Vol.2010-OS-115 No.3
2010/8/3

プログラムの更新を可能とするCheckpoint/Restart 機構

室井雅仁[†1]　鵜川始陽[†2]　岩崎英哉[†2]

近年，連続的に利用されるシステムが広く普及したことに伴ない，予期せぬ実行時エラーに対処するため，耐故障性に関する研究がなされてきている．耐故障性に関する研究の一つとして，Checkpoint/Restart と呼ばれる手法がある．この手法は正常動作しているプロセスの状態を予め保存しておき，障害発生時に保存しておいた状態を復元する手法である．しかし，通常 Checkpoint/Restart は，同じプログラムを利用するプロセスへ保存した情報を復元するので，バグが原因の障害の場合，同じ障害が発生してしまう．そこで本論文では，バグを修正するなどプログラムの内容を更新したプロセスへ状態を復元することが可能な Checkpoint/Restart 機構の設計と実装を行った．提案機構を利用することで，予め保存した状態を更新後のプログラムを持つプロセスへ復元可能になり，バグの修正に本機構を利用することで，サービスを継続したままプログラムが原因の障害を防げることを確認した．

A Checkpoint/Restart Mechanism for Updateing Server Program

Masahito Muroi,[†1] Tomoharu Ugawa[†2]
and Hideya Iwasaki[†2]

Checkpoint/Restart is a well-known and useful technology for the recovery from a crash due to an unexpected runtime error. This technology first saves the snapshot of a process's running state, and if an error occurs, restores the snapshot to re-construct the state of the running process and resume the execution. However, Checkpoint/Restart usually restores a saved state to a process executing the same program used by the checkpointed process. Thus the same runtime error may occur in the future, since the bug is not fixed. To resolve this problem, this paper described the design and implementation of a Checkpoint/Restart system which enables users to restore a snapshot to a process executing a program whose bug has been fixed. We have confirmed that the new system are able to prevents the restored process from suffering from the same runtime error.

1. はじめに

インターネットを利用した Web サーバや大規模科学計算など，昼夜を問わず連続的に利用されるシステムが近年広く普及してきた．これらのシステムが提供するサービスは，中断することなく利用可能であることが求められている．そこで，プログラムの不具合が原因でシステムに予期せぬ実行時エラーによる障害が発生した時に，サービス停止を最小限に抑えるため，様々な耐故障性に関する研究がなされてきた．

耐故障性に関する研究の一つとして，Checkpoint/Restart と呼ばれる手法がある．Checkpoint/Restart は，正常動作しているプロセスの状態を予め保存しておき，障害が発生した時に保存しておいた状態を復元して，プロセスを正常な状態に戻すという手法であり，現在まで多くの研究[1)–3)] が行われてきた．

通常 Checkpoint/Restart は，保存したプロセスの状態を復元する時，同じプログラムを利用しているプロセスへ状態を復元する．そのため，プログラムのバグが原因で障害が発生した場合，プロセスが持つバグは修正されていないので，再度同じ障害が発生する可能性がある．

そこで本論文では，サービスを停止することなくプログラムのバグ修正等の更新を可能にするため，プログラムの内容を更新したプロセスへ状態を復元することが可能な Checkpoint/Restart 機構を実現する．本機構を用いることで，予め保存した状態を更新後のプログラムを持つプロセスへ復元可能になるため，バグの修正に本機構を利用することで，サービスを継続したまま，プログラムが原因の障害が再度発生することを防ぐことができる．

プログラムが更新されたプロセスへ実行状態の復元を可能にするため，以下に示す方針で提案機構を設計した．プログラムの更新は通常，プログラムが行うひとまとまりの処理内容を単位として行われる．このような変更を施した場合，更新後のプログラムにおいて，保存した状態に対応する状態が存在しない可能性がある．そのため，更新後のプログラムで対応する状態が存在するかどうか，保存した実行情報から正確に判断できるようなタイミングで Checkpoint を行う．このタイミングでの Checkpoint を，以下「実行状態に応じたタ

†1 電気通信大学大学院情報工学専攻
Departments of Computer Sience, The University of Electro-Communications
†2 電気通信大学大学院情報理工学研究科
Graduate School of Informatics and Engineering, The University of Electro-Communications

