DEOS-FY2012-SD-01J

JST-CREST

研究領域

「実用化を目指した組込みシステム用ディペンダブル・オペレーティングシステム」

DEOSプロジェクト

サービス延命を
可能とする基盤システムソフトウェア

2012/11/15

河野健二（慶應義塾大学）
光来健一（九州工業大学）
山田浩史（東京農工大学）

1. OS のディペンダビリティ

オープンシステムディペンダビリティ[1]を高めるためには，オペレーティングシステム(OS)の信頼性を確保することが必須となる．しかしながら，OS の肥大化･複雑化に伴って，OS 内には障害の原因と成り得るソフトウェアバグが含まれていたり[2]，OS をのっとる攻撃が登場している[3]．そのため，OS 自身の信頼性は高いとは言い難い．そこで，本では OS の障害を前提としたソフトウェア実行基盤について述べる．

1.1. OS の重要性

OS はプログラムを実行する基盤となるソフトウェアであり，サービスを構成する上で欠くことのできないソフトウェアレイヤのひとつである．OS はプログラムが実行できるようハードウェアを制御している．アプリケーションサーバやデータベースサーバのプログラムは OS とやりとりをしながらその動作を進めていく．また，ファイルのアクセス制御やウィルススキャナといったセキュリティに関する機構も通常のプログラムと同様に OS とやりとりをしながら処理を進めていく．OS はアプリケーションを効率良く実行できるよう，常に進化するアプリケーションに追従して日々新たな機能が追加されていっている．つまり，常に進化しつづけなければならないソフトウェアの 1 つである．

プログラムの実行基盤である OS には非常に高い信頼性が求められる．なぜなら，OS の不具合はその上で動作する全てのプログラムに影響するからである．たとえば，OS がバグを実行して停止したとする．OS が停止すると，バグを誘発したプログラムだけでなく，稼働していたプログラム全てが停止してしまう(図 1)．また，悪意のあるユーザの攻撃によって，OS 内に悪意のあるプログラムを埋め込まれるといった，OS が乗っ取られることはあってはならない．なぜなら，OS 内を自由に書き換えられてしまうため，サービスの停止やセキュリティ機構の無効化が容易に行えてしまうからである．こうした不具合は結果として，サービス停止やデータの消失，情報漏洩につながり大規模な障害を引き起こしてしまう．

図 1 OS のクラッシュ

1.2. OS の現状

しかしながら，現在の OS は高い信頼性を有しているとは言い難い．様々なアプリケーションの要求に応えるために OS は肥大化･複雑化しつづけてきた．その結果，基盤として動作する OS 内にもソフトウェアバグが存在し，その影響が顕在化することが少なくない．Palix らは Linux 2.6 シリーズの対象に Linux に含まれているバグを調査した[2]．その調査報告では，NULL ポインタチェック忘れや初期化忘れといった静的解析で判るようなシンプルなバグでさえも，常に約 700 個のバグが存在していることを示している．また，新しい機能にはバグが多い点などを明らかにしている．モデルチェックを用いてバグを積極的に避けることが可能である[4]が，商用で使われている Linux などの OS カーネルにおいて適用することは難しい．例えば割り込みハンドラのような非同期で呼ばれる処理部分に対して正しくモデルを作ることは困難である．

また，近年の悪意のあるソフトウェア(以下マルウェア)は，カーネルレベルルートキットを用いて OS をのっとることがある[3]．こうしたマルウェアはたちが悪く，ウィルススキャナといったセキュリティ機構に検知されないように自身の存在や活動を隠蔽することができる．結果として，既存の多くのセキュリティ機構の検知をかいくぐり，自身の活動をユーザに気づかれることなく行うことができる．そのため，カーネルルートキットの危険性は極めて高く，その検出手法や防御手法は世界で広く研究されている[5,6,7]．カーネルレベルルートキットは OS のバグを突き，OS 内のデータ構造や値を改竄する．たとえば，カーネル内で保持しているプロセスリストからマルウェアのプロセスに該当するリストを取り除くことで，アプリケーションが取得するプロセスリストからマルウェアの存在を隠蔽することができる．他にもファイルシステムのメタデータを改竄したり，システムコールの返り値を変更したりと様々な隠蔽手法が存在する．

1.3. OS の障害に向けて

アプリケーションやハードウェアの環境変化に OS は追従そして変化せねばならず，その OS から完全にソフトウェアバグを除去することは現実的ではない．また，OS を狙う攻撃も進化を続ける．そのため，OS は障害を起こすものであるというのを事実として受け入れてサービスを運用していく必要がある．そこで，OS に障害が起こることを前提として，その上でサービスの信頼性を確保するアプローチをとる．本報告書では，たとえ OS に不具合が生じたとしても，その影響を除去もしくは緩和して，提供しているサービスへの影響を最小限に抑える枠組みについて述べる．

2. Open System のための基盤ソフトウェア

オープンシステムにおいては常にソフトウェアは環境の変化にさらされており，環境の変化によるソフトウェアの動作の変化を迅速に補足し，環境変化への対応を促すようなシステム構成となっている必要がある．1 章でも論じたように，ソフトウェアシステム全体のディペンダビリティを担保するためには，その基盤となるオペレーティングシステム自身が高いディペンダビリティを有している必要がある．しかしながら，Linux をはじめ現在のオペレーティングシステムはバグのない形で実装することは事実上不可能になっていることは 1 章でも述べたとおりである．

