(19)日本国特許庁(JP)　(12)公開特許公報(A)　(11)特許出願公開番号
特開2009-157684
(P2009-157684A)
(43)公開日　平成21年7月16日(2009.7.16)

(51)Int.Cl.		F I			テーマコード（参考）
G06F 9/48	(2006.01)	G06F	9/46	455B	5B048
G06F 9/46	(2006.01)	G06F	9/46	350	
G06F 11/26	(2006.01)	G06F	11/26		

審査請求 未請求　請求項の数 12　ＯＬ　（全 14 頁）

(21)出願番号　特願2007-335724 (P2007-335724)
(22)出願日　平成19年12月27日 (2007.12.27)

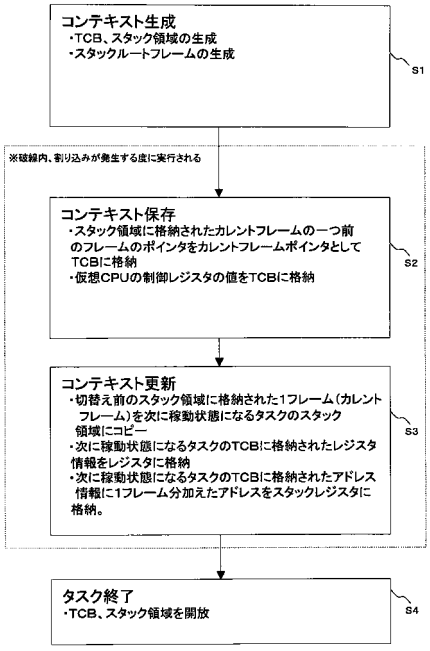
（特許庁注：以下のものは登録商標）
１．フロッピー

(71)出願人　301063496
東芝ソリューション株式会社
東京都港区芝浦一丁目１番１号
(71)出願人　501486327
株式会社インターデザイン・テクノロジー
東京都港区芝三丁目４３番１６号
(74)代理人　100101111
弁理士　▲橋▼場　満枝
(72)発明者　石井　正悟
東京都港区芝浦一丁目１番１号　東芝ソリューション株式会社内
(72)発明者　由良　浩司
東京都港区芝三丁目４３番１６号　株式会社インターデザイン・テクノロジー内

Ｆターム(参考) 5B048 DD14

(54)【発明の名称】仮想化プログラム、シミュレーション装置、仮想化方法

(57)【要約】　（修正有）
【課題】マルチタスク処理を行うことが可能なＯＳに対し、複数のタスクを切替えて高速にシミュレーション実行することができる仮想化プログラムを提供する。
【解決手段】ネイティブ・コード・シミュレータにおいて、マルチタスクＯＳが管理するタスク毎にタスク固有なスタックを持てるようにする仮想化プログラムであって、ターゲットＣＰＵで特殊な制御レジスタ操作によって行っていたコンテキスト生成・退避・復元・消去をネイティブ・コード・シミュレータが提供するＡＰＩで行うものである。マルチタスクＯＳのポーティングでは、そのＡＰＩを呼び出すようにソースコードを変更する。そのＡＰＩにて、タスク毎に固有なスタックを割り当てて、タスク切替え時にはスタック切替えを行い、コンテキスト・スイッチングを可能にする。
【選択図】図２



【特許請求の範囲】
【請求項１】
マルチタスク処理を行うことができるＯＳのネイティブ・コード・シミュレーションにおいて、シミュレーション対象であるターゲットＯＳ上でタスクのそれぞれが起動されるごとに、前記タスクのそれぞれが呼び出す関数順に関数呼び出し情報をＬＩＦＯ構造でタスクごとに格納するスタック領域と、前記タスクのそれぞれが最後に実行中断したときのレジスタ内容を最新レジスタ情報としてタスクごとに格納するレジスタ保存領域と、前記タスクそれぞれの状態を示す状態情報をタスクごとに格納するタスク状態格納領域とを生成し、
所定のタイミングで、稼動しているタスクが最後に呼んだ関数の少なくとも一つ前に関数を呼び出した時点のレジスタ内容を前記最新レジスタ情報として前記レジスタ保存領域に格納するとともに、前記稼動しているタスクの状態を示す状態情報を前記タスク状態格納領域に格納し、
格納されている次に稼動すべき待機中のタスクの最新レジスタ情報および状態情報に基づき、前記稼動しているタスクから前記待機中のタスクへとタスクの稼動を切替える処理をコンピュータに実行させる仮想化プログラム。
【請求項２】
請求項１に記載の仮想化プログラムにおいて、
前記切替えは、前記待機中のタスクの最新レジスタ情報に基づいたレジスタ内容を、ホストＣＰＵのスタックレジスタに格納することで、ネイティブ・コード・シミュレーションにおけるターゲットＯＳ上のタスクの稼動を切替えることを特徴とする仮想化プログラム。
【請求項３】
請求項１または請求項２に記載の仮想化プログラムにおいて、
前記レジスタ保存領域への格納は、ホストＣＰＵの所定のレジスタに格納された値を、前記稼動しているタスクの最新レジスタ情報として格納することを特徴とする仮想化プログラム。
【請求項４】
請求項３に記載の仮想化プログラムにおいて、
前記切替えは、前記待機中のタスクの最新レジスタ情報を前記所定のホストＣＰＵのレジスタに格納することでタスクの稼動を切替えることを特徴とする仮想化プログラム。
【請求項５】
マルチタスク処理を行うことができるＯＳのネイティブ・コード・シミュレーションにおいて、シミュレーション対象であるターゲットＯＳ上で生成されたタスクのそれぞれがターゲットＯＳ上で起動されるごとに、前記タスクのそれぞれが呼び出す関数順に関数呼び出し情報をLIFO構造でタスクごとに格納するスタック領域と、前記タスクのそれぞれが最後に実行中断したときのレジスタ内容を最新レジスタ情報としてタスクごとに格納するレジスタ保存領域と、前記タスクそれぞれの状態を示す状態情報をタスクごとに格納するタスク状態格納領域とを生成するコンテキスト生成部と、
所定のタイミングで、稼動しているタスクが最後に呼んだ関数の少なくとも一つ前に関数を呼び出した時点のレジスタ内容を前記最新レジスタ情報として前記レジスタ保存領域に格納するとともに、前記稼動しているタスクの状態を示す状態情報を前記タスク状態格納領域に格納するコンテキスト格納部と、
格納されている次に稼動すべき待機中のタスクの最新レジスタ情報および状態情報に基づき、前記稼動しているタスクから前記待機中のタスクへとタスクの稼動を切替えるタスク切替え部と、
を備えるシミュレーション装置。
【請求項６】
請求項５に記載のシミュレーション装置において、
前記タスク切替え部は、前記待機中のタスクの最新レジスタ情報に基づいたレジスタ内