イミング」と呼ぶ．また，提案機構は既存のプログラムにも障壁なく利用可能にするため，Checkpoint の対象となるプログラムを変更せずに利用できるようにする．

本論文の構成は次の通りである．2 節で本研究の関連研究を挙げ，3 節では提案機構の利用方法を述べる．4 節は提案機構の概要を説明し，5 節で提案機構の実装について述べる．6 節で提案機構の評価を行い，7 節で本論文をまとめる．

2. 関 連 研 究

2.1 Checkpoint/Restart

Checkpoint/Restart に関する研究は，プロセスの状態を保存する Checkpoint のタイミングにより，定期的なタイミングとプログラムの実行状態に応じたタイミングの 2 種類に分類できる．

はじめに定期的なタイミングについて考える．このタイミングは，プロセスの処理内容に関わらず，一定間隔毎にプロセスの状態を保存する．既存研究として，Libckpt[1]，TICK[2] がある．Libckpt は，ユーザプログラムのライブラリとして提供され，その機能を利用するには，対象となるプログラムを変更する必要がある．一方，TICK はカーネルスレッドとして実現されており，対象のプログラムを変更する必要はない．

しかし，これらの機構は定期的に状態の保存を行うため，何らかのひとまとまりの一連の動作の最中で Checkpoint が行われてしまう可能性があり，実行状態に応じたタイミングでの Checkpoint になっていないという問題点がある．そのため障害発生時に，復元すべき状態が実行情報として保存されていない可能性があり，さらに更新後のプログラムへ実行状態を復元する時に，対応する実行状態の有無の判断が難しい．したがって，これらの機構を本研究の目的に利用することはできない．

次に実行状態に応じたタイミングについて考える．このタイミングは，Checkpoint を行うのに適したあらかじめ指定された動作をプロセスが行った時に，プロセスの状態を保存する．そのため，定期的なタイミングと異なり，実行状態の復元時に必要となる状態が必ず保存するように指定することができる．その実現方法のひとつとして，システムコールを利用した方法があり，具体的には BProc[4] が採用している VMADump[5] が挙げられる．

VMADump はあるファイルディスクリプタへの `read`，`write` システムコールを利用して，プロセスの状態の保存と復元を行う．そこで，プログラム内の Checkpoint すべきタイミングにおいて，システムコールを呼び出すことで，実行状態に応じたタイミングの Checkpoint が可能である．しかし，VMADump はプログラム中にシステムコールを明示的に記述する必要があるので，プログラムを変更する必要がある．

Checkpoint/Restart は，耐故障性以外の用途にも利用されている．Zap[6] は Checkpoint/Restart を利用したプロセス移送の研究である．Zap は，対象プロセスを pod と呼ばれる仮想化空間で実行し，プロセス移送前後で利用する計算機資源の一貫性を保つ．

これらの研究は，状態の保存方法や目的は異なるが，状態の復元先は同じプログラムから生成されたプロセスとなる．対して本研究は，保存したプロセスの状態を変更し，プログラムの内容を更新したプロセスへ状態を復元する．

2.2 動的ソフトウェア更新

本研究は Checkpoint/Restart を利用して，プログラムが提供するサービスを終了せずに，プログラムの内容を更新する．これに関連して，動作中のプロセスへプログラムの更新内容を挿入して，実行を終了せずに更新する研究がある．

Ginseng[7]，POLUS[8] は，それぞれ C 言語を対象とした研究である．Ginseng は，コンパイル時にプログラムの内容を動的に更新可能な形に変更する．同様に POLUS も，コンパイル時に動的な更新を可能にするライブラリをプログラム内に挿入することで，実行中の動的更新を可能にする．これらの機構を利用するには，プログラムの再コンパイルが必要となり，既に実行中のプロセスを更新することはできない．

OPUS[9] は，プログラムの内容を変更する必要はなく，既に実行中のプロセスを動的に更新する．しかし，OPUS はプロセスが正常動作している事を前提としており，障害を起こす状態のプロセスに対処することができない．

3. 利 用 方 法

提案機構の利用方法の概要を**図 1** に示す．

初めに，状態の保存方法について説明する．対象プロセスの実行状態を保存するために，ユーザは以下のものを用意する．

(1) 実行形式である保存対象のプログラム
(2) 実行形式である保存対象のプログラムと状態の保存場所へのパス，状態の保存を行うタイミングとなる関数のリスト，対象プロセスのプロセス ID を記述した設定ファイル