本章では環境変化への適応を可能とする基盤ソフトウェアのための基本モデル，基本コンセプト，およびそれを実現するための機能要件について述べる．本システムではセキュリティ上の脆弱性もある種のフォールトと捉え，情報漏洩やマルウェア等への感染を一種のフェイラと捉えることによって，セキュリティ対策そのものも環境変化への対応と同等の枠組みで対処できるようになっている．本章ではその点についてもまとめる．

2.1. セキュリティの捉え方

ディペンダリティを確保するための枠組みとセキュリティを確保するための枠組みは，おそらく歴史的な理由から別個に論じられ扱われることが多い．フォールトトレラント・コンピューティングをバックグラウンドとする研究者・技術者と，セキュリティをバックグラウンドとする研究者・技術者とが十分に融合されていないためではないかと思われる．

本報告書ではセキュリティもこれまでに述べてきた枠組みの中で統一的に扱うようになっている．通常のフォールト（いわゆるバグ）とセキュリティホールでは発見の難しさが異なるため，セキュリティのための個別の枠組みが必要であるという主張もある．また，フォールトが顕在化したフェイラとセキュリティホールをつかれたことによるセキュリティインシデントでは被害の深刻さが異なることも多い．こうした現象面での違いは多くあるものの，セキュリティホールあるいは脆弱性も，究極的にはソフトウェアの不具合，すなわちフォールトであるという見方も可能である．セキュリティホールや脆弱をふ

さぐためにとられている処置の多くは，ソフトウェアの設定変更やパッチの適用などであり，いわゆるバグを修正する方法と本質的な違いは見られない．

また，セキュリティインシデントは，脆弱性やセキュリティホールを突かれ，それがセキュリティ上の問題となって顕在化していると捉えることができる．フェイラの多くは，システムの停止，応答時間の異常な増大，誤動作といった比較的認識しやすい形態をとるのに対し，セキュリティインシデントの場合は，何らかの不具合が生じていることが外部から認識しにくい場合が多い．たとえば，キーロガーにより銀行口座の情報などが漏洩している場合でも，一般のユーザから見ればソフトウェアシステムは期待通りに動作をし続けており，フェイラとして認識されることはない．銀行の口座から不正に預金が引き出されて始めて，情報漏洩の可能性に気づくに過ぎない．キーロガーへの感染といった外部から認識の難しい事象であっても，それを何らかの形で外部から「フェイラ」として認識可能にすることができれば，セキュリティインシデントも（多くの場合はインシデントの至る前に）フェイラとして認識可能である．

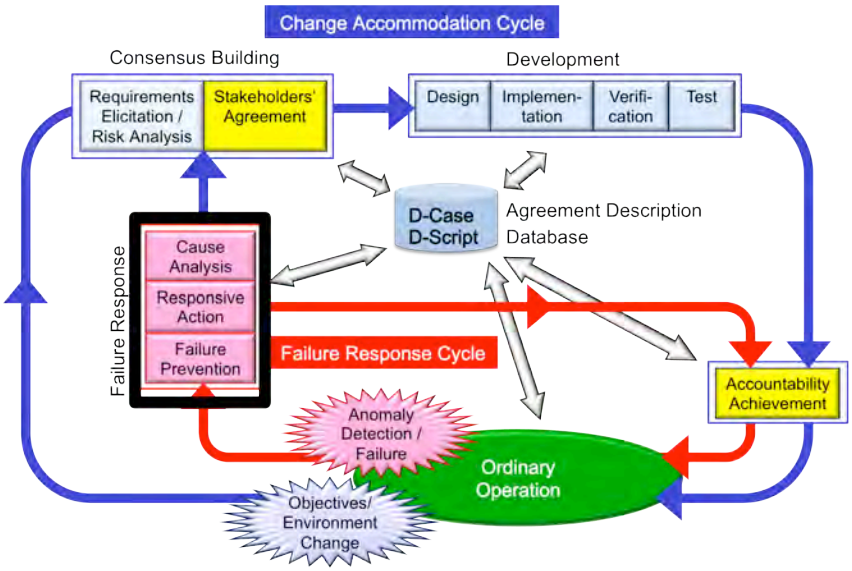

図 2 DEOS プロセスにおける本基盤ソフトウェアが貢献する部分

2.2. 基本モデル

2.1.1 概要

本報告書で述べる基盤ソフトウェア群は DEOS プロセスと呼ばれるディペンダビリティ確保のための一連のソフトウェアプロセスに組み込まれて動作することを前提に設計されている．しかし，本報告書で述べる個々の要素技術はいずれも単独で動作可能であり，必ずしも DEOS プロセスと一体化して利用する必要はない．

図２に DEOS プロセスの概要を示す．DEOS プロセスによる意志決定の契機となるのは，稼働中のソフトウェアシステムに何からの異常やその予兆を検知したり，あるいは動作環境の変化に伴って何らかの対応が必要となった時点である．DEOS プロセスに従って何らかの意志決定がなされ，具体的な対処方法が決まる．その具体的な対処方法は多岐に渡るものの，基本的には稼働中のソフトウェアシステムに対する何らかの変更として実施される．たとえば，ソフトウェアの設定変更，アップデート，システム再起動，あるいは一部機能を縮退しての動作継続などである．本報告書ではさまざまな状況に応じてソフトウェアシステムを広い意味で更新可能とするための基盤技術について述べる．具体的には図２中の黒く囲われた部分を実現するための基盤ソフトウェア技術である．なお，DEOS プロセスにおける意志決定プロセスそのものに興味のある読者は他の研究報告書を参照されたい．

