特 別 研 究 報 告 書

題 目

# TwinOSにおけるヘルスチェック機能の実現と評価

指導教員

報 告 者

長田 一帆

岡山大学工学部 情報工学科

平成22年2月5日 提出

# 要約

近年，計算機に対する不正な攻撃が増加している．その対策として，不正な攻撃を OS によって監視する手法が多く提案されている．しかし，計算機への不正侵入を許し OS 自体が乗っ取られてしまった場合，OS が改ざんされ，システムの信頼性が大きく低下してしまう．これを防ぐには，攻撃対象となる計算機を外部から監視する必要がある．

一方，計算機ハードウェアの性能向上に伴い，一台の計算機上に複数の OS を走行させ，その上で多様なサービスを実現したいという要求がある．この要求を満たす方式として，一台の計算機上に 2 つの Linux を独立に走行させる TwinOS が提案され，実装されている．

そこで，TwinOS を用いたヘルスチェック機能が提案され，設計されている．ヘルスチェック機能とは，ヘルスチェック機能は攻撃対象となる OS から独立して動作するセキュリティシステムである．先行 OS を監視用 OS，共存 OS をサービス用 OS とし，監視用 OS にヘルスチェック機能を実装する．これにより，先行 OS から共存 OS のメモリ領域を操作できるが，共存 OS からは，先行 OS が使用するメモリにアクセスできない．これを利用することで，サービス用 OS の検査データの授受において，OS 間通信を必要としない構成が可能である．この手法により，監視処理を外部から隠ぺいできる．つまり，先行 OS から共存 OS を監視することで信頼性の高いシステムの構築が可能になる．

本論文では，TwinOS におけるヘルスチェック機能の実装と評価を行った．このヘルスチェック機能では，7 つの機能を実装した．これらの機能を組み合わせて，OS の検査を行う．具体的には，カーネルと LKM を検査する．異常が発見されれば，異常が発見された OS を直ちに停止させる．評価では，検査する処理のオーバヘッドの測定と，測定結果の考察を行った．残された課題として，検査の常駐化と実際のネットワークを介した攻撃に対して，ヘルスチェック機能が有効であることの証明がある．

# 目次

# 図 目 次

# 表 目 次

# 第 1 章

# はじめに

近年，インターネットの普及により，ネットワークを介した攻撃が増加している．また，攻撃方法自体も単純な PC 環境の破壊目的からスパイウェアを用いた個人情報の収集や迷惑メール送信のように，営利目的の攻撃へ変わってきた．これらの攻撃は，OS 自体を改ざんし，管理者が容易に検出できないような攻撃方法をとる．このため，OS のセキュリティ向上への要求が高まっており，様々なシステムの研究がされている．しかし，既存のシステムの多くは攻撃対象となる OS 上に実装されている．このため，OS が乗っ取られた場合，システムが無効化される危険があり，信頼性の低下につながるという問題がある．

そこで，TwinOS を用いたヘルスチェック機能が提案されている．ヘルスチェック機能は攻撃対象となる OS から独立して動作するセキュリティシステムである．たとえ OS が乗っ取られた場合でもシステムが無効化される危険が少ない．

本論文では，TwinOS におけるヘルスチェック機能の実装についての課題とその対処を述べる．具体的には，課題として，不正な LKM のロードへの対処と LKM のロードによるカーネルテキスト部の変化への対処があることを示す．次に，TwinOS を用いた場合の，ヘルスチェック機能と OS との関係について述べる．最後に，ヘルスチェック機能の性能の評価について述べる．

# 第 2 章

# TwinOS

## 2.1 特徴と構成

1 台の計算機上で 2 つの OS を同時に走行させる TwinOS 方式が提案されている [1]．現在，TwinOS は，以下の設計方針を基に 2 つの Linux2.4 カーネルに TwinOS 方式を適用，実現されている．