ユーザは用意した設定ファイルを提案機構へ読み込ませて，状態を保存するプロセスに関する情報をプログラムとして自動生成する．そして，生成されたプログラムをコンパイルし提案機構に渡す．提案機構は，対象プロセスに関する情報を利用してプロセスの動作を監視

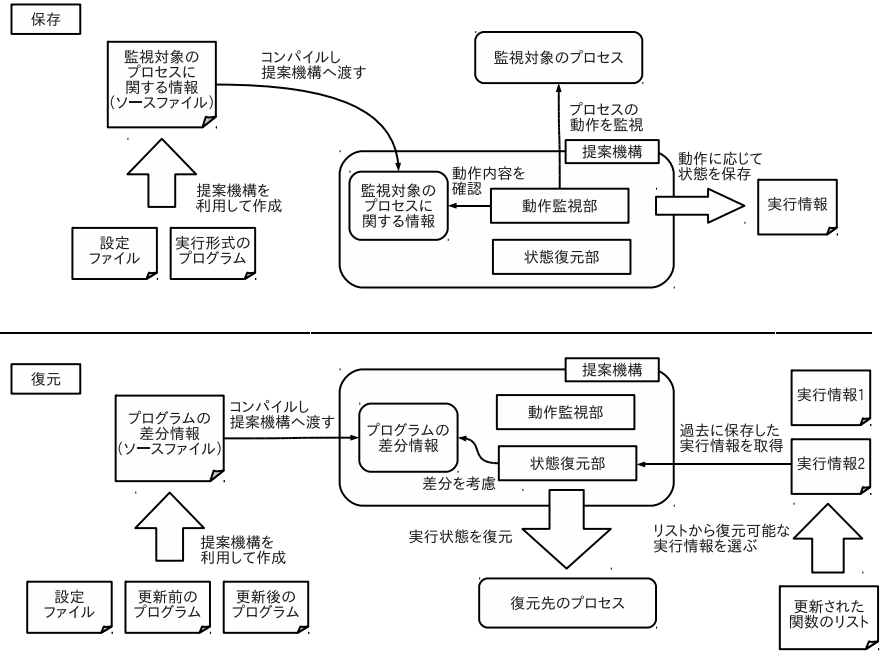

図 1 提案機構の動作概要
Fig. 1 Overwiew of the system

し，監視対象のプロセスが設定ファイルに記述された動作をした時に，実行状態を保存する．

次に復元の手順を説明する．更新後のプロセスへ状態を復元する時，ユーザは以下のものを用意する．

(1) 更新前後のプログラム（実行形式）
(2) 更新された関数のリストを記述したファイル
(3) 更新前後の実行形式であるプログラムと復元する実行状態へのパス，復元先プロセスのプロセス ID を記述した設定ファイル

ユーザは提案機構を利用して，更新された関数のリストから復元できる実行情報を選び出す．そして，設定ファイルを提案機構に読み込ませて，更新前後の差分をプログラムとして

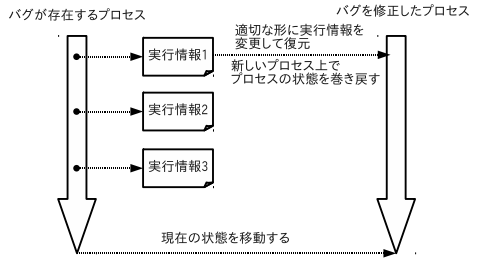

図 2 実行状態の復元
Fig. 2 Restoring a process state into an updated process

自動生成する．生成されたソースコードをコンパイルして提案機構へ渡すと，渡された差分情報を考慮しつつ，復元先のプロセスへ状態を復元する．

4. 提 案 機 構

本研究では，プログラムの内容を更新したプロセスへ，状態の復元が可能な Checkpoint/Restart 機構を提案する．以下簡単のため，状態を保存したプロセスを「更新のないプロセス」，内容が更新されたプログラムを持つプロセスを「更新のあるプロセス」と記述する．

図 1 に示したように，提案機構は実行中のプロセスから，その実行状態を取り出し実行情報として保存する．保存する実行状態はプログラムの更新を反映可能にするために，プロセスが持つメモリ上のデータ等だけでなく，現在プロセスが呼び出している関数の呼び出し経緯等の，実行時スタックにある情報も含む．ここで，関数の呼び出し経緯とは，例えば関数 `main` が関数 `f` を呼び出し，さらに `f` が関数 `g` を呼び出している場合の，`main - f - g` といった関数の呼び出し順をさす．

