって，資源削減処理のオーバヘッドは 1.48 ms（= 4.32 − 2.84）である．

また，プール資源が使用するメモリ量の変化を**図 9** に示す．図 9 より，次のことが分かる．

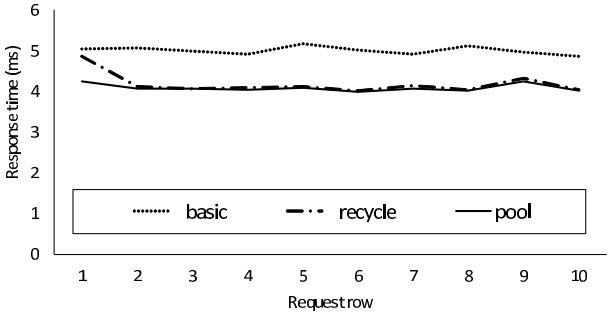

図 10 1 要求ごとの Web サーバの応答時間
Fig. 10 Response time of Web server per request.

(2) 資源プール機能（pool）は，資源追加処理により，最初からメモリ量が多い．しかし，プロセスの生成と終了を繰り返していると，資源削減処理によりメモリ量の削減が行われ，新たな資源生成ができなくなることはない．また，資源削減処理は，資源量調整間隔で資源削減を行うため，50 個のプログラム実行が終了した後でもメモリ量の削減が進む．これに対し，資源再利用機能（recycle）は，資源追加処理を有しないため，最初のメモリ量は 0 である．しかし，プロセスの生成と終了を繰り返していると，資源削減処理を有しないためメモリ量が増加しており，たとえば同様にプロセスの生成と終了を繰り返すと新たな資源生成ができなくなると推察できる．

4.4 Web サーバを用いた評価

Web サーバへ CGI プログラムの実行をともなうページを要求する場合の応答時間について述べる．

最初に，1 要求ごとの Web サーバの応答時間を実測し，「同じプログラムによるプロセスの生成と終了が繰り返し発生する環境においても，最初のプロセス生成処理では再利用資源がないため，プロセス生成を高速化できない」という問題を資源追加処理により解決できることを示す．1 要求ごとの Web サーバの応答時間を**図 10** に示す．資源プール機能（pool）では，事前に資源追加処理により，表 1 の「保持できる資源」の (f) 各部の領域が対応付けられた仮想空間である資源を作成し資源プールに追加している．また，資源量調整間隔（T1，T2）は 4.3 節基本評価と同様に 2 ms とした．図 10 より，以下のことが分かる．

(1) 資源プール機能（pool）の応答時間は，資源再利用機能（recycle）に比べ，最初の 1 要求の応答時間から約 0.63 ms（= 4.88 − 4.25）短い．これは，最初のプロセス生成処理でも資源プールから資源を利用でき，プロセス生成を高速化できるためである．

当然のことながら，2 回目の要求以降の応答時間は，資源プール機能（pool）と資源再利用機能（recycle）は資源再利用によりプロセス生成を高速化できたため同等であり，資源再利用をまったくしない場合（basic）より短い．

次に，複数の要求がある場合の Web サーバの応答時間

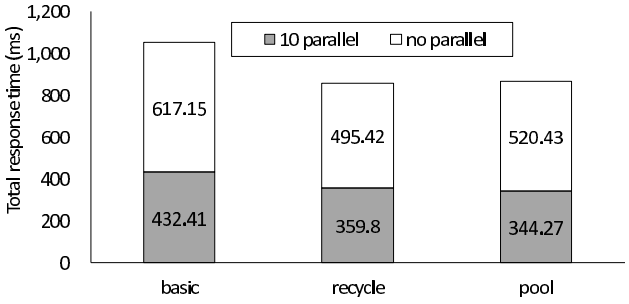

図 11 Web サーバの応答時間の和
Fig. 11 Total response time of Web server.

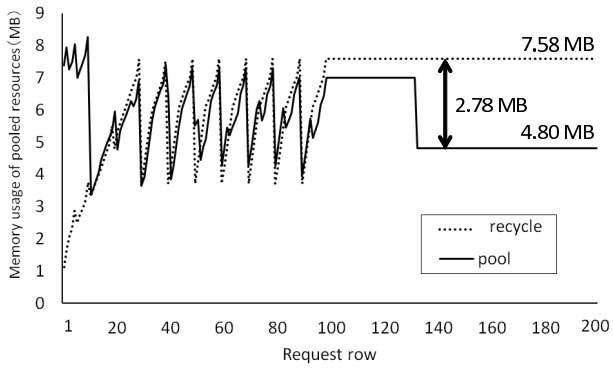

図 12 メモリ使用量の変化
Fig. 12 Memory usage of pooled resources.