**(1)** 相互に他 OS の処理負荷の影響を受けない．

**(2)** 両 OS とも入出力性能を十分に利用できる．

このため，１台の計算機ハードウェアにおいて，プロセッサやメモリ，および入出力機器といった各資源の効果的な共有と占有が必要である．２つの OS の独立性を保つために，共有するハードウェアを最小限とすることが有効であるため，プロセッサのみを共有させる．プロセッサ以外のハードウェアは分割し，分割したそれぞれを各 OS に占有させる．分割するハードウェアのうち，入出力機器は，OS の起動時に指定したものだけを占有させる．各ハードウェアの分割と共有方法を以下に示し，図 2.1 に TwinOS の構成を示す．

**プロセッサ**

時分割することによって共有する．このため，OS の起動処理は順番に行い，走行中は，タイマ割込みを契機に走行 OS を切替える．

**メモリ**

若番と老番に 2 分割する．そして，先に起動する OS(以降，先行 OS と呼ぶ) にメモリの老番アドレス空間を，後から起動する OS(以降，共存 OS と呼ぶ) に若番アドレス空間を割当てる．

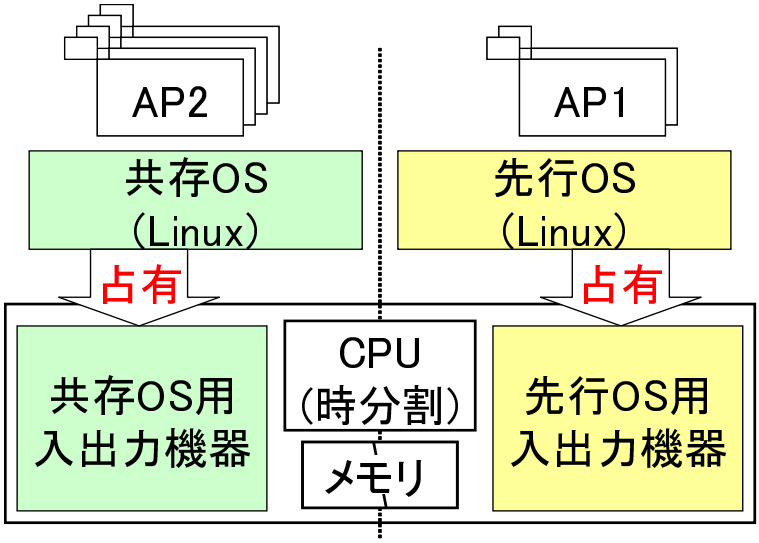


図 2.1 TwinOS の構成

**入出力機器**

各 OS 毎に指定された入出力機器のみを占有制御する．割り込み/例外処理は，当該割り込みを発生した入出力機器を占有制御している OS の割り込み処理が呼び出されるように制御する．このため，走行していない OS が占有する入出力機器からの割り込みの場合は，OS を切り替える．

## 2.2 TwinOS の起動

### 2.2.1 起動に用いるシステムコール

TwinOS が共存 OS の起動のために提供するシステムコールについて述べ，これらシステムコールを用いた TwinOS 起動の流れについて示す．

(1)netos

netos システムコールを発行すると，先行 OS は指定された共存 OS の圧縮カーネルイメージをメモリ上に展開する．この状態では共存 OS は動作しておらず，共存 OS のカーネルイメージがメモリ上に存在するだけである．

(2)os2_trace_os1

os2_trace_os1 システムコールを発行すると，先行 OS はプロセッサのトレースモードを利用して，先行 OS の監視のもとで共存 OS を起動する．先行 OS は，共存 OS の起動処理によって走行環境を破壊しないように，共存 OS を起動する．先行 OS の走行環境を破壊する恐れのある命令は，先行 OS がエミュレートし，実行されないようにする．

### 2.2.2 起動方式

TwinOS の起動方式を図 2.2 に示し，以下で詳細を述べる．TwinOS は以下の順番で起動する．

(1) 先行 OS を起動

(2) 共存 OS のカーネルをメモリ上に展開

先行 OS は，圧縮された共存 OS のカーネルをメモリ上に展開し，先行 OS の走行環境を保存する．その後，共存 OS セットアップルーチンを起動し，メモリ上にある圧縮された共存 OS カーネルイメージを若番アドレス上に展開する．カーネルイメージの展開後，先行 OS に処理を戻し，走行環境を復元する．