復元時には，プログラムの更新前後での差分を実行情報へ反映させた後，更新のあるプロセスへ復元する．実行状態の保存時には，将来プログラムに加えられる更新内容はわからない．そのため更新内容の反映は，更新のあるプロセスへ状態の復元時に行う．

提案機構を利用することで**図 2** に示す様に，過去に保存した状態とプロセスの現在の実行状態を，更新のあるプロセスへ復元できる．

4.1 プログラムの更新

4.1.1 対象とする更新内容

提案機構は，プログラムコードに対する更新を対象とし，プロセスが実行中に利用する仮想メモリ上のデータ構造，例えば構造体のメンバ変数の変更などは，更新の対象外とする．具体的には，関数の追加と削除，関数内部の書き換えを対象とする．

データ構造に対する更新を対象外としたのは，対象プログラムへの特殊な変更なしに，提案機構を利用可能な設計としたことによる．プログラムが実行中に作成するデータ構造について，プロセスの外部からメモリ上に配置されるデータの特定は難しく，データ構造の更新は困難であるためである．

4.1.2 更新可能な実行状態

提案機構を利用して，更新のあるプロセスへ復元できる実行状態には，更新のあるプロセスにおいて対応する実行状態が存在しなければ，保存した実行情報を復元することはできないという制限がある．

例えば，関数の呼び出し経緯が，`main - f - g - h` という実行情報を復元する場合を考える．復元先となる更新のあるプロセスにおいて，関数 `h` が削除されていたとすると，対応する状態が更新のあるプロセスに存在しないため，状態を復元することができない．また同様に，同名の関数 `h` が存在しても，その処理内容が異なっている場合，更新のあるプロセスで同じ処理を行っている実行状態が存在しない．

以上のことから提案機構は，更新の前後で変更のない関数のみ実行している実行情報を，更新のあるプロセスへ復元することができる．

4.2 Checkpoint のタイミング

提案機構では，プログラム内の関数の実行を Checkpoint の単位として指定可能にする．一般的なプログラムでは，何かしらのまとまった処理内容を，関数として記述する．そのため，ある関数の実行開始または終了の瞬間を Checkpoint のタイミングとすることで，実行状態に応じた Checkpoint が可能になる．

Checkpoint のタイミングは，プログラムを利用する状況に応じて異なる．そこで提案機構では，プログラムの実行開始後でも，Checkpoint のタイミングを変更できるようにする．実行開始後に変更できるようにすることで，Web サーバのように時間帯によって動作頻度が変化するプログラムにも，柔軟に対応できるようになる．

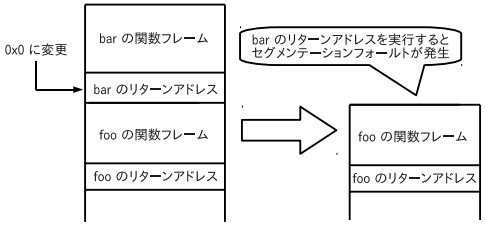

図 3 Stack Inspection の概要
Fig. 3 Stack Inspection

5. 実　　装

提案機構はプログラムの種類に関わらず動作可能にするため，OS 内で動作するように実装した．具体的には，Linux 上に，Linux が提供する Loadable Kernel Module を用いて実装した．

5.1 実行状態の把握

関数単位で Checkpoint のタイミングを指定可能にするため，提案機構では，Stack Inspection を応用して，関数単位でのプログラムの動作を OS に通知できるようにした．Stack Inspection とは，特定の関数の実行が終了したことを OS へ通知する手法であり， OPUS で提案された．関数 `foo` が関数 `bar` を呼び出した場合の動作を，**図 3** を利用して説明する．

プログラム内で `foo` が `bar` を呼び出すと，`bar` の関数フレームがスタックに積まれる．この時，関数フレームの一部として，呼び出し元の `foo` へのリターンアドレスも，関数スタックに積まれる．Stack Inspection では，適当なタイミングで関数スタックを走査して，`bar` の関数フレームを探し出し，そのリターンアドレスを 0 番地へ変更する．`bar` の実行が終了すると，プロセスはこのリターンアドレスを取り出し，0 番地に制御を移そうとする．この結果，セグメンテーションフォールトが発生し，OS へ処理を移すことができる．この様にして Stack Inspection は，特定の関数の実行終了を OS へ通知する．提案機構はこの様にして，発生させたシグナルを捕捉して Checkpoint を行う．