容を、ホストＣＰＵのスタックレジスタに格納することで、ネイティブ・コード・シミュレーションにおけるターゲットＯＳ上のタスクの稼動を切替えることを特徴とするシミュレーション装置。
【請求項７】
請求項５または請求項６に記載のシミュレーション装置において、
前記コンテキスト格納部は、ホストＣＰＵの所定のレジスタに格納された値を、前記稼動しているタスクの最新レジスタ情報として前記レジスタ保存領域に格納することを特徴とするシミュレーション装置。
【請求項８】
請求項７に記載のシミュレーション装置において、
前記タスク切替え部は、前記待機中のタスクの最新レジスタ情報を前記所定のホストＣＰＵのレジスタに格納することでタスクの稼動を切替えることを特徴とするシミュレーション装置。
【請求項９】
マルチタスク処理を行うことができるＯＳのネイティブ・コード・シミュレーションにおいて、シミュレーション対象であるターゲットＯＳ上で生成されたタスクのそれぞれがターゲットＯＳ上で起動されるごとに、前記タスクのそれぞれが呼び出す関数順に関数呼び出し情報をLIFO構造でタスクごとに格納するスタック領域と、前記タスクのそれぞれが最後に実行中断したときのレジスタ内容を最新レジスタ情報としてタスクごとに格納するレジスタ保存領域と、前記タスクそれぞれの状態を示す状態情報をタスクごとに格納するタスク状態格納領域とを生成し、
所定のタイミングで、稼動しているタスクが最後に呼んだ関数の少なくとも一つ前に関数を呼び出した時点のレジスタ内容を前記最新レジスタ情報として前記レジスタ保存領域に格納するとともに、前記稼動しているタスクの状態を示す状態情報を前記タスク状態格納領域に格納し、
格納されている次に稼動すべき待機中のタスクの最新レジスタ情報および状態情報に基づき、前記稼動しているタスクから前記待機中のタスクへとタスクの稼動を切替える仮想化方法。
【請求項１０】
請求項９に記載の仮想化方法において、
前記切替えは、前記待機中のタスクの最新レジスタ情報に基づいたレジスタ内容を、ホストＣＰＵのスタックレジスタに格納することで、ネイティブ・コード・シミュレーションにおけるターゲットＯＳ上のタスクの稼動を切替えることを特徴とする仮想化方法。
【請求項１１】
請求項９または請求項１０に記載の仮想化方法において、
前記レジスタ保存領域への格納は、ホストＣＰＵの所定のレジスタに格納された値を、前記稼動しているタスクの最新レジスタ情報として格納することを特徴とする仮想化方法。
【請求項１２】
請求項１１に記載の仮想化方法において、
前記切替えは、前記待機中のタスクの最新レジスタ情報を前記所定のホストＣＰＵのレジスタに格納することでタスクの稼動を切替えることを特徴とする仮想化方法。
【発明の詳細な説明】
【技術分野】
【０００１】
本発明は、マルチタスク処理を行うことができるＯＳを含めたソフトウェアのシミュレーションにて、ターゲットＯＳ上で生成されたマルチタスクをホストＣＰＵのネイティブコードでシミュレーション実行する仮想化プログラム、シミュレーション装置、仮想化方法に関する。
【背景技術】

【０００２】
一般的なシミュレーション方式として、ＩＳＳ方式（Instruction Set Simulation：命令セットシミュレーション）と、ネイティブ・コード・シミュレーション方式がある。
【０００３】
ＩＳＳ方式は、ターゲットＣＰＵ（シミュレーション対象となる計算機のＣＰＵ）（仮想ＣＰＵ）のアーキテクチャに沿ったバイナリコードのタスクを実行時に解析して模擬実行する方式であるが、このＩＳＳ方式では、シミュレーションの実行に長時間を要するため、ＯＳを含めたソフトウェア全体を現実的な時間内でシミュレートすることはできない。一方、ネイティブ・コード・シミュレーション方式は、ホストＣＰＵ（シミュレーションを実行する計算機の演算処理装置）のアーキテクチャに沿ったバイナリコードのタスクを、ホストＣＰＵで直接実行する方式であるため、ターゲットＣＰＵのアーキテクチャに沿ったバイナリコードへの解析、変換が不要となり、シミュレーションの実行を高速化することが可能である。
【０００４】
また、ターゲットＯＳ（ターゲットＣＰＵ上で稼動するシミュレーション対象となるＯＳ）がマルチタスクＯＳの場合、ターゲットＯＳはターゲットＣＰＵが有するスタックレジスタやＰＳＲ（Program Status Register）といった特殊な制御レジスタを操作することによって、タスクのコンテキスト（タスクの実行に必要なデータ）の保存、復元を行い、マルチタスクのコンテキスト・スイッチングを実現する。これらの特殊な制御レジスタの操作はアセンブラ言語で記述されている。
【０００５】
なお、本発明の関連ある従来技術として、タイミング検証の精度を低下させることなく、Ｃベースのネイティブ・コード・シミュレーションを実現することにより、シミュレーションの高速化を実現したハードウェア／ソフトウェア協調検証方法が開示されている（例えば特許文献１）。また、半導体装置に搭載されるハードウェア／ソフトウェアの協調検証を実行する際に必要なソフトウェア検証モデルを生成する方法に関し、バジェット処理の追加方法を最適化してシミュレーション精度を維持しつつ、その性能を向上させる方法が開示されている（例えば特許文献２）。
【特許文献１】特開２００４－２３４５２８号公報
【特許文献２】特開２００５－２９３２１９号公報
【発明の開示】
【発明が解決しようとする課題】
【０００６】
しかしながら、ネイティブ・コード・シミュレーションを実行するためには、ターゲットＯＳを含めたソフトウェアを全てＣ言語ベースに置き換えなければならない。
【０００７】
また、ネイティブ・コード・シミュレータでは上述のスタックレジスタがモデル化されてない。よって、ターゲットＯＳがタスク毎のコンテキストを保存および復元することができないため、マルチタスクの実行をする際には、そのタスクの管理をシミュレーションエンジンとターゲットＯＳが密に連携して行う必要がある。本来、ターゲットＯＳ上のタスクはターゲットＯＳにて管理されるべきである。シミュレーションエンジンが提供する仮想ＣＰＵ上でターゲットＯＳが稼動し、ターゲットＯＳの管理下でタスクが実行されるべきであるが、上述のようにシミュレーションエンジンとターゲットＯＳの密な連携によるタスクの管理では、本来の階層の整合がとれていない構成となり、ターゲットＯＳ種別ごとにシミュレーションエンジン側の処理を用意しなければならない。
【０００８】
本発明は上述した課題を解決するためになされたものであり、ネイティブ・コード・シミュレータにおいて、ターゲットＯＳ上のタスク毎に固有なスタック領域を持ち、タスク切替えのタイミングでターゲットＯＳによるコンテキスト・スイッチングを可能とすることで、マルチタスクＯＳ上のタスクをネイティブコードシミュレーションを可能とするシ