(3) 共存 OS の起動

ユーザプログラムは os2_trace_os1 システムコールを発行し，トレースモードを利用し，共存 OS を起動する．ここでは，ページテーブルの初期化，CPU タイプのチェック，および 4MB ページングに切り替えなどを行う．図 2.2 の (A),(B),(C) の処理により，カーネルテキスト部が変更される．以下に各処理の詳細について述べる．

(A) ページテーブルの初期化

起動処理で一時的に使用するページテーブルを初期化する．

(B) CPU タイプのチェック

CPU の種類を確認し，拡張機能を利用するかチェックする．

(C) 4MB ページングに切り替え

CPU が 4MB ページングをサポートしている場合，ページディレクトリを 4MB ページングに初期化する．

(4) 共存 OS の init 処理

バスの初期化などを行う．図 2.2 の (D) の処理により，カーネルテキスト部が変更される．

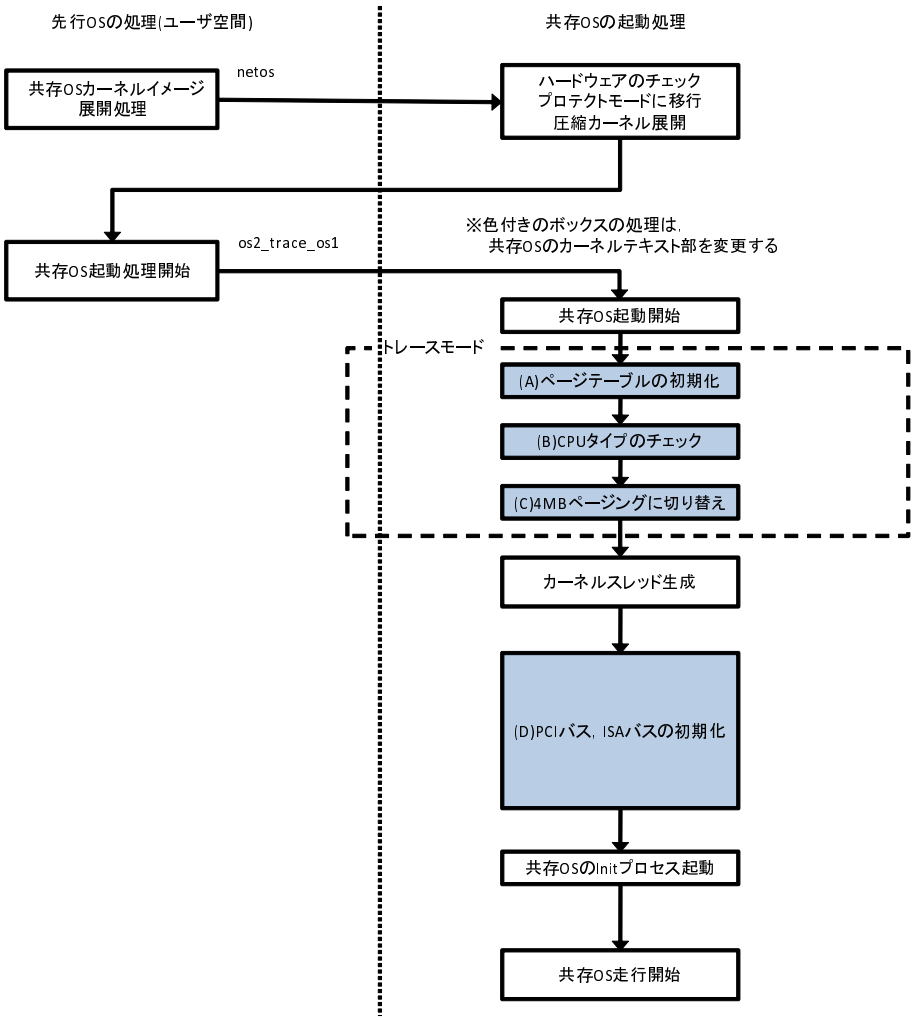


図 2.2 TwinOS 起動の流れ

(5) 共存状態へ移行

先行 OS と共存 OS が独立動作し，共存状態になる．

# 第 3 章

# TwinOSを用いたヘルスチェック機能

## 3.1　従来のヘルスチェック機能

### 3.1.1　ヘルスチェック機能の目的

ヘルスチェック機能とは，OS を検査し異常を検出するための機能であり，攻撃者による不正な改ざんの検出と被害の拡大防止を目的にしている．ヘルスチェック機能を攻撃対象となる OS 上に実装すると，OS が改ざんされた時のシステムの信頼性が低下する．このため，ヘルスチェック機能の実現は，攻撃対象となる OS から分離することが重要である．つまり，攻撃対象ではない OS 上にヘルスチェック機能を実装し，攻撃対象となる OS を監視する必要がある．これまでの設計と実装により，TwinOS の先行 OS を監視用 OS，共存 OS をサービス用 OS とすることで，ヘルスチェック機能は攻撃対象となる OS から分離することに成功している [2][3].

### 3.1.2　実装されている機能

従来のヘルスチェック機能では，以下の機能が実装されている．

(1) 正常データ作成機能

(2) 正常起動検査機能

(3) 正常走行検査機能

(4) 共存 OS 停止機能

(5) 共存 OS 再開機能

上記の各機能の詳細は，3.5 節で示す.

### 3.1.3 従来のヘルスチェック機能の処理の流れ

従来のヘルスチェック機能の処理の流れを図 3.1 と図 3.2 に載せ，以下で説明する.

**従来の正常データ作成機能**

(1) 共存 OS のカーネルイメージ展開する