およびメモリ使用量について，資源プール機能の有効性を述べる．ApacheBench による 10 並列の 10 要求（合計 100 要求）を実行後，並列要求なしで 100 要求を実行した．また，資源プール機能（pool）では，事前に資源追加処理により，表 1 の「保持できる資源」の (f) 各部の領域が対応付けられた仮想空間である資源を作成し資源プールに追加している．なお，資源削減処理オーバヘッドの影響を抑えるため，資源量調整間隔（T1，T2）は 50 ms とした．Web サーバの応答時間の和を**図 11** に示す．図より，以下のことが分かる．

(1) 資源プール機能（pool）は，資源再利用機能（recycle）に比べ，資源追加処理により最初の 10 並列の 10 要求（合計 100 要求）の応答時間は 15.53 ms（= 359.80 − 344.27）（約 4.3%）短い．しかし，次の並列要求なしの 100 要求の応答時間は，25.01 ms（= 520.43 − 495.42）（約 4.8%）長い．これは，資源削減処理によるオーバヘッドが原因である．なお，たとえばサーバ計算機の利用コア数を 1 から 2 に増やすことで，資源削減処理を別コアで行いオーバヘッドを削減できると推察する．

当然のことながら，資源プール機能や資源再利用機能の応答時間は，資源再利用をまったくしない場合（basic）に比べ約 20%短い．また，このときのメモリ使用量の変化を**図 12** に示す．図より，次のことが分かる．

(2) 最初の 10 並列の要求を処理する付近では，資源追加処理のため，資源プール機能は再利用機能に比べメモリ使用量が多い．

(3)10 並列の要求が繰り返し行われる付近では，資源プール機能と再利用機能は同様な処理を行うため，メモリ使用量の変化は同様である．このとき，メモリ使用量の増加はプロセス終了（10 個）時に発生し，メモリ使用量の減少はプロセス生成（10 個）時に発生している．

(4) 並列要求なしの要求が始まると，1 個のプロセスの生成と終了を繰り返すため，両機能ともメモリ使用量の変化は非常に少ない．その後，資源プール機能では，資源量調整間隔で資源プールの資源量を削減する処理を行うため，メモリ使用量は減少する．この結果，全要求の処理終了時のメモリ使用量は，資源再利用機能に比べ資源プール機能は 2.78 MB（= 7.58 − 4.80）（37%）削減できる．

以上により，資源プール機能は，応答時間を短縮できることを示した．ただし，資源削減処理のオーバヘッドにより，応答時間が長くなる場合もある．また，資源量調整間隔で資源削減処理を行うことにより，資源プールが利用するメモリ使用量を抑制できることを示した．今回の評価では 2.78 MB の削減である．しかし，Web サーバが行う処理は高度化し複雑化しており，多数の資源を利用する環境では多くの削減量が期待できる．また，Web サーバに限らず，多くのプログラムを使ってプロセスの生成と終了が繰り返される環境でもメモリ使用量の抑制が期待できる．

4.5 既存 OS との比較

4.4 節の複数要求の測定のように，ApacheBench による 10 並列の 10 要求（合計 100 要求）を実行したときの Web サーバの応答時間として，4.4 節で述べた CGI プログラムの実行をともなうページ（html+CGI）を要求する場合に加え，静的なページ（html）を要求する場合も測定した．また，同様な測定を Linux についても行った．

Web サーバの応答時間を**図 13** に示す．(A)，(B) いずれの場合も，***Tender*** の資源再利用をまったくしない場合（basic）は Linux より応答時間が長い．主な原因は，***Tender*** が資源の分離と独立化を実現するためのプログラム構造を持つためである．2.1 節で述べたように，各資源を操作するためのプログラムの呼び出しは，資源インタフェース制御と呼ばれるプログラムを介している．このため，プログラムの呼び出しを直接行っている Linux に比べ，OS 処理オーバヘッドは大きい．そこで，資源の分離と独立化の特徴を生かした資源プール機能の有効性を述べる．図より，以下のことが分かる．

(1) CGI プログラムの実行をともなうページ（html+CGI）を要求する場合 (A)，応答時間は，資源再利用をまったくしない場合（basic）> Linux > 資源再利用機能（recycle）> 資源プール機能（pool）の関係にある．つまり，資源プール機能は，Linux より応答時間を短縮でき，最も短い．また，前節で述べたように，資源プー