ミュレーション装置、仮想化プログラム、仮想化方法を提供することを目的とする。
【課題を解決するための手段】
【０００９】
上述した課題を解決するため、本発明は、仮想プログラムであって、マルチタスク処理を行うことができるＯＳのネイティブ・コード・シミュレーションにおいて、シミュレーション対象であるターゲットＯＳ上で生成されたタスクのそれぞれがターゲットＯＳ上で起動されるごとに、前記タスクのそれぞれが呼び出す関数順に関数呼び出し情報（Call Stack）をLIFO（後入れ先出し）構造でタスクごとに格納するスタック領域と、前記タスクのそれぞれが最後に実行中断したときのレジスタ内容を最新レジスタ情報としてタスクごとに格納するレジスタ保存領域と、前記タスクそれぞれの状態を示す状態情報をタスクごとに格納するタスク状態格納領域（タスク・コントロール・ブロック：TCB）とを生成し、所定のタイミングで、稼動しているタスクが最後に呼んだ関数の少なくとも一つ前に関数を呼び出した時点のレジスタ内容を前記最新レジスタ情報として前記レジスタ保存領域に格納するとともに、前記稼動しているタスクの状態を示す状態情報を前記タスク状態格納領域に格納し、格納されている次に稼動すべき待機中のタスクの最新レジスタ情報および状態情報に基づき、前記稼動しているタスクから前記待機中のタスクへとタスクの稼動を切替える処理をコンピュータに実行させるものである。
【００１０】
また、本発明は、上述に記載の仮想化プログラムにおいて、前記切替えは、前記待機中のタスクの最新レジスタ情報に基づいたレジスタ内容を、ホストＣＰＵのスタックレジスタに格納することで、ネイティブ・コード・シミュレーションにおけるターゲットＯＳ上のタスクの稼動を切替えることを特徴とするものである。
【００１１】
また、本発明は、上述に記載の仮想化プログラムにおいて、前記レジスタ保存領域への格納は、ホストＣＰＵの所定のレジスタに格納された値を、前記稼動しているタスクの最新レジスタ情報として格納することを特徴とするものである。
【００１２】
また、本発明は、上述に記載の仮想化プログラムにおいて、前記切替えは、前記待機中のタスクの最新レジスタ情報を前記所定のレジスタに格納することでタスクの稼動を切替えることを特徴とするものである。
【００１３】
上述した課題を解決するため、本発明は、シミュレーション装置であって、マルチタスク処理を行うことができるＯＳのネイティブ・コード・シミュレーションにおいて、シミュレーション対象であるターゲットＯＳ上で生成されたタスクのそれぞれがターゲットＯＳ上で起動されるごとに、前記タスクのそれぞれが呼び出す関数順に関数呼び出し情報（Call Stack）をLIFO（後入れ先出し）構造でタスクごとに格納するスタック領域と、前記タスクのそれぞれが最後に実行中断したときのレジスタ内容を最新レジスタ情報としてタスクごとに格納するレジスタ保存領域と、前記タスクそれぞれの状態を示す状態情報をタスクごとに格納するタスク状態格納領域（タスク・コントロール・ブロック：TCB）とを生成するコンテキスト生成部と、所定のタイミングで、稼動しているタスクが最後に呼んだ関数の少なくとも一つ前に関数を呼び出した時点のレジスタ内容を前記最新レジスタ情報として前記レジスタ保存領域に格納するとともに、前記稼動しているタスクの状態を示す状態情報を前記タスク状態格納領域に格納するコンテキスト格納部と、格納されている次に稼動すべき待機中のタスクの最新レジスタ情報および状態情報に基づき、前記稼動しているタスクから前記待機中のタスクへとタスクの稼動を切替えるタスク切替え部と、を備えるものである。
【００１４】
また、本発明は、上述に記載のシミュレーション装置において、前記タスク切替え部は、前記待機中のタスクの最新レジスタ情報に基づいたレジスタ内容を、ホストＣＰＵのスタックレジスタに格納することで、ネイティブ・コード・シミュレーションにおけるター

ゲットＯＳ上のタスクの稼動を切替えることを特徴とするものである。
【００１５】
また、本発明は、上述に記載のシミュレーション装置において、前記コンテキスト格納部は、ホストＣＰＵの所定のレジスタに格納された値を、前記稼動しているタスクの最新レジスタ情報として前記状態格納領域に格納することを特徴とするものである。
【００１６】
また、本発明は、上述に記載のシミュレーション装置において、前記タスク切替え部は、前記待機中のタスクの最新レジスタ情報を前記所定のホストＣＰＵのレジスタに格納することでタスクの稼動を切替えることを特徴とするものである。
【００１７】
上述した課題を解決するため、本発明は、仮想化方法であって、マルチタスク処理を行うことができるＯＳのネイティブ・コード・シミュレーションにおいて、シミュレーション対象であるターゲットＯＳ上で生成されたタスクのそれぞれがターゲットＯＳ上で起動されるごとに、前記タスクのそれぞれが呼び出す関数順に関数呼び出し情報（Call Stack）をLIFO（後入れ先出し）構造でタスクごとに格納するスタック領域と、前記タスクのそれぞれが最後に実行中断したときのレジスタ内容を最新レジスタ情報としてタスクごとに格納するレジスタ保存領域と、前記タスクそれぞれの状態を示す状態情報をタスクごとに格納するタスク状態格納領域（タスク・コントロール・ブロック：TCB）とを生成し、所定のタイミングで、稼動しているタスクが最後に呼んだ関数の少なくとも一つ前に関数を呼び出した時点のレジスタ内容を前記最新レジスタ情報として前記レジスタ保存領域に格納するとともに、前記稼動しているタスクの状態を示す状態情報を前記タスク状態格納領域に格納し、格納されている次に稼動すべき待機中のタスクの最新レジスタ情報および状態情報に基づき、前記稼動しているタスクから前記待機中のタスクへとタスクの稼動を切替えるものである。
【００１８】
また、本発明は、上述に記載の仮想化方法において、前記切替えは、前記待機中のタスクの最新レジスタ情報に基づいたレジスタ内容を、ホストＣＰＵのスタックレジスタに格納することで、ネイティブ・コード・シミュレーションにおけるターゲットＯＳ上のタスクの稼動を切替えることを特徴とするものである。
【００１９】
また、本発明は、上述に記載の仮想化方法において、前記状態格納領域への格納は、ホストＣＰＵの所定のレジスタに格納された値を、前記稼動しているタスクの最新レジスタ情報として格納することを特徴とするものである。
【００２０】
また、本発明は、上述に記載の仮想化方法において、前記切替えは、前記待機中のタスクの最新レジスタ情報を前記所定のホストＣＰＵのレジスタに格納することでタスクの稼動を切替えることを特徴とするものである。
【発明の効果】
【００２１】
本発明によれば、マルチタスク処理を行うことが可能なＯＳに対し、複数のタスクを切替えて高速にシミュレーション実行することができる。
【発明を実施するための最良の形態】
【００２２】
本実施の形態におけるシミュレーション装置の構成を図１を参照しつつ説明する。
【００２３】
シミュレーション装置１００は、ホストＣＰＵ１０（ＣＰＵ：Central Processing Unit）、ホストＯＳ１１（ＯＳ：Operating System）、シミュレーションエンジン１、仮想ＣＰＵ１２、ターゲットＯＳ１３、タスク１４Ａ、タスク１４Ｂ（タスクを総称するときはタスク１４と記す）、ＨＷモデル２０（ＨＷ：ＨａｒｄＷａｒｅ）、を備える。
【００２４】