意志決定のための DEOS プロセスそのものを支援するためのソフトウェア・ツール群とは別に，DEOS プロセスにおける意志決定の結果を迅速にかつ的確にソフトウェアシステムに適用するソフトウェアの実行基盤が必要となる．本報告書ではそのようなソフトウェア実行基盤のうち，とくに重要な次の 2 点について基本的なコンセプトとその実装例について述べる．

- サービスの延命処置： DEOS プロセスによる意志決定の結果，何らかの変更がソフトウェアシステムに対して行われる．このようなソフトウェアの変更を外部からは透過的にシームレスに行われる必要がある．本報告書で述べる基盤ソフトウェアシステムでは，こうした変更をシームレスに行うことを可能としている．ソフトウェアに対する変更が外部からシームレスに行われることを強調して，この機能をサービスの延命処置，とりわけカーネルの延命処置と呼んでいる．
- フェイラの検知： DEOS プロセスによる意志決定を開始するためには，ソフトウェアの動作に広い意味での異常が生じていること，あるいは異常にいたる前の兆候を迅速かつ正確に捉える必要がある．期待した動作との統計的なズレを検知することにより，フェイラの前兆を広範に捉えられるようになっている．

図 3 にフェイラの検知とサービス延命処置との関係を示す．本報告書で述べる基盤ソフトウェアシステムでは，さまざまなモニタリングが可能となっており，そのモニタリング結果に統計的な解析を行うことで，期待したどうからのズレを早い段階で検知できるようになっている．モニタリング機構そのものはメカニズムとして提供されており，モニタリング結果から異常やその前兆を検知する仕組みはポリシーとして独立に提供できるようになっている．このモニタリング結果や統計処理の結果を DEOS プロセスに提供することにより，意志決定を早めその精度を高めることに貢献できるようになっている．

DEOS プロセスにより対処方法が決定されると，サービスの延命処置が行われる．この延命処置によって多少の犠牲を払うことはあっても，ソフトウェアシステムの状態は健全な状態に復帰し，サービスの継続が可能となる．

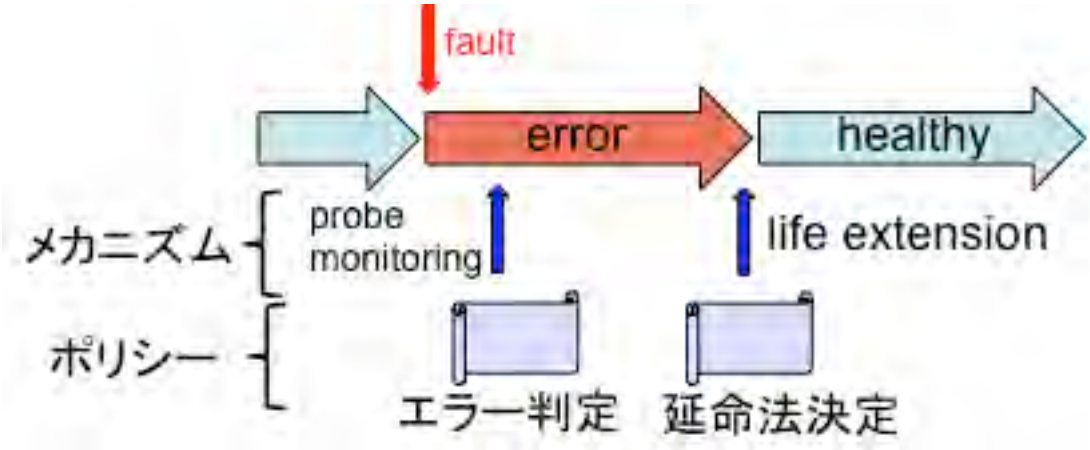

図 3 フェイラの検知とサービス延命処置

以下，本システムにおけるメカニズムとポリシーの分離について詳細に説明する．

2.1.2 モニタリング機構

本報告書で述べるモニタリング機構ではメカニズムとポリシーの分離を徹底した形で行っている．このモニタリング機構はエラーまたはフェイラを検知するための材料となるモニタリング情報を提供する機構である．実際にエラーが発生しているのか，フェイラが発生しているのかといった判断はこの機構の範囲外であり，すべてポリシーとして記述するようになっている．

このモニタリング機構はカーネル・レベルやアプリケーション・レベルなど，さまざまなレベルにおけるイベントの発生や処理にかかる時間などを計測する．本システムの特徴的な点は，従来のモニタリグシステムとは異なり，単に受動的にシステムの状態を観察するだけではないという点にある．モニタ

リングシステムの方から能動的に何らかのイベントを流し込み，そのイベントに対するシステムの反応を観察し，システムが適切に動作しているかどうかを検証できるようになっている．この機構を用いることによって，マルウェアへの感染といった通常のモニタリングだけでは発見しにくい事象も発見できるようになっている．

本基盤システムの持つモニタリング機構では，指定されたイベントを観察するための仕組み，指定されたイベントを能動的に注入する仕組みなどを提供しており，いつ，どのようなタイミングでどの程度詳細なモニタリングを行うのか，いつどのようなタイミングでイベントの注入を行うのかといった点はすべてポリシーとして記述する形になっている．このポリシーは宣言的な形で記述することができるだけではなく，プログラミング言語の形で複雑なポリシーをアルゴリズムとして記述できるようになっている．ポリシーそのものをプログラマブルな形で提供できるようにすることで，モニタリング結果をさまざまな形で組み合わせたり，複雑な統計処理を行ったりして，さまざまなエラーやフェイラが検知できるようになる．図 2 において，D-Script という名でディスクに格納されているものがこのポリシーに相当する．