提案機構は呼び出しから戻る時を捕捉すべき関数をリストとして保持する．そのため，特定すべき関数のリストを変更することで，プロセスの実行中でも Checkpoint 対象となる関数の種類を変更できる．また，対象プロセスがシステムコールを呼び出した時，Stack Inspection のために関数フレームを確認して，戻り番地を 0 に変更する．

上述の方法による Checkpoint は，関数を呼び出すタイミングでなく，関数呼び出しから戻るタイミングでだけ行われる．提案機構が目指す Checkpoint のタイミングは，プロセスの実行状態に応じたタイミングなので，これで十分である．なぜなら，ある関数が終了したタイミングは，プログラム内で何かしらのまとまった処理が終了した時であるためである．

5.2 実行情報について

Checkpoint における実行情報の保存場所には，メモリ内に保持する方法や，ファイルへ書き出す等いくつかの方法がある．提案機構は耐故障性を目的としているので，メモリを利用した読み出しの高速化等を行う必要はない．そのため，実行情報はファイルとして保存する．

5.2.1 保存する内容

保存すべき実行情報は，プロセスの状態を復元するのに必要な情報である．そのため提案機構は，対象となるプロセスが持つ仮想メモリ上のデータだけでなく，カーネル内に存在するプロセスに関する情報もいくつか保存する．また，プログラムの差分を考慮するために，関数の呼び出し経緯やメモリ上のリターンアドレスの位置，およびレジスタの状態も同時に保存する．

初めにプロセスの仮想メモリについて説明する．仮想メモリ内のデータは，大きく分けて実行テキスト，ヒープ，スタックの 3 種類に分類できる．ヒープ，スタック内の情報は，同じプログラムから生成されたプロセスでも，プロセスの状態に応じて変化する．対して実行テキストは，プロセスの状態が異なっていても変化しない．また，更新のあるプロセスへ状態を復元する場合，復元先の実行テキストは保存時と異なったものになる．以上のことから，提案機構はプロセスの仮想メモリ内にある，ヒープとスタックを実行情報として保存する．メモリ上のデータを保存する際には，データの内容だけでなく，そのデータ領域に設定された読み書きのフラグ情報も保存する．

次にカーネル内の情報について説明する．カーネル内にはプロセスに関する様々な情報が存在する．提案機構はその中で，プロセスの状態に深く関係する，ファイルディスクリプタ，プロセスが登録したシグナルハンドラ，及び TCP ソケットの情報を保存する．ファイルディスクリプタについては利用中のディスクリプタ番号以外に，オープン中のファイルへのパス，オープン時のフラグ，ファイルオフセットも保存する．

5.2.2 実行状態の確認

保存した状態と同等の状態が更新のあるプロセスにも存在すれば，保存した状態を更新のあるプロセスへ復元できる．

ある状態において特定の関数が実行中かどうかは，プロセスが持つスタックを利用して確認できる．提案機構は，変更した関数の関数フレームがスタック上に存在しない時，図 2 に示す方法で，更新のないプロセスの状態を更新のあるプロセスへ復元できる．

予め保存した実行情報を復元する場合，実行情報の内容を確認し，変更した関数が実行されていないものが復元できる．また，実行中のプロセスの状態を移動させる場合，変更した関数の関数フレームがスタックから降ろされた時に，プロセスの状態を移動させる．スタック上から関数フレームがなくなる瞬間は，Checkpoint のタイミングを取得するのと同様に，Stack Inspection を用いて特定する．

5.2.3 実行情報の変更

更新のないプロセスと更新のあるプロセスが同じ状態の時，2 つのプロセスの差異は，実行テキストのみとなる．そこで，実行情報内に存在する実行テキストに関する情報を，更新のあるプロセスに適する形へ変更し，更新のあるプロセスに復元する．実行情報内の実行テキストに関する情報とは，スタック上のリターンアドレスと，プロセスが設定したシグナルハンドラ，レジスタ内のプログラムカウンタである．