ホストＣＰＵ１０は、シミュレーション装置１００の所定のアーキテクチャに基づきハードウェアとして実装された演算処理装置であり、ホストＯＳ１１は、ホストＣＰＵ１０のアーキテクチャ上で稼動するＯＳである。

【００２５】

シミュレーションエンジン１は、仮想ＣＰＵ１２とＨＷモデル２０の実行環境を提供し、各モデルをスケジューリングするタイミングを調整しつつ、ソフトウェアおよびハードウェアをシミュレーション実行する仮想化プログラムである。

【００２６】

また、仮想ＣＰＵ１２は、ホストＯＳ１１上でターゲットＣＰＵの動作を模擬実行する仮想ＣＰＵである。尚、本実施の形態では、仮想ＣＰＵ１２での実行モードは全て同一特権モードで実行されるものとする。尚、ネイティブ・コード・シミュレータではスタックレジスタがモデル化されていないが、仮想ＣＰＵ１２のスタックレジスタはホストＣＰＵ１０のスタックレジスタをマッピングしている。つまり、ターゲットＯＳ１２が仮想ＣＰＵ１２のスタックレジスタを読み／書きすると、ホストＣＰＵ１０のスタックレジスタが読み／書きされる。

【００２７】

ターゲットＯＳ１３は、仮想ＣＰＵ１２上で稼動するマルチタスク処理を行うことができるＯＳである。尚、本実施の形態におけるターゲットＯＳ１３は、多重仮想記憶ＯＳではなく、モノリシック空間ＯＳとする。

【００２８】

タスク１４は、ターゲットＯＳ１３上で稼動する所定の処理を実行するユーザプログラムである。またタスク１４は、ホストＣＰＵ１０のアーキテクチャに準拠したネイティブコードで生成（コンパイル）されたプログラムである。

【００２９】

ＨＷモデル２０は、例えばＣＣＤカメラ、スキャナ等、所定の周辺機器の動作をソフトウェアとしてモデル化したものである。

【００３０】

本実施の形態のシミュレーション装置１００は、上述の仮想ＣＰＵ１２、ターゲットＯＳ１３、タスク１４、ＨＷモデル２０に相当する構成のハードウェア／ソフトウェア環境の動作を検証するためのものである。以下、必要に応じこのモデル化された環境全体を検証対象と記す。

【００３１】

尚、シミュレーションエンジン１は、タスクのコンテキストの生成・退避（保存）・復元・消去の処理をＡＰＩとして備え、例えばターゲットＯＳ１３にて各ＡＰＩが呼ばれることで機能が実現される。ターゲットＯＳ１３のポーティングでは、シミュレーションエンジン１のＡＰＩを呼び出すようにコードが一部変更されることで、タスク毎に固有なスタック領域が割り当てられる。タスク切替え時にはスタック領域の切替えが行われることで、コンテキストの切替えが可能となる。タスク１４はネイティブコードを直接実行する方式なので、スタック領域の内部構造はホストＣＰＵ１０上のホストＯＳ１１が使用するスタック領域の構造と同一とすることで、シミュレーションエンジン１はサポートするホストＣＰＵ１０、ホストＯＳ１１毎のコンテキスト操作ＡＰＩの処理を有することになる。

【００３２】

また、タスク１４Ａ、タスク１４Ｂが並列して実行されるようにするために、シミュレーションエンジン１は、コンテキスト生成部２、コンテキスト保存部３（コンテキスト格納部）、およびコンテキスト復元部４（タスク切替え部）を備える。ここで、コンテキスト生成部２、コンテキスト保存部３、およびコンテキスト復元部４について説明するが、以下の説明では、現在稼働中のタスクをタスク１４Ａとし、次に稼動すべきタスク（待機中のタスク）をタスク１４Ｂとすることで、タスク１４Ａからタスク１４Ｂに稼動タスクを切替えることとして説明する。