netos システムコールを発行し，共存 OS のカーネルイメージをメモリ上に展開する.

(2) カーネルテキスト部のハッシュ値を取得する.

検査範囲であるカーネルテキスト部のハッシュ値を取得する．検査範囲については 3.1.4 項で説明する.

(3) ハッシュ値を正常データとする

(2) で取得したハッシュ値を正常データとする.

(4) 共存 OS 起動処理を開始する

os2_trace_os1 システムコールを発行し，共存 OS の起動処理を開始する.

(5) カーネルテキスト部のハッシュ値を取得する

共存 OS 起動処理の終了後にカーネルテキスト部のハッシュ値を取得する.

(6) ハッシュ値を正常データとする

(5) で取得したハッシュ値を正常データとする.

(7) カーネルテキスト部のハッシュ値を取得し続ける

共存 OS がバスの初期化を行う時に，共存 OS のカーネルテキスト部が変更される．この変更をトラップするために，共存 OS のカーネルテキスト部が変更されるまで取得処理を実行し続ける.

(8) ハッシュ値と正常データとする

(7) で取得したハッシュ値と正常データとする.

(9) 結果を表示

正常データの作成が正常に終了したことをユーザに通知する.

### 従来の検査機能

(1) 正常データが存在するかを調べる

正常データが存在していれば次の処理を実行する．正常データが存在していなければこの機能を終了する．

(2) 共存 OS のカーネルイメージ展開する

netos システムコールを発行し，共存 OS のカーネルイメージをメモリ上に展開する．

(3) カーネルテキスト部のハッシュ値を取得する．

検査範囲であるカーネルテキスト部のハッシュ値を取得する．検査範囲については 3.1.4 項で説明する．

(4) ハッシュ値と正常データを比較する

(3) で取得したハッシュ値と正常データを比較する．値が同じであれば次の処理を行う．値が異なれば異常終了する．

(5) 共存 OS 起動処理を開始する

os2_trace_os1 システムコールを発行し，共存 OS の起動処理を開始する．

(6) カーネルテキスト部のハッシュ値を取得する

共存 OS 起動処理の終了後にカーネルテキスト部のハッシュ値を取得する．

(7) ハッシュ値と正常データを比較する

(6) で取得したハッシュ値と正常データを比較する．値が同じであれば次の処理を行う．値が異なれば異常終了する．

(8) カーネルテキスト部のハッシュ値を取得し続ける

共存 OS がバスの初期化を行う時に，共存 OS のカーネルテキスト部が変更される．この変更をトラップするために，共存 OS のカーネルテキスト部が変更されるまで取得処理を実行し続ける．