プログラムの各関数が仮想メモリ上にマップされる位置は，更新のないプロセスと更新のあるプロセスで異なる．しかし関数に変更がない場合，各関数の先頭アドレスから関数内の命令までの距離は，マップされる位置に関わらず一定になる．そこで提案機構は，関数がマップされる位置のずれを，スタック上のリターンアドレスとプロセスが設定したシグナルハンドラに加えることで，更新のあるプロセスの実行テキストに適した形へ変更する．

5.3 復元の手順

提案機構は，実行中のプロセスへ実行状態を復元するため，復元先のプロセスが現在持つ実行状態は破棄される．また，復元する間の実行状態の一貫性を保つために，復元先のプロセスの実行を一旦中断して実行状態を復元する．その手順を次に示す．

(1) 復元先のプロセスを実行する．
(2) 復元先となるプロセスのプログラムカウンタの値を 0 番地に設定し，セグメンテーションフォールトを発生させ，そのシグナルの実行を提案機構が捕捉する．
(3) はじめにプロセスの仮想メモリ上のデータを復元する．
  (a) 復元先のプロセスが持つメモリ上のデータを，実行テキストを除いて全て削除する．
  (b) ファイルに保存してあるメモリ上のデータを，保存時と同じアドレスへ再マップする．静的変数等のデータセグメントは，実行テキストの変化に合わせて，

再マップする位置を変更する．再マップの際，各データ領域の読み書きのフラグも，状態の保存時と同じ状態に設定する．

(c) スタック上にあるリターンアドレスを，更新のあるプロセスに合わせて変更する．

(4) 復元する実行状態でオープン中のファイルを再度オープンする．ファイルオフセットやファイルディスクリプタも，保存時と同じ値に設定する．

(5) シグナルハンドラを，更新のあるプロセスへ合わせて再度設定する．

(6) レジスタの値を復元する．

(7) ネットワークソケットが存在する場合，ソケットの情報を復元する．提案機構は，更新のないプロセスと更新のあるプロセスが同一計算機内に存在するとき，接続中のソケットを復元できる．

6. 評　　価

実験環境として，CPU が Intel Core 2 Quad 6600（2.40 GHz，4 コア），メモリが 4 GB のマシンを利用した．また，Linux kernel 2.6.26.7 向けの Linux Kernel Module として，提案機構を実装した．

6.1 プログラムの更新

提案機構の有効性を検証するために，脆弱性をもつプロセスの実行状態を，脆弱性を修正したプロセスへ復元して，脆弱性を取り除く実験を行った．提案機構を利用することで，プロセスが提供するサービスを中断することなく，更新のあるプロセスへ実行状態が復元できることを確認した．

ディレクトリトラバーサル脆弱性をもつ Web サーバを自分で作成して，検証を行った．ディレクトリトラバーサル脆弱性とは，クライアントが要求したファイルパスの確認が不十分で，Web サーバが意図しないファイルをクライアントへ送信してしまう脆弱性である．検証に利用した Web サーバでは，クライアントから送られてくるリクエスト内の"../"という文字列を除外しないため，計算機内にある任意のファイルをオープンできてしまう．

クライアントから Keep-Alive を指定し，"../"を利用して/etc/passwd へのパスをリクエストとして，脆弱性のある Web サーバへ送信すると，Web サーバは/etc/passwd の内容をクライアントへ送信した．Keep-Alive を指定したので，クライアントとの接続を保持したまま，次のリクエストが送信される前に，Web サーバの実行状態を脆弱性を取り除いたプロセスに復元した．そして，クライアントから同じリクエストを受信すると，Web サーバはリクエストを正常に解釈し，/etc/passwd の内容を送信しないことを確認した．この検証より，提案機構を利用して更新のないプロセスの実行状態を，更新のあるプロセスへ復元できることが実証された．

表 1 システムコールの処理にかかる時間（μsec）
Table 1 Overhead of systemcalls (μsec)

提案機構なし	提案機構あり					
	監視対象外	$n=1$	2	3	4	5
2.28	2.29	3.66	3.88	4.12	4.46	4.87

表 2 状態の保存と復元に必要な時間（sec）
Table 2 Overhead of Checkpoint and Restart (sec)

メモリサイズ	Checkpoint	Restart
100MB	1.36	1.50
200MB	3.11	3.19
300MB	4.81	4.45
400MB	6.44	5.46
500MB	8.15	6.90