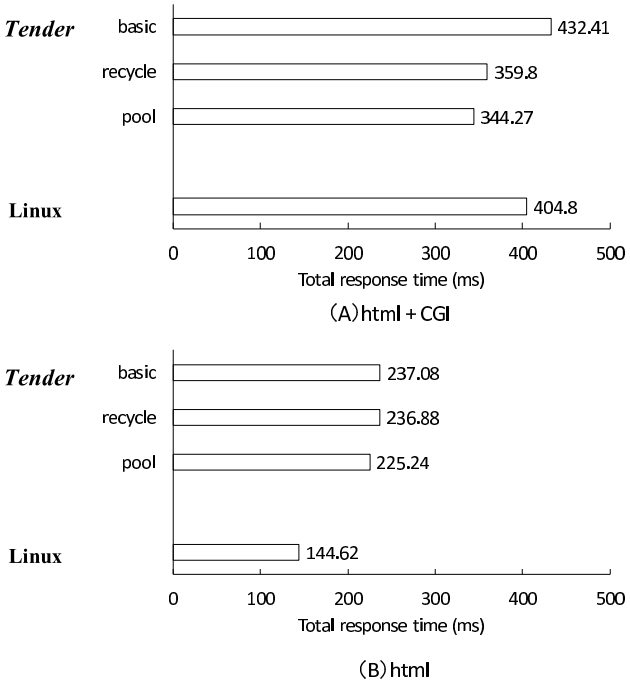

図 13 10x10 要求に対する合計応答時間
Fig. 13 Total response time for 10x10 requests.

ル機能では，資源量調整間隔で資源プールの資源量を削減する処理を行うため，資源再利用機能に比べメモリ使用量は減少する．なお，Linux では，CGI プログラムの実行により新規プロセス生成が多発する（実測したところ，ページ例外発生は 72,887 回であった）ため，応答時間が遅くなる．

(2) 静的なページ（html）を要求する場合 (B)，応答時間は，資源再利用をまったくしない場合（basic）＞資源再利用機能（recycle）＞資源プール機能（pool）＞ Linux の関係にある．つまり，資源プール機能は，資源再利用機能より短いものの Linux より応答時間を短縮できない．これは，次の要因による．この場合，Apache Web サーバは，並列要求数に応じて子プロセスを生成するものの，子プロセスは削除しない．つまり，新規プロセス生成は少ない．このため資源再利用機能や資源プール機能は有効に働かない．したがって，資源再利用をまったくしない場合，資源再利用機能，および資源プール機能の応答時間差は，CGI プログラムの実行をともなうページ（html+CGI）を要求する場合に比べ小さい．一方，Linux では，CGI プログラムの実行により新規プロセス生成が少ない（実測したところ，ページ例外発生は 473 回であった）ため，応答時間は短い．

5. 関連研究

文献 [5] では，fork システムコールにおけるプロセス生成の遅さについて言及している．vfork() や CoW 機能でプロセス生成を高速化しているものの，Chrome では，最大 100 ms の遅れが生じ，Node.js では，秒単位の遅れが exec() の前に生じることが指摘されている．また，文献 [6] では，プロセスのメモリ使用量が増大した場合に，ページテーブルのサイズが増大し，プロセス生成時の fork() システムコールや execve() システムコールの処理時間が増大することが指摘されている．

プロセス生成処理を高速化する手法として，使用するシステムコールを変更する手法がある．たとえば，UNIX の vfork システムコールや Linux の clone システムコールを使用し，親プロセスのページテーブルの複写を省略する方法がある．vfork は，発行後，exit システムコールまたは，execve システムコールを発行する前に他の関数を呼び出さないようにする必要があり，clone は，複写する範囲を引数で指定する必要がある．文献 [7] では，組み込みシステムにおけるプロセス生成処理の高速化として，親プロセスが AP を動的リンクライブラリとして事前に読み込むことで，子プロセス生成処理時に AP バイナリの配置を省略し，処理を高速化している．文献 [8], [9] では，マルチコアプロセッサ環境において，同一 AP の複数回実行による複数プロセス生成時に，単一コア上で一括で AP の内容を読み込み，プロセス生成処理を高速化している．文献 [7] の手法は，複数種類の AP を実行する場合のみ有効であり，文献 [8], [9] の手法は，同一種類の AP から複数個のプロセスを生成する場合のみ有効である．一方，提案手法は，プール資源を利用できれば，つねにプロセス生成処理を高速化できる．文献 [10] では，プロセスの状態の保存やロールバックといった投機的実行の実現に必要である機能をシステムコールとして提供している．提供されたシステムコールを bash で使用することにより，AP の実行を投機的に行い，AP の開始にかかる処理時間を短縮している．これらの使用するシステムコールを変更する手法は，AP の使用用途に最適化した利用が可能である．一方，システムコールを発行する AP のソースコードを変更する必要があり，AP のソースコードの変更を必要としない提案手法と異なる．