このようにしておくことで，あらかじめ想定していたエラーやフェイラであれば，それを検知し，何らかの復旧作業を行うようなことが自動化できる．また，想定外のエラーやフェイラであれば，エラーやフェイラの発生しているモジュールを切り離しておくなどの緊急対策のみを自動的に行っておくようなことも可能である．

2.1.3 サービスの延命処置

サービスの延命処理についても同様に徹底したメカニズムとポリシーの分類を行っている．次の節で述べるように，サービスの延命処理とはサービス継続のために行うさまざまな処理を包含したものとなっており，エラーやフェイラの状況に応じて，DEOS プロセスにおける意志決定の結果を受けて実施されるものである．

延命処置の具体的な手法はすべてメカニズムとして提供されている．たとえば，システムの再起動を行うといった単純な障害回復手法もそのメカニズムのひとつとして提供されている．そのような延命手法のうちのどれを選択し，どのタイミングでそれを実行するかといった事柄はすべてポリシーとして記述されることを想定しており，モニタリグ機構と同様に D-Script としてプログラミング可能な形で記述できるようになっている．

なお，モニタリング機構やサービスの延命処置機構としてどのようなプリミティブを提供すべきかという点は十分に検討しておく必要がある．基盤システムの持つモニタリングや延命処置のためのプリミティブは DEOS プロセスにおける意志決定全体に影響を及ぼす場合がある．たとえば，ある種のフェイラ X は検知不能という前提で意志決定プロセス全体を設計する場合と，そのフェイラ X は検知可能という前提で意志決定プロセス全体を設計する場合では，両者に大きな違いが出てくる場合がある．延命処置機構についても同様であり，あるフェイラ Y が発生したとき，システムを停止する以外に選択肢がない場合と，同じフェイラ Y に対する何らかの対処法を持つ基盤システムとでは当然，上位の意志決定プロセスは異なってくる．

2.3. 延命可能なカーネル

延命可能なカーネルとは，オペレーティングシステム・カーネルに対して何からのエラー状態が検知された場合であっても，システムの停止時間を最小限に抑えつつカーネルを健全な状態に復帰させることをいう．エラー状態から健全な状態にシステムを復帰させることにより，サービスの継続が可能となり，システムの外部から見ると，一度，停止しかかったシステムに延命処置が施され，何事もなかったかのようにサービスが継続される．

カーネルの延命措置は旧来からあるさまざまな概念を統合し発展させたものである．たとえば，ソフトウェア・エージング(aging)[8]を未然に防止する手法であるソフトウェア・レジュベネーション(rejuvenation) [9,10]はカーネルの延命措置のひとつであるといえる．デバイスドライバの不具合から回復するための shadow driver [11] などもそのような手法のひとつである．

本報告書で述べる基盤システムでは既存の方式を組み合わせてカーネルの延命措置を可能にしているだけではなく，RootHammer [12,13], フェーズベース再起動 [14], カーネルの縮退動作などさまざまな新規機能を提供しており，多様な障害に対してより適切な延命措置が可能なソフトウェア基盤となっている． 4.2 節ではその具体例としてオペレーティングシステムの高速アップデート機能について紹介する．

2.4. フェイラおよびその予兆検知

フェイラやその予兆を検知することは障害発生を未然に防ぎ，障害発生時の原因究明においても重要である．フェイラやその予兆検知を行うためには，システムの振る舞いを多面的にモニタリングする必要がある．しかし，多くの情報をモニタリングしようとすると，測定のためのオーバヘッドが増大するだけでなく，監視対象のソフトウェアシステムの内部にプローブを差し込む必要があり，ソフトウェアシステムの大幅な改変が必要となるケースもある．

我々の取り組みでは，ソフトウェアシステムの外部から取得可能な情報のみから統計処理によってさまざまなモニタリングを可能としている．たとえば，管理図という管理工学で確立された手法を応用することによって，システムの応答時間のみから性能異常の兆候を検出し，さらには異常の原因となったコンポーネントの絞り込み支援まで可能としている．4.1 節ではサーバのログ情報のみから機械的な統計処理だけを用いて原因究明を行う手法を紹介する．

障害の予兆・検知システムはセキュリティ・インシデントも含め，システムの異常動作として捉えることを可能するように設計されている．基盤ソフトウェアがこのような機能を有することによってはじめてセキュリティ上の問題をフェイラとして捉え直すことができ，結果としてセキュリティのための特別な枠組みを用意せずともシステム全体のセキュリティを担保できる枠組みとなっている．障害の予兆検知システムは，オペレーティングシステムそのものの動作を検証するための機構を備えている．具体的にはオペレーティングシステムが実行するシステムコールと，それに呼応して行われるデバイスへのアクセスの相関を監視しており，正常に動作している場合との統計的なずれを認識することにより，オペレーティングシステムの動作異常（多くの場合はマルウェアへの感染）を検知することが可能になっている(図 4).