6.2 オーバーヘッド

提案機構を利用して実行状態の保存と復元を行った場合，システムコールの時と，実行状態の保存と復元時にオーバーヘッドが発生する．このオーバーヘッドを測定した．

はじめに，システムコール捕捉処理のオーバーヘッドとして，open したファイルをすぐに close する処理にかかる時間を計測した．結果を**表 1** に示す．表中の n はプロセスのスタック上にある，関数フレームの数を表す．

提案機構を利用した場合，Checkpoint の対象となるプロセスかどうかの判断のため，全てのプロセスが発行するシステムコールを捕捉する．そのため，提案機構による動作の監視対象外のプロセスが発行したシステムコールにも，オーバーヘッドが発生してしまう．しかし，測定結果より監視対象外の場合，提案機構による影響は誤差の範囲である．提案機構によって Stack Inspection の対象となったプロセスの場合，関数フレームが 5 つスタック上に存在した時で，システムコール全体の処理時間が約 2 倍となった．

次に実行状態の保存と復元に必要な時間を，プロセスが利用するメモリサイズを変化させて測定した．測定結果を**表 2** に示す．

プロセスが持つメモリサイズに応じて，状態の保存と復元ともに処理時間が増加した．それぞれの処理にかかる時間の主な原因は，プロセスのメモリの情報読み出しファイルへの

書き出し，またはファイルの内容を読み込みメモリへ再マップする処理となる．そのため，実際の実行速度は提案機構を利用する計算機環境に依存するが，プロセスが利用するメモリサイズに比例して，保存と復元に必要な時間も変化する．

7. ま と め

プログラムの内容を更新したプロセスへ状態を復元することが可能な Checkpoint/Restart 機構の設計と実装を行った．提案機構を利用することで，サービスを停止することなく，プログラムのバグ修正等の更新が可能であることを確認した．

今後の課題として，計算機をまたいだ接続中のネットワークソケットの復元や，動的ライブラリへの対応が必要であると考えられる．

参 考 文 献

1) Plank, J.S., Beck, M., Kingsley, G. and Li, K.: Libckpt: transparent checkpointing under Unix, *TCON'95: Proceedings of the USENIX 1995 Technical Conference Proceedings on USENIX 1995 Technical Conference Proceedings*, Berkeley, CA, USA, USENIX Association, pp.18–18 (1995).
2) Gioiosa, R., Sancho, J. C., Jiang, S. and Petrini, F.: Transparent, Incremental Checkpointing at Kernel Level: a Foundation for Fault Tolerance for Parallel Computers, *SC '05: Proceedings of the 2005 ACM/IEEE conference on Supercomputing*, Washington, DC, USA, IEEE Computer Society, p.9 (2005).
3) Sankaran, S., Squyres, J.M., Barrett, B. and Lumsdaine, A.: The LAM/MPI checkpoint/restart framework: System-initiated checkpointing, *in Proceedings, LACSI Symposium, Sante Fe*, pp.479–493 (2003).
4) Hendriks, E.: BProc: the Beowulf distributed process space, *ICS '02: Proceedings of the 16th international conference on Supercomputing*, New York, NY, USA, ACM, pp.129–136 (2002).
5) Hendriks, E.: VMADump, http://bproc.sourceforge.net (2002).
6) Osman, S., Subhraveti, D., Su, G. and Nieh, J.: The design and implementation of Zap: a system for migrating computing environments, *OSDI '02: Proceedings of the 5th symposium on Operating systems design and implementation*, New York, NY, USA, ACM, pp.361–376 (2002).
7) Neamtiu, I., Hicks, M., Stoyle, G. and Oriol, M.: Practical dynamic software updating for C, *PLDI '06: Proceedings of the 2006 ACM SIGPLAN conference on Programming language design and implementation*, New York, NY, USA, ACM, pp.72–83 (2006).
8) Chen, H., Yu, J., Chen, R., Zang, B. and Yew, P.-C.: POLUS: A POwerful Live Updating System, *ICSE '07: Proceedings of the 29th international conference on Software Engineering*, Washington, DC, USA, IEEE Computer Society, pp.271–281 (2007).
9) Altekar, G., Bagrak, I., Burstein, P. and Schultz, A.: OPUS: online patches and updates for security, *SSYM'05: Proceedings of the 14th conference on USENIX Security Symposium*, Berkeley, CA, USA, USENIX Association, pp.19–19 (2005).