投機的実行を用いて，OS 内部の処理を高速化する機能の研究がある．投機的実行は，処理を事前に行うことで，連続した処理を並列に実行できる手法であり，資源が必要となる前に生成しておくことも投機的実行といえる．文献 [11], [12] では，NFS における要求やディスク同期を投機実行し，処理を高速化する．文献 [13], [14] では，分散システムと分散共有メモリシステムにおいて，受信するすべてのメッセージが正しい順序で到着したと仮定し，プロセスを投機的に実行開始させる．また，外部記憶装置上にあるデータを投機的にメモリ上に読み込んでおき，ディスク I/O 処理の時間を短縮するディスクプリフェッチ [15], [16], [17], [18] がある．これらの手法は，投機的実行の機能を OS 機能として実現するため，AP の変更を必

要としない点で，提案手法と共通する．しかし，多くは，処理時間の長大化が顕著なネットワークや外部記憶装置のI/O処理の高速化であり，必要になる前にプール資源を生成しプロセス生成処理を高速化する提案手法とは，高速化する処理の対象が異なる．また，使用するメモリ量を削減できる点も異なる．

6. むすび

プロセス生成処理を高速化する資源プール機能について述べた．資源プール機能は，プロセスを構成する資源を再利用する処理に加え，資源追加処理と資源削減処理を有する．これらの処理により，再利用できる資源の事前生成を行うことで資源再利用の確率を向上させることができ，定期的に資源プールの資源量を削減することでメモリ資源の枯渇を抑制できる．資源プール機能を実現する資源プール機構の基本機構を示し，資源プールの資源量の調整法を述べた．

評価の基礎データとして，プロセス生成処理における各資源の再利用効果を定式化した．基本評価として，資源プール機能を利用した処理の時間は，資源再利用しない場合に比べ65%，資源再利用機能に比べ45%短いことを示した．また，資源量調整間隔での資源削減処理により，再利用資源で利用するメモリ量の削減が行われ，新たな資源生成ができなくなることはないことを述べた．Webサーバの応答時間の評価では，再利用できる資源の事前生成を行うことで応答時間を短縮できることを述べた．また，資源プール機能が利用するメモリ量は資源量調整間隔での資源削減処理により減少する（37%）ことを示した．しかし，資源削減処理はオーバヘッドとなっており，資源削減処理を行わなければ，さらに20%処理時間を短縮できる．

また，Webサーバの応答時間を用いてLinuxと比較し，資源プール機能は，プロセス生成をともなう場合（CGIプログラム実行）は応答時間をLinuxより短くできるが，プロセス生成をともなわない場合（htmlファイルの表示）は短くできないことを述べた．

なお，資源プール機能は，資源の分離と独立化を実現するためのプログラム構造を持つという ***Tender*** の特徴を生かして実現されている．したがって，たとえば，Linuxでも，処理が独立して行える資源については，プロセスが利用する資源ごとに「使用中」「使用後で解放されているが内容有り」「未使用」で区別し管理できれば，同様な機能の実現が可能であり効果も期待できる．

残された課題として，追加法において，今後のプロセス生成の流れを予測する方法の検討がある．また，OSSなどを利用して，サービスの性質と資源量削減処理オーバヘッドおよび資源量調整間隔や最小保有数の閾値との関係を明確にすることがある．

謝辞 論文作成にあたり，有益なご指摘をいただいた査読者の方々に感謝します．また，本原稿の執筆にあたり，ご協力いただいた岡山大学大学院自然科学研究科の乃村能成准教授に感謝します．なお，本研究の一部は，受託研究（株式会社日立製作所）による．