【００３３】
コンテキスト生成部２は、タスク１４のそれぞれが呼び出す関数順に関数呼び出し情報（Call Stack）をLIFO（後入れ先出し）構造でタスク１４ごとに格納するためのスタック領域と、タスク１４の最新レジスタ情報を格納するためのレジスタ保存領域とを生成し、またタスク１４それぞれの状態を示すタスク状態情報をタスク１４ごとに格納するタスク状態格納領域（以下、ＴＣＢ（Task Control Block）と表記）を生成する。尚、本実施の形態の説明のように、レジスタ保存領域をＴＣＢ内部に持つこともあり、その場合ＴＣＢにはレジスタ保存領域とタスク状態情報領域が存在する。
【００３４】
コンテキスト保存部３は、現在稼動しているタスク１４Ａが最後に呼んだ関数を呼び出した時点のレジスタ内容を最新レジスタ情報としてタスク１４ＡのＴＣＢに格納するとともに、稼動しているタスク１４Ａの状態を示すタスク状態情報をタスク１４ＡのＴＣＢに格納する。尚、コンテキスト保存部３は、タスク１４Ｂに対しても同様の処理をする。
【００３５】
コンテキスト復元部４は、次に稼動すべき待機中のタスク１４Ｂに対するＴＣＢに格納された最新レジスタ情報およびタスク状態情報に基づき、稼動しているタスク１４Ａから待機中のタスク１４Ｂへとタスクの稼動を切替える。コンテキスト復元部４は、タスク１４ＢからタスクＡへの切替えも同様の処理をする。
【００３６】
上述のコンテキスト保存部３と、コンテキスト復元部４との処理が所定のタイミング（例えば、ターゲットＯＳ１３の割り込みスケジューラ（IRS：Interrupt Routine Scheduler）による割り込みの発生や、ユーザのシステムコール呼び出しに伴う割り込み指示）により行われることで、マルチタスク処理を可能とするターゲットＯＳ１３上で、タスクの切替えが実現される。
【００３７】
次に、本実施の形態におけるシミュレーション装置１００の処理を、図２のフローチャートに基づき説明する。尚、以下の説明においては、タスクの生成においてはタスク１４Ａがタスク１４Ｂを生成し、またタスクの切替えにおいてはタスク１４Ａからタスク１４Ｂに切り替わるものとして説明する。
【００３８】
タスク１４Ａからタスク１４Ｂのタスク生成要求がなされた場合、コンテキスト生成部２は、タスク１４Ｂに対応するＴＣＢ、スタック領域を生成し、生成したスタック領域のルートフレーム（タスク実行開始時に必要な情報が格納される管理領域）を生成する（ステップＳ１）。
【００３９】
ここで、タスク１４Ａの関数の構成および、タスク１４Ａ、タスク１４Ｂのスタック領域、ＴＣＢについて、図３を参照しつつ説明する。
【００４０】
コンテキストを生成する際のタスク１４Ａの関数呼び出しが、図３の「タスク１４Ａの関数構造（コンテキスト生成）」に示すように、ｍａｉｎ（）を主関数として、ユーザプログラムの関数ｆｕｎｃ＿ａ（）があり、ｆｕｎｃ＿ａ（）の中から、ターゲットＯＳにて用意されたタスクを生成するためのＡＰＩであるＣＲＥ＿ＴＡＳＫ（）を呼び出している。ＣＲＥ＿ＴＡＳＫ（）の中にさらにシミュレーションエンジン１にて用意された関数であり、コンテキスト生成部２の機能であるｃｏｎｔｅｘｔ＿ｃｒｅａｔｅ（）が呼び出される構造であるものとする。
【００４１】
ターゲットＯＳのタスク生成機能であるＣＲＥ＿ＴＡＳＫ（）が呼ばれることで作成されるスタック領域には、図３の「スタック領域、ＴＣＢ関係」にて示すように、ＬＩＦＯのスタック構造にて、呼ばれる関数順に関数呼び出し情報が格納されていく。上述のタスク１４Ａでは、スタック領域のルートフレームがタスク起動時に作成され、その後、タス

ク１４Ａの処理が進むにつれ、関数が呼ばれた順にｍａｉｎ（）フレーム、ｆｕｎｃ＿ａ（）フレーム、ＣＲＥ＿ＴＡＳＫ（）フレーム、ｃｏｎｔｅｘｔ＿ｃｒｅａｔｅ（）フレーム（フレーム：関数呼び出し情報を格納する単位）がスタック領域に積まれる。

【００４２】

尚、タスク生成機能であるＣＲＥ＿ＴＡＳＫ（）によって、タスク１４Ｂが生成され、コンテキスト生成部２のｃｏｎｔｅｘｔ＿ｃｒｅａｔｅ（）によって必要な設定がなされるが、ここで、ＣＲＥ＿ＴＡＳＫ（）と、その内部にて呼ばれるｃｏｎｔｅｘｔ＿ｃｒｅａｔｅ（）の処理について説明する。

【００４３】

ＣＲＥ＿ＴＡＳＫ（）は、タスク１４Bを起動するとともにＴＣＢ（以下、タスク１４Ａに対応するＴＣＢをＴＣＢ（Ａ）と表記し、タスク１４Ｂに対応するＴＣＢをＴＣＢ（Ｂ）と表記する）を作成し、そのタスクに関する情報の初期値（レジスタの初期値、タスク状態情報の初期値）を設定する。また、ＣＲＥ＿ＴＡＳＫ（）は、起動されたタスクとして実行するプログラムを配置するためのアドレス空間（スタック領域を含む）を割り当てる。尚、本実施の形態におけるＣＲＥ＿ＴＡＳＫ（）は、ターゲットＯＳ１３がモノリシック空間ＯＳであるため、実メモリのアドレス空間を切り分けてスタック領域として割り当てる。一方、仮想記憶ＯＳの場合は、ページテーブルなど仮想空間制御情報を作成する。

【００４４】

またｃｏｎｔｅｘｔ＿ｃｒｅａｔｅ（）は、スタック領域のルートフレームを作成し、起動されたタスク実行開始時に必要な情報（例えば、そのタスクの環境変数、ｍａｉｎ関数の開始アドレス等）をルートフレームに設定する。さらに、タスクスケジューリングの対象となるように、作成したＴＣＢをターゲットＯＳ１３が管理するＴＣＢキュー（タスク・レディ・キューなど）に登録する。

【００４５】

このようにタスク生成機能を実現するＣＲＥ＿ＴＡＳＫ（）とその内部で呼び出すｃｏｎｔｅｘｔ＿ｃｒｅａｔｅ（）によって、タスク１４Ｂのコンテキストを格納する領域を作成することができる。尚、作成されるスタック領域は、ホストＯＳ１１が検証対象のスレッドに割り当てたスタック領域を細分化してもよく、またはメモリアロケード関数（malloc関数等）などホストＯＳ１１のメモリ領域割り当てサービスで獲得した領域を使用しても良い。

【００４６】

このようにコンテキストの生成が完了すると、ｃｏｎｔｅｘｔ＿ｃｒｅａｔｅ（）が終了し（ｃｏｎｔｅｘｔ＿ｃｒｅａｔｅ（）呼び出しフレームがスタック領域から開放される）、ＣＲＥ＿ＴＡＳＫ（）が終了する（ＣＲＥ＿ＴＡＳＫ（）呼び出しフレームがスタック領域から開放される）。

【００４７】

尚、タスク１４ＡからタスクＢ１４のタスク生成要求が成され、タスク１４Aがタスク１４Ｂを動的にタスク生成したが、ターゲットＯＳ１３がＯＳ構成情報に基づきタスク１４Ｂを静的にタスク生成してもよい。

【００４８】

図２のフローチャートに説明を戻す。次に、上述のようにタスク１４Ｂが生成された後のタスク１４Ａとタスク１４Ｂを並列に実行するためのタスク切替え処理について説明する。

【００４９】

コンテキスト保存部３は、ホストＣＰＵ１０のスタックに格納されている、カレントフレームから１フレーム分前のフレームをカレントフレームポインタとしてＴＣＢに格納する。また、コンテキスト保存部３は、仮想ＣＰＵの制御レジスタであるＰＳＲ（タスクの状態を示す値を格納するレジスタ）に格納されている値（レジスタ情報）をＴＣＢに格納する（ステップＳ２）。