この仕組みを用いて，カーネルルートキット，キーロガー，ボットなど多くのマルウェアを一網打尽に検出可能となっている．

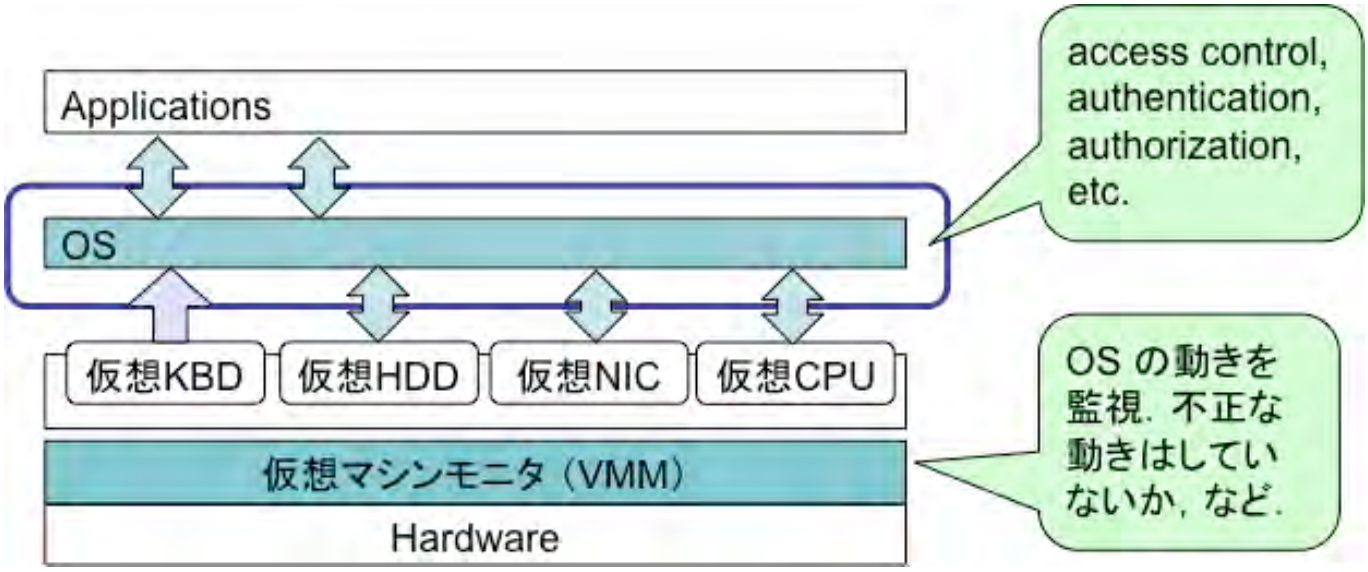

図 4 振る舞いの監視

2.5. 具体的な事例

ここまで述べてきた基本コンセプトを具体的に理解するため，簡単な事例を紹介する．メモリリークによるソフトウェア・エイジング [8] を検出しソフトウェア・レジュベネーション [9,10] を行う事例を考える．メモリリークによるソフトウェア・エイジングを検出するには，メモリの使用量を観察すればよい．モニタリング機構は定期的にメモリの使用量を報告する．これを受けて，メモリの使用量の変化を統計的に解析し，実際にメモリリークによるエイジングが発生していることを検出する．どのような統計処理を行って判定を行うのかという部分はポリシーとしてメカニズムから分離されていることに注意されたい．

モニタリング機構によりソフトウェア・エイジングが検出された時点で，近い将来にソフトウェア・レジュベネーションを行うことが決定される．これは D-Script により自動的に決定される場合もあるし，ソフトウェア・エイジングが起きているという報告を受けてより上位層で決定される場合もあろう．いずれにせよ，ソフトウェア・レジュベネーションを行うことと，そのタイミングがポリシーによって決定される．その決定を受け，実行時システムは指定されたタイミングで指定された方法によってソフトウェア・レジュベネーションを行う．

なお，ソフトウェア・エイジングとその対処法であるソフトウェア・レジュベネーションは産業界でも実用的に用いられている対策手法である．

3. 基本アーキテクチャ

3.1. 概要

本実行基盤を実現するために，D-System Monitor ならびに D-Visor を活用する．概略図を図 5 に示す．サービスを実行する環境を仮想マシンとして用意し，D-System Monitor および D-Visor がそれを監視しながらサービスを運用する．D-Visor はハードウェアを仮想化し，D-System Monitor および OS 間に Isolation を提供する．こうすることで，監視対象の OS とは明確に分離された監視環境を実現することができる．これにより，たとえ監視対象の OS がクラッシュした場合でも，D-System Monitor がその影響を受けることはない．また，監視対象の OS が攻撃者にのっとられたとしても，D-System Monitor を攻撃することが極めて難しくなる．仮に攻撃が進化して D-System Monitor を攻撃可能にな

ったとしても，D-System Monitor を監視する D-System Monitor を用意すればよく，攻撃の進化にも対応できるアーキテクチャとなっている．

D-System Monitor は D-Visor を通して，監視対象 OS が行う特権レジスタへのアクセスや，特権命令の実行，入出力やその内容などを正確に把握することができる．さらに，監視対象 OS に対して割り込みを注入したり，システムコールを起動するソフトウェア割り込みを注入したりすることで、それらに対する OS の動作を逐一観察できるようになる．このような手法を用いることで，OS が期待通りに動作していること，すなわち健全に動作していることを保証する．また，高速に OS を健全に戻すための仕組みも提供する．