参考文献

[1] 谷口秀夫，青木義則，後藤真孝，村上大介，田端利宏：資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).
[2] 田端利宏，谷口秀夫：プロセス構成資源の効率的な再利用を目指した資源管理方法の提案，情報処理学会論文誌：コンピューティングシステム，Vol.44, No.SIG10(ACS2), pp.48–61 (2003).
[3] 石井陽介，谷口秀夫：位置透過に利用可能な構成要素を用いたプロセス変身機能，情報処理学会論文誌，Vol.44, No.7, pp.1666–1679 (2003)
[4] 佐伯顕治，田端利宏，谷口秀夫：***Tender*** の資源再利用機能を利用した高速 fork & exec 処理の実現と評価，電子情報通信学会論文誌 D，Vol.J91-D, No.12, pp.2892–2903 (2008).
[5] Baumann, A., Appavoo, J., Krieger, O. and Roscoe, T.: A fork() in the road, *Proc. Workshop on Hot Topics in Operating Systems* (*HotOS'19*), pp.14–22 (2019).
[6] 松本亮介，三宅悠介，力武健次，栗林健太郎：高集積マルチテナントWebサーバの大規模証明書管理，情報処理学会研究報告，Vol.2017-IOT-37, No.1, pp.1–8 (2017).
[7] Jung, C., Woo, D., Kim, K. and Lim, S.: Performance Characterization of Prelinking and Preloading for Embedded Systems, *Proc. 7th ACM & IEEE International Conference on Embedded Software* (*EMSOFT'07*), pp.213–220 (2007).
[8] 横山和俊，谷口秀夫：マルチプロセッサシステムにおける並列処理の変化に向けた高速プロセス生成手法，電子情報通信学会論文誌 D-I，Vol.J77-D-1, No.9, pp.619–627 (1994).
[9] Kulkarni, A., Ionkov, L., Lang, M. and Lumsdaine, A.: Optimizing process creation and execution on multi-core architectures, *International Journal of High Performance Computing Applications*, Vol.27, No.2, pp.147–161 (2013).
[10] Wester, B., Chen, P.M. and Flinn, J.: Operating System Support for Application-specific Speculation, *Proc. 6th Conference on Computer Systems* (*EuroSys'11*), pp.229–242 (2011).
[11] Nightingale, E.B., Chen, P.M. and Flinn, J.: Speculative Execution in a Distributed File System, *Proc. 12th ACM Symposium on Operating Systems Principles* (*SOSP'05*), Vol.39, No.5, pp.191–205 (2005).
[12] Nightingale, E.B., Veeraraghavan, K., Chen, P.M. and Flinn, J.: Rethink the Sync, *Proc. 7th Symposium on Operating Systems Design and Implementation* (*OSDI'05*), pp.1–14 (2005).
[13] Jefferson, D., Beckman, B., Wieland, F., et al.: Distributed Simulation and the Time Warp Operating System, *Proc. 11th ACM Symposium on Operating Systems Principles* (*SOSP'87*) (1987).
[14] Ţăpuş, C., Smith, J.D. and Hickey, J.: Kernel level speculative DSM, *Proc. 3rd IEEE/ACM International Symposium on Cluster Computing and the Grid*, pp.487–494, DOI: 10.1109/CCGRID.2003.1199405 (2003).
[15] Fraser, K. and Chang, F.: Operating System I/O Speculation: How two invocations are faster than one,

USENIX 2003 Annual Technical Conference (*ATC'03*), pp.325–338 (2003).
[16] Esfahbod, B.: Preload – An Adaptive Prefetching Daemon, Master's thesis, Graduate Department of Computer Science, University of Toronto (2006).
[17] Soundararajan, G., Mihailescu, M. and Amza, C.: Context-aware Prefetching at the Storage Server, *USENIX 2008 Annual Technical Conference* (*ATC'08*), pp.377–390 (2008).
[18] Joo, Y., Ryu, J., Park, S. and Shin, K.G.: Improving Application Launch Performance on SSDs, *Journal of Computer Science and Technology*, Vol.27, No.4, pp.727–743 (2012).

谷口 秀夫 （正会員）

1978 年九州大学工学部電子工学科卒業．1980 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．1987 年同所主任研究員．1988 年 NTT データ通信株式会社開発本部移籍．1992 年同本部主幹技師．1993 年九州大学工学部助教授．2003 年岡山大学工学部教授．2010 年岡山大学工学部長．2014 年岡山大学理事・副学長．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『並列分散処理』（コロナ社）等．電子情報通信学会，ACM 各会員．本会フェロー．

山内 利宏 （正会員）

1998 年九州大学工学部情報工学科卒業．2000 年同大学大学院システム情報科学研究科修士課程修了．2002 年同大学院システム情報科学府博士後期課程修了．2001 年日本学術振興会特別研究員（DC2）．2002 年九州大学大学院システム情報科学研究院助手．2005 年岡山大学大学院自然科学研究科助教授．現在，同准教授．博士（工学）．オペレーティングシステム，コンピュータセキュリティに興味を持つ．2010 年度 JIP Outstanding Paper Award，2012 年度情報処理学会論文賞等受賞．電子情報通信学会，ACM，USENIX，IEEE 各会員．本会シニア会員．

田村 大

2016 年岡山大学工学部情報系学科卒業．2018 年同大学大学院自然科学研究科博士前期課程修了．同年三菱電機（株）電力システム製作所入所．現在，同所員．オペレーティングシステムに興味を持つ．