【００５０】
このコンテキスト保存部３にて行われる処理を、図４に基づきタスク１４Ａを例として説明する。
【００５１】
コンテキストをＴＣＢに保存する際のタスク１４Ａの関数呼び出しが、図４の「タスク１４Ａの関数構造（コンテキスト保存）」に示すような構造であるものとする。すなわち、ｍａｉｎ（）を主関数として、ユーザプログラムの関数ｆｕｎｃ＿ｂ（）があり、ｆｕｎｃ＿ｂ（）の中からターゲットＯＳ１３が提供する何らかのシステムコールを呼び出し（ｓｙｓｃａｌｌ（）関数が呼ばれる）、システムコールの中でタスク切替えを行うためにタスク切替え関数（Ｔａｓｋ＿ｓｗｉｔｃｈ（））が呼ばれ、さらにＴａｓｋ＿ｓｗｉｔｃｈ（）の内部関数として、コンテキスト保存部３の機能であるｃｏｎｔｅｘｔ＿ｓａｖｅ（）が呼ばれる構造であるとする。
【００５２】
上述の状況においては、仮想ＣＰＵ１２は、現在ｃｏｎｔｅｘｔ＿ｓａｖｅ（）を実行しており、仮想ＣＰＵ１２のスタックレジスタは、ｃｏｎｔｅｘｔ＿ｓａｖｅ（）フレームのアドレス（図中、ｆｐと表記）が格納されている状態である。ここで、ｃｏｎｔｅｘｔ＿ｓａｖｅ（）は、自身のｃｏｎｔｅｘｔ＿ｓａｖｅ（）フレームの一つ前の関数（すなわちＴａｓｋ＿ｓｗｉｔｃｈ（））のフレームのアドレスをＴＣＢ（Ａ）に格納する。
【００５３】
また、ｃｏｎｔｅｘｔ＿ｓａｖｅ（）は、現在仮想ＣＰＵ１２の制御レジスタに格納されているレジスタ情報をタスク１４Ａのレジスタ保存領域に保存することで、ＴＣＢ（Ａ）に格納する。
【００５４】
このような処理を行い、ｃｏｎｔｅｘｔ＿ｓａｖｅ（）が終了すると、そのｃｏｎｔｅｘｔ＿ｓａｖｅ（）フレームも開放される。
【００５５】
このように、コンテキスト保存部３の機能であるｃｏｎｔｅｘｔ＿ｓａｖｅ（）は、ｃｏｎｔｅｘｔ＿ｓａｖｅ（）を呼び出した関数の関数呼び出し情報のアドレスをカレントフレームポインタとしてＴＣＢ（Ａ）に格納し、さらに、現在仮想ＣＰＵ１２の制御レジスタに格納されたレジスタ情報をＴＣＢ（Ａ）に格納する。この格納された情報に基づき、後にタスク１４Ａからタスク１４Ｂに切り替わりその後またタスク１４Ｂからタスク１４Ａに切り替わるとき、タスク１４Ａは切り替わる前の状態から処理を継続実行できる。
【００５６】
次にタスク１４ＡのコンテキストがＴＣＢ（Ａ）に保存された後の処理を図２のフローチャートに戻り説明する。
【００５７】
コンテキスト復元部４は、現在稼動しているタスク１４Ａのスタック領域に格納された関数呼び出し情報のうち現在実行されている関数の呼び出しフレーム（カレントフレーム）の１フレーム分をタスク１４Ｂ（次に稼動すべき待機中のタスク）のスタック領域にコピーする。また、コンテキスト復元部４は、ＴＣＢ（Ｂ）に格納されたレジスタ情報を仮想ＣＰＵ１２の制御レジスタに格納し、さらにＴＣＢ（Ｂ）に格納されているアドレスに１フレーム分加えたアドレスをカレントフレームポインタとして仮想ＣＰＵ１２のスタックレジスタに格納する（ステップＳ３）。
【００５８】
このコンテキスト復元部４にて行われる処理を、図５に基づき説明する。
【００５９】
図５では、現在タスク１４Ａが実行されており、上述のコンテキスト保存部３の処理（ｃｏｎｔｅｘｔ＿ｓａｖｅ（））が終了し、コンテキスト復元部４の機能であるｃｏｎｔｅｘｔ＿ｒｅｓｔｏｒｅ（）が呼ばれた状態を示している。
【００６０】

まず、タスク１４Ａでｃｏｎｔｅｘｔ＿ｒｅｓｔｏｒｅ（）がＴａｓｋ＿ｓｗｉｔｃｈ（）から呼ばれることで、ｃｏｎｔｅｘｔ＿ｒｅｓｔｏｒｅ（）は、自身のｃｏｎｔｅｘｔ＿ｒｅｓｔｏｒｅ（）関数呼び出しフレーム（１フレーム分）をタスク１４Ｂのスタック領域上にコピーする。その後、ｃｏｎｔｅｘｔ＿ｒｅｓｔｏｒｅ（）は、ＴＣＢ（Ｂ）に格納されているアドレスを、コピーした自身のｃｏｎｔｅｘｔ＿ｒｅｓｔｏｒｅ（）関数呼び出しフレームサイズ分だけ追加したアドレスに更新する。すなわち、ｃｏｎｔｅｘｔ＿ｒｅｓｔｏｒｅ（）は、ＴＣＢ（Ｂ）のスタックフレームポインタをＴａｓｋ＿ｓｗｉｔｃｈ（）関数呼び出しフレームのアドレスからｃｏｎｔｅｘｔ＿ｒｅｓｔｏｒｅ（）関数呼び出しフレームのアドレスに更新する。

【００６１】

その後、ｃｏｎｔｅｘｔ＿ｒｅｓｔｏｒｅ（）は、更新されたポインタの内容を仮想ＣＰＵ１２のスタックレジスタに格納し、更に、ＴＣＢ（Ｂ）内のレジスタ保存領域に格納されたレジスタ情報を仮想ＣＰＵ１２の制御レジスタに格納する（尚、図５においてはレジスタ情報に関しては図示せず）。

【００６２】

このようにすることで、仮想ＣＰＵ１２は、タスクが切り替わったことを何ら意識することなく自身のスタックレジスタに格納されているアドレスの処理（すなわち、タスク１４Ｂのスタック領域に積まれたｃｏｎｔｅｘｔ＿ｒｅｓｔｏｒｅ（））をそのまま実行する。

【００６３】

その後、上述のように一連の切替え処理が完了したため、ｃｏｎｔｅｘｔ＿ｒｅｓｔｏｒｅ（）の処理が終了し（ｃｏｎｔｅｘｔ＿ｒｅｓｔｏｒｅ（）フレームが開放される）、タスク１４ＢのＴａｓｋ＿ｓｗｉｔｃｈ（）の処理が終了する（Ｔａｓｋ＿ｓｗｉｔｃｈ（）フレームが開放される）。また、タスク１４Ｂのシステムコール呼び出し（ｓｙｓｃａｌｌ（））が終了することで（ｓｙｓｃａｌｌ（）フレームが開放される）、タスク１４Ｂは待機状態になる前の処理（図５においてはユーザ関数であるｆｕｎｃ＿ｄ（））を継続実行する。