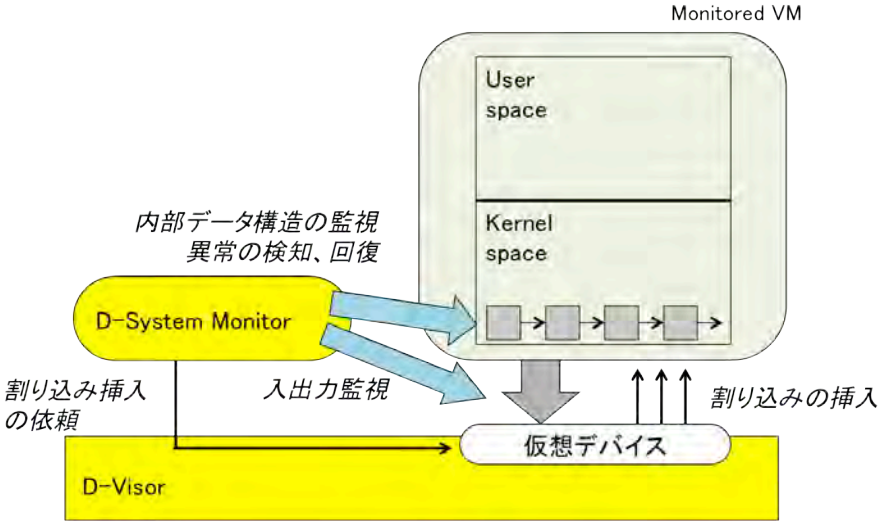

図 5 基本アーキテクチャ

3.2. D-Visor

D-Visor は仮想マシンモニタ(VMM)と同じ役割を果たし，サービスを稼働させるための仮想マシン，および D-System Monitor を稼働させる実行環境を与える．D-Visor は D-System Monitor が OS の状態を監視できるよう D-System Monitor サポート API を提供している．本 API をサポートすれば，D-System Monitor はどのような VMM 上でも稼働させることができる．つまり，環境に適した VMM を選択し，D-System Monitor を稼働できる．たとえば，組み込みシステム向けの VMM[15]，データセンタ向けの VMM[16]，デスクトップ向けの VMM[17]というように環境に適した VMM を利用して，OS を監視することができる．

また我々は，D-Visor を支援する機構についても研究開発を進めている．D-System Monitor を用いると仮想マシン間の性能分離が難しくなる．この問題を解決する Resource Cage 機構[18]を開発し，きめ細やかな資源管理を実現している．他にも D-Visor のソフトウェア・レジュベネーションを高速に実現する Roothammer[9,10]も研究開発している．D-Visor をソフトウェア・レジュベネーションする際には，その上で動作するすべての仮想マシンを停止しなければならなかったが，Roothammer を用いることで，仮想マシンを長時間停止することなく D-Visor をレジュベネートできるようになる．こうした機能の他に, D-Visor の開発を支援するモデルチェッキング手法[19]といったものについても研究開発を進めている．

他にも D-System Monitor を支援するための機能についても研究を進めている．我々は既存のセキュリティソフトウェアを D-System Monitor 内で稼働可能させることで，D-System Monitor の開発を容易にする VM Shadow 機構[20]も研究開発している．D-System Monitor 内に監視対象の OS が保有するプロセスリストやファイルシステムの view を構成することで，D-System Monitor 内で既存のセキュリティソフトウェアを稼働させることができる．

3.3. D-System Monitor

D-System Monitor は監視対象の VM を監視し，OS が健全に動作しているかを確認する．また，その OS の健全性を回復する機構を備えている．D-System Monitor サポート API を利用して，監視対象の VM 内の情報を取得して OS が期待通りに動作しているかを監視する．ここで取得する情報は入出力や OS 内のデータ構造といった情報である．こうした情報が通常のそれと異なる場合に OS 内で何か異常が生じていると判断できる．

D-Visor を利用して割り込みといったイベントを監視対象の VM に挿入することができる．たとえば，あるイベントに対する OS の反応を観察することで，OS が健全に動作しているかを把握することができる．我々は，キーボード入力とディスクの入出力との相関をとることで大部分のキーロガーを検知する機構やネットワーク入出力の相関からボットを検知する機構について研究開発を進めている．

他にも OS の健全性を高速に回復する機構についても研究開発を進めている．たとえば，OS の起動フェーズに着目した高速健全性回復機構であるフェーズベース再起動がある．これはスナップショット機能を利用して，高速に再起動と同等の効果を得る手法である[14]．これはソフトウェア・レジュベネーションを行う際のサービスのダウンタイム短縮に貢献するだけでなく，OS がクラッシュして停止した際にもそのダウンタイム短縮に貢献することができる．他にも再起動後の性能劣化を抑える方式についても研究開発している[21]．通常，OS を再起動すると OS が保有するキャッシュも同時に破棄されてしまう．キャッシュの中にはバッファキャッシュも含まれ，OS 再起動後にはアプリケーションのパフォーマンスが著しく低下してしまう場合がある．そこで，我々の方式では，D-Visor を利用して，OS が保有するバッファキャッシュを保存し，OS 再起動後にそれを再び割り当てる．こうすることで，再起動後もすぐにバッファキャッシュを構成でき，アプリケーションのパフォーマンス劣化を抑制することができる．

また，D-System Monitor のアップデートを高速に行う機構[22]についても研究開発を進めている．D-System Monitor のアップデート支援機構については 4 章で詳しく述べる．

上記のような D-System Monitor が保持している機能は D-Script によって実行される．D-System Monitor でモニタリングした情報は D-RE 内の D-Box 内にログとして保存される．ログから監視対象の OS がどのような状態にあるかを把握できる．D-System Monitor がとるべき行動は D-Script に記述されており，D-Script が実行する D-System Monitor の機能を選択していく．