【００６４】

上述のｃｏｎｔｅｘｔ＿ｓａｖｅ（）、ｃｏｎｔｅｘｔ＿ｒｅｓｔｏｒｅ（）の処理が、ターゲットＯＳ１３の割り込みスケジューラによる割り込み発生や、システムコール呼び出し（ｓｙｓｃａｌｌ（））が発生するタイミングでＴａｓｋ＿ｓｗｉｔｃｈ（）から呼ばれることで、タスク１４Ａ、タスク１４Ｂのタスクの切替えが行われる。

【００６５】

上述のようにタスク１４Ｂが稼動状態になり、次にタスク１４Ｂからタスク１４Ａに切り替わる場合は、タスク１４Ｂの処理中にｃｏｎｔｅｘｔ＿ｓａｖｅ（）が呼ばれ、仮想ＣＰＵ１２の現在の制御レジスタ、およびスタックポインタ（１フレーム分減算したもの）をＴＣＢ（Ｂ）に保存する処理がなされる。その後、タスク１４Ｂの処理としてｃｏｎｔｅｘｔ＿ｒｅｓｔｏｒｅ（）が呼ばれることで、ｃｏｎｔｅｘｔ＿ｒｅｓｔｏｒｅ（）は、自身のｃｏｎｔｅｘｔ＿ｒｅｓｔｏｒｅ（）関数呼び出しフレームをタスク１４Ａにコピーし、ＴＣＢ（Ａ）に格納されたスタックフレームポインタに１フレーム分加えたもの（ｃｏｎｔｅｘｔ＿ｒｅｓｔｏｒｅ（）関数呼び出しフレームのサイズ分を加えたもの）に更新し、更新後のスタックフレームポインタの内容をスタックレジスタに格納するとともにＴＣＢ（Ａ）に格納された状態情報も制御レジスタに格納することで、タスク１４Ａにて、自身のｃｏｎｔｅｘｔ＿ｒｅｓｔｏｒｅ（）処理が引き続きなされるようにする。

【００６６】

図２のフローチャートに戻り、最後にタスクの終了について、タスク１４Ａを例に説明する。タスク１４Ａ内の関数が１つずつ終了し（スタック領域のフレームが１つずつ開放され）、最後のｍａｉｎ（）が終了することで、ルートフレームとともにタスク１４Ａとして割り当てられたスタック領域が開放され、タスク１４Ａは終了する（ステップＳ４）

。

【００６７】

尚、本実施の形態では、２つのタスクのコンテキストが生成され、２つのタスクの切替え処理について記したが、より多いタスクの生成、切替えも適用可能である。

【００６８】

また、本実施の形態におけるコンテキスト保存部３は、スタック領域に格納された最後に実行された関数の１フレーム分を減算したアドレスを格納アドレス情報としてＴＣＢに格納し、コンテキスト復元部４は、格納アドレス情報に対しコピーした１フレーム分を加算したアドレスを仮想ＣＰＵ１２のスタックレジスタに格納したが、タスク切替えがどこのプログラムロケーションで発生したことにするのかによって、何フレーム分減算、加算するか、あるいは減算、加算を行わないかが異なる。本実施の形態ではＴａｓｋ＿ｓｗｉｔｃｈ（）関数の中でタスク切替えが発生したことにする場合とした。

【００６９】

また、本実施の形態では、仮想ＣＰＵ１２での実行モードは全て同一特権モードで実行されるものとしたが、割り込みスケジューラのみは別の特権モードとすることもできる。その場合は、割込み処理用のスタック領域をターゲットＯＳ１３起動時に作成し、割り込みスケジューラが割り込みハンドラをスケジュールするときに、にて割込み処理用スタックへの切替えを行えば良い。

【００７０】

また、本実施の形態では、ターゲットＯＳ１３を単一空間（モノリシック空間ＯＳ）のものとしたが、多重仮想記憶ＯＳの場合には複数タスクのタスク固有な空間を同時に参照・操作することはできない。つまり、多重仮想記憶でのスタック・フレーム・コピー操作が上述の実施の形態のような直接コピーでは実現できない。その場合、コピー元タスクのコンテキストで一旦ターゲットＯＳ１３のシステム領域のようなタスク共通領域にコピーして、タスクのコンテキストを切替えてから、コピー先タスクのスタックへコピーを行うことにより、上述の実施の形態と同等のコンテキスト切替えが可能になる。よって、本発明は、多重仮想記憶ＯＳにも適用可能である。

【００７１】

また、本実施の形態では仮想ＣＰＵ１２での実行モードは全て同一特権モードで実行されるものとしたが、システム保護を目的として、ターゲットＣＰＵでの実行モードにはユーザ・モード／特権モード／例外モードなどの複数の実行モードがあり、バンク切替えなどの仕組みによって実行モード毎に使用するスタックを切替えるのが一般的である。

【００７２】

そして、実行モードを切替えるときには、切替え後の実行モード用スタック領域のフレームの中に、切替え前の実行モード用スタック領域のカレントフレームポインタが格納される。その結果、現在の実行モードから旧実行モードに復帰する際、実行モード切替え発生時のカレントフレームポインタを復元することが可能となる。

【００７３】

本実施の形態におけるシミュレーションエンジン１によって、仮想ＣＰＵ１２、ターゲットＯＳ１３、タスク１４がシミュレーションエンジン１から見ると一つの実行単位であり、タスク１４Ａからタスク１４Ｂへのタスク切替えタイミングやタスク切替え条件はターゲットＯＳ１３によってのみ決定され、シミュレーションエンジン１からは透過となる。よって、ターゲットＯＳのタスク切替えアルゴリズムが変わってもシミュレーションエンジン１への変更は不要である。

【００７４】

本実施の形態において、仮想化プログラムは上述したシミュレーション装置の内部に予めインストールされているものとして記載したが、本発明における仮想化プログラムは記憶媒体に記憶されたものも含まれる。ここで記憶媒体とは、磁気テープ、磁気ディスク（フロッピーディスク、ハードディスクドライブ等）、光ディスク（ＣＤ－ＲＯＭ、ＤＶＤディスク等）、光磁気ディスク（ＭＯ等）、フラッシュメモリ等、シミュレーション装置

に対し脱着可能な媒体や、さらにネットワークを介することで伝送可能な媒体等、上述したシミュレーション装置におけるコンピュータで読み取りや実行が可能な全ての媒体をいう。