4. 適用事例

4.1. 障害の原因究明支援

サーバシステムの障害の原因究明支援は重要である．特に性能異常はその原因究明が難しいことで知られている． 性能異常の原因を特定するための方法の一つに，過去に発生した性能異常と比較するという方法がある．これは，予期できないクライアントの増加やハードウェアの故障などにより類似した異常が繰り返し発生することが知られているためである．発生している性能異常が過去に発生したものと同じであると分かれば，既に分かっている対処法を適用できる．また，過去に原因究明が行われていない性能異常に対しても，原因究明を支援することができる．例えば，毎日同じ時間に同じ性能異常が発生していることやサーバアップデート時に同じ性能異常が発生していることが分かれば，解決法発見のために役立つ情報となる．

そのような着眼点から，性能異常を表現するために性能異常シグネチャを作成する[23]．性能異常シグネチャとは性能異常の特徴を記述したものである．性能異常シグネチャを作成することができれば， 異常発生時，過去の性能異常シグネチャの中に現在の状態と同じものがあるかどうかを調べ，同じシグネチャが存在する場合，既に分かっている対処法を適用することができる．

ここで紹介する手法はログの出現パターンから性能異常シグネチャを作成する手法である．ログを用いる利点は次の通りである．ログには開発者が有用だと判断した情報が出力されるため，シグネチャを作成するのに適している．また，ログはアプリケーションが同じであれば実行環境によらず類似した内容となるため，同じアプリケーションを使っているユーザ間でシグネチャを共有しやすくなる．

ログの出現パターンは各ログの出現回数とする．これは異常発生時には正常時と比べて各ログの出現回数が大きく変わることが知られているためである．ログの出現パターンはベクトル形式で表現し，各列がメッセージの種類に対応し，その成分が各メッセージの出現比率を表す．性能異常シグネチャは性能異常ごとにログの出現パターンをグループ化したものであり，ニューラルネットワークを誤差逆伝播法で訓練することで実現する．学習時にはニューラルネットワークにログの出現パターンと発生している異常の種類を入力する．そして，利用時にはログの出現パターンを入力すると発生している性能異常を出力する．

このようにして実現した性能異常シグネチャを用いて実験を行ったところ 90% 以上の精度で性能異常を分類することができ，原因究明支援に有効であることがわかる．

4.2. D-System Monitor の高速アップデート

ソフトウェアをアップデートする際には OS の再起動を伴う場合が多くある．OS をアップデートするときには, 古い OS を一旦停止して, その後新しい OS の起動イメージを使って OS を稼働させる．また，アプリケーションをアップデートする際にも OS の再起動を利用することがある．ライブラリや共有コンポーネントをアップデートする際には，それらを利用するアプリケーションをすべて再起動する必要があるため，OS の再起動を用いてすべてのアプリケーションを再起動させることが一般的に用いられている．

OS の再起動はダウンタイムを伴うため，アップデートをする際にはサービスの停止を余儀なくされてしまう．アップデートには新たな機能の追加やセキュリティホールの修正といった重要な項目が含まれているため，アップデートが報告されたらすぐに適用することが求められる．しかし，アップデートの適用はサービスの一時的な停止に繋がるため，すぐに適用することが難しい．結果として，アップデート適用までの間は新たな機能を使えず，最悪の場合，攻撃者に報告されたセキュリティホールを突かれて深刻な被害に及ぶ可能性がある．また，D-System Monitor のような監視機能のダウンタイムは，その間監視機構がストップすることを意味し，ソフトウェアをアップデートしている最中にサービスの停止や攻撃を受ける可能性が生じてしまう．

そこで，本実行基盤はソフトウェアをアップデートする際のダウンタイムを短縮する ShadowReboot 方式[22]を提供する．ShadowReboot では，稼働しているアプリケーションから OS 再起動の動作を隠蔽する. 具体的には, OS の再起動を行う際, 同じ状態を持つ VM を複製し, そちらの VM 上で OS 再起動を実行する．複製の元となった VM では引き続きアプリケーションを稼働し続けられる．再起動修了後に，VM のスナップショットを取得し，そのスナップショットを復元することでソフトウェアのアップデートを実現する．仮想ディスクの更新内容をスナップショット復元時にも保持し続けることで，通常のアップデートを短いダウンタイムで行うことができる．現在のところ，プロトタイプ実装において，アップデートに伴うダウンタイムを 83 〜 98 % 削減できることがわかっている[22].

5. おわりに

OS はアプリケーションやハードウェアの進化に伴って，肥大化・複雑化を繰り返してきた．その結果，OS には高い信頼性が求められているにもかかわらず，OS 内にはソフトウェアバグが存在したり OS 自身をのっとる攻撃が登場したりといった報告がなされている．結果として，これらの影響でサービスが停止したり，最悪の場合，サービスの改竄や情報漏洩が起きてしまう．本報告書では，OS の障害を前提とした，サービス延命を可能とする実行基盤について述べた．本実行基盤を用いると，障害な

どが起きた際にカーネルに対して延命措置を施すことで，何事もなかったかのようにサービスを継続することができる．本実行基盤の特徴は，ソフトウェアバグによる障害やネットワークからの OS を乗っ取る攻撃などを統一的に扱うことができる点にある．このような統一的な取り扱いにより，障害対策やセキュリティ対策のための機構を個別に作り込む必要がなくなり，共通の基盤として提供可能となっている．本実行基盤は，D-RE の構成要素である D-System Monitor および D-Visor を駆使することで実現している．また，本報告書では実行基盤を支える技術である，障害の原因究明支援手法，ならびに D-System Monitor の高速アップデート手法について具体的に述べた．この 2 つは ET2012 内でデモを行っている．

参考文献

[1] M. Tokoro (ed), Open Systems Dependability – Dependability Engineering for Ever-Changing Systems, CRC Press, 2012.
[2] N. Palix, G. Thomas, S. Saha, C. Calves, J. Lawall, and G. Muller, “Faults in Linux: Ten Years Later,” in Proceedings of the ACM International Conference on Architectural Support for Programming Languages and Operating Systems, 2011, pp.305—318.

[3] Agobot. http://www.f-secure.com/v-descs/agobot.shtml

[4] G. Klein, K. Elphinstone, G. Heiser, J. Andronick, D. Cock, P. Derrin, D. Elkaduwe, K. Engelhardt, M. Norrish, R. Kolanski, T. Sewell, H, Tuch, and S. Winwood, “seL4: Formal Verification of an OS kernel,” in Proceedings of the ACM Symposium on Operating Systems Principles, 2009, pp. 207—220.

[5] X. Jiang, X. Wang, and D. Xu, “Stealthy Malware Detection Through VMM-based “Out-of-the Box” Semantic View Reconstruction,” in Proceedings of the ACM Conference on Computer and Communications Security, 2007, pp.128—138.

[6] A. Seshadri, M. Luk, N. Qu, and A. Perrig, “SecVisor: A Tiny Hypervisor to Provide Lifetime Kernel Code Integrity for Commodity OSes,” in Proceedings of the ACM Symposium on Operating Systems Principles, 2009, pp.335—350.

[7] Fides: Selectively Hardening Software Application Components against Kernel-level or Process-level Malware, in Proceedings of the ACM Conference on Computer and Communications Security, 2012.

[8] D. L. Parnas, “Software Aging”, Proc. of the 16th International Conference on Software Engineering, 1994, pp.279—287.

[9] Y. Huang, C. Kintala, N. Kolettis, and N. D. Fulton, “Software Rejuvenation: Analysis, Module and Applications,” in Proceedings of the International Symposium on Fault-Tolerant Computing, 1995, pp.381—390.

[10] V. Castelli, R. E. Harper, P. Heidelberger, S. W. Hunter, K. S. Trivedi, K. Vaidyanathan, and W. P. Zeggert, “Proactive Management of Software Aging,” in IBM Journal of Research and Development, Vol. 45, No. 2, pp.311—332, 2001.

[11] M. M. Swift, M. Annamalai, B. N. Bershad, and H. M. Levy, “Recoverying Device Drivers,” in Proc. of the USENIX Symposium on Operating System Design and Implementation, 2004, pp.1—16

[12] K. Kourai and S. Chiba, “A Fast Rejuvenation Technique for Server Consolidation with Virtual Machines,” in Proc. of the International Conference on Dependable Systems and Networks, 2007, pp.245—255.

[13] K. Kourai and S. Chiba, “Fast Software Rejuvenation of Virtual Machine Monitors,” IEEE Transaction on Dependable and Secure Computing, Vol. 8, No. 6, pp.839—851, 2011.

[14] Y. Yamakita, H. Yamada, and K. Kono, “Phase-based Reboot: Reusing Operating System Execution Phases for Cheap Reboot-based Recovery,” in Proc. of the International Conference on Dependable Systems and Networks, 2011, pp.169—180.

[15] T. Nakajima, Y. Kinebuchi, H. Shimada, and A. Courbot, T. Lin, “Temporal and Spatial Isolation in a Virtualization Layer for Multi-core Processor based Information Appliances,”, in Proceedings of the Asia and South Pacific Design Automation Conference, 2011, pp.645—652.

[16] P. Barham, B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, R. Neugebauer, I. Pratt, and A. Warfield, “Xen and the Art of Virtualization,” in Proc. of the ACM Symposium on Operating Systems Principles, 2003, pp.164—177.

[17] VMware workstation, http://www.vmware.com

[18] 新井昇鎬, 光来健一, 千葉滋, “仮想マシンを用いた IDS オフロードにおける CPU 資源管理”, 第 114 回 OS 研究会研究報告，2010

[19] Shuichi Oikawa, “Enforcing Dependable Operations by Model Checking a Virtulization Layer,” in Proceedings of the International Workshop on Open Systems Dependability, 2011, pp.254—256

[20] 飯田貴大, 光来健一 “VM Shadow: 既存 IDS をオフロードするための実行環境”, 第 119 回 OS 研究会研究報告，2011,

[21] K. Kourai, “Fast and Correct Performance Recovery of Operating Systems Using a Virtual Machine Monitor”, in Proceedings of the ACM International Conf. on Virtual Execution Environments, 2011, pp.99—110.

[22] H. Yamada, and K. Kono, “Traveling Forward in Time to Newer Operating Systems using ShadowReboot,” in Proc. of the ACM Asia-Pacific Workshop on Systems, 2011, pp.12:1—pp.12:5

[23] 高橋宏明，岩田聡，山田浩史，河野健二，“ログを利用した性能異常の分類手法”，日本ソフトウェア科学会第 29 回大会

「実用化を目指した組込みシステム用ディペンダブル・オペレーティングシステム」（DEOS プロジェクト）は科学技術振興機構（JST）の戦略的創造研究推進事業 CREST の研究領域のひとつです。

DEOS プロジェクト