【図面の簡単な説明】

【００７５】

【図１】本実施の形態におけるシミュレーション装置の構成の一例を示す図である。

【図２】本実施の形態におけるシミュレーション装置の処理の一例を示すフローチャートである。

【図３】本実施の形態におけるコンテキスト生成の処理におけるタスクの関数構成およびスタック領域構成を示す図である。

【図４】本実施の形態におけるコンテキスト保存の処理におけるタスクの関数構成およびスタック領域構成を示す図である。

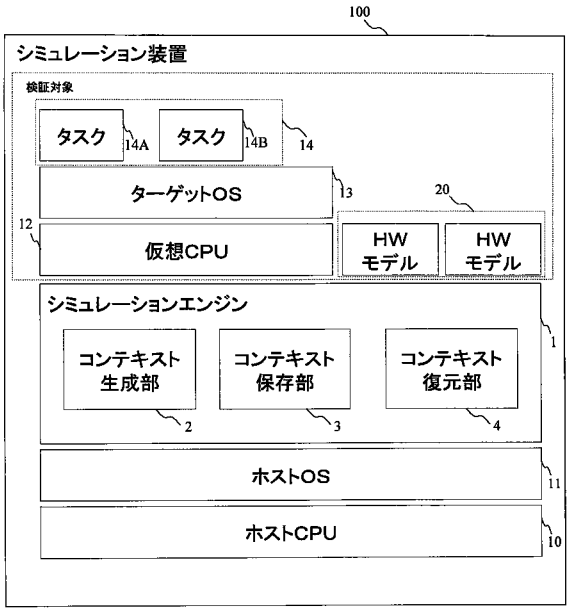
【図５】本実施の形態におけるコンテキスト復元の処理におけるタスクの関数構成およびスタック領域構成を示す図である。

【符号の説明】

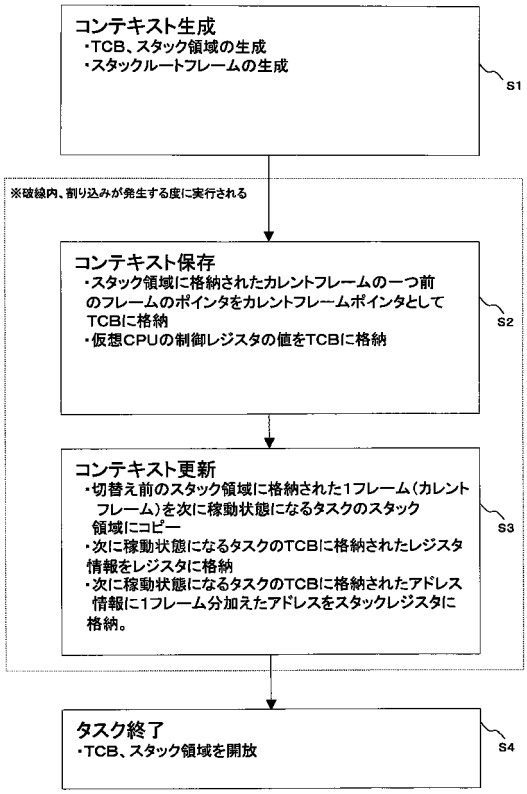
【００７６】

１　シミュレーションエンジン、２　コンテキスト生成部、３　コンテキスト保存部、４　コンテキスト復元部、１０　ホストＣＰＵ、１１　ホストＯＳ、１２　仮想ＣＰＵ、１３　ターゲットＯＳ、１４、１４Ａ、１４Ｂ　タスク、２０　ＨＷモデル、１００　シミュレーション装置。

【図１】

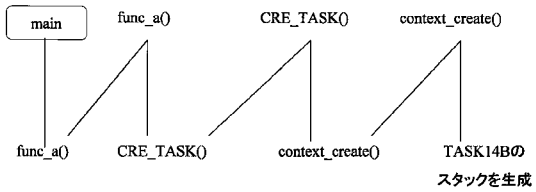


【図２】

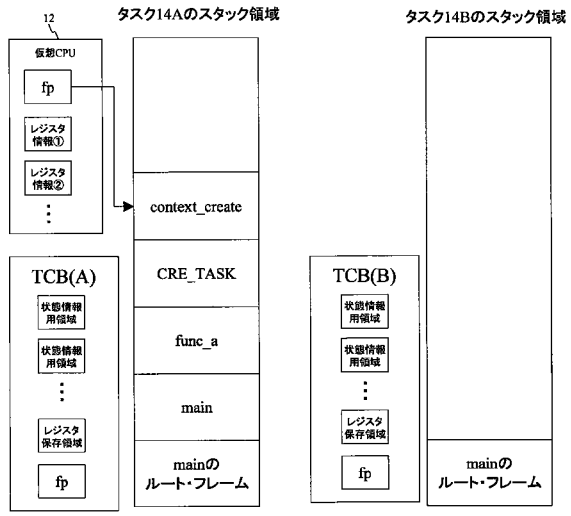


【図３】

タスク14Aの関数構造（コンテキスト生成）

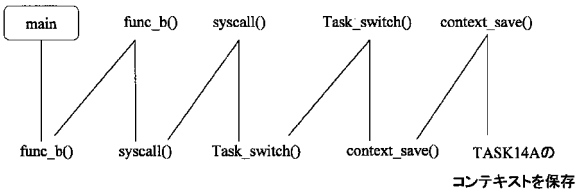


スタック領域、TCB関係

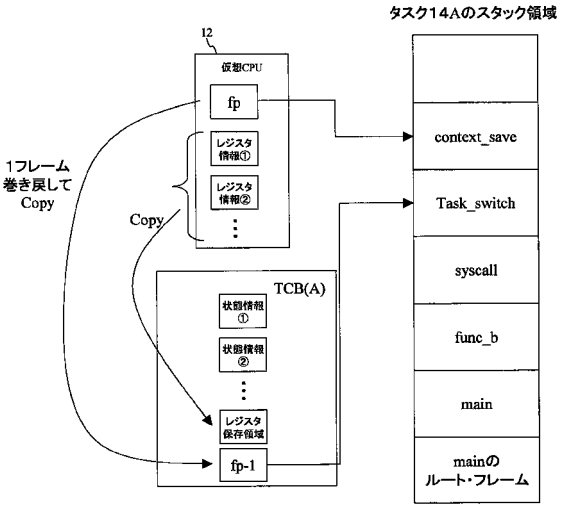


【図４】

タスク14Aの関数構造（コンテキスト保存）

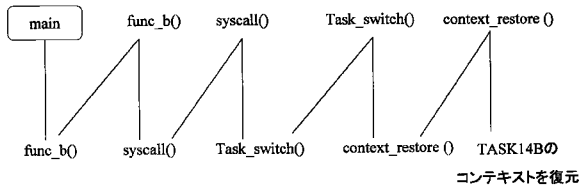


スタック領域、TCB関係



【図５】

タスク14Aの関数構造（コンテキスト更新）



スタック領域