(9) ハッシュ値と正常データを比較する

(8) で取得したハッシュ値と正常データを比較する．値が同じであれば次の処理を行う．値が異なれば異常終了する．

(10) 結果を表示

検査が正常に終了したことをユーザに通知する．

### 3.1.4 従来の検査範囲

従来のヘルスチェック機能は，カーネルの検査のみを行っていた．具体的には，メモリ上の以下の 2 つの領域を対象とした検査を行う．これらの領域から取得したデータを検査データとして用いるまた，監視対象は OS のため，検査データの基準となるデータ (以降，正常データ) が必要である．共存 OS が安全な状態で取得したデータを正常データとする．

(1) カーネルテキスト部

通常，プログラムはテキスト部を超えて実行することはない．そこで，攻撃者は悪意あるコードを実行するため，テキスト領域上への悪意あるコードの挿入，データ部に配置した悪意あるコードのアドレスへ，ジャンプ先アドレスを変更するといった方法をとる．

(2) システムコールテーブル

システムコールテーブルは，システムコールのジャンプ先アドレスを示すテーブルである．上記と同じ理由で，ジャンプ先アドレスの変更のために用いられる．

通常，上記 2 つの領域は変更されることがないため，検査はデータの比較により行う．変更が発見された場合，カーネルが改ざんされているため，異常としている．

## 3.2 ヘルスチェック機能の課題

従来のヘルスチェック機能では，カーネルの検査のみを行っていたため，LKM に対処することができなかった．ヘルスチェック機能実現における課題を以下に示す．

(課題 1) 不正な LKM のロードへの対処

OS への攻撃の一つに，不正な LKM のロードがある．不正な LKM をカーネルに組み込むと，カーネルテキスト部が変更されないまま OS が改ざんされる．よって，LKM に対処するために，ロードされる LKM 全てに対して，検査を行う必要がある．また，ロードされた LKM の全てのデータを検査範囲とすることはできない．LKM のデータにはテキスト部とデータ部がある．データ部はロードされる度に内容が変わる．このため，LKM のテキスト部のみを検査対象とする必要がある．

(課題 2) LKM のロードによるカーネルテキスト部の変更

LKM をロードする際，ページテーブルに空き領域がない場合が発生することがある．このとき，マッピングのためにページテーブルが新規に作成され，それによりカーネルテキスト部が変更される．この変更以前のデータを正常データとしていた場合，正常な LKM をロードしていても OS の異常と判断されるため，対処する必要がある．

## 3.3 対処

### 3.3.1 不正な LKM のロードへの対処

不正な LKM による OS の改ざんに対処するために,「正常 LKM 登録機能」と「正常 LKM 検査機能」を実装した．これらの機能の処理の流れついては，3.5 節で詳しく説明する．以下で，これらの機能の設計と実現方法について述べる．

(設計 1) 共存 OS の LKM のロードを先行 OS 側が認識できる

検査する契機となる共存 OS の LKM のロードを，先行 OS 側が認識できる必要がある．

(設計 2) 検査対象となる LKM のテキスト部のみを検査範囲とする

物理メモリに展開された LKM のデータの全てを検査範囲とすることはできない．展開されたデータには，データ部も含まれている．データ部はロードする度に変更されるため，テキスト部だけを検査対象とする必要がある．

これらの設計の実現方法を以下で示し，説明する．

(1) 共存 OS の LKM のロードを先行 OS が認識できる

この要求は，共存 OS の特定のページテーブルを監視することで対処する．特定のページテーブルとは，図 3.3 の 0x1C88 から 0x1C8B の大きさ 4byte のエントリを先頭アドレスとするページテーブルである．LKM をロードすると，このページテーブルに LKM の展開先アドレスが追加される．ページテーブルの変更を契機として，先行 OS 側が LKM のロードを知ることができる．

(2) 検査対象となる LKM のテキスト部を検査範囲とする

LKM をロードすると，LKM が物理メモリに展開される．この展開された LKM のデータには，ヘッダとテキスト部とデータ部が含まれる．物理メモリに展開された LKM のデータ構造を，図 3.4 に載せる．図 3.4 のテキスト部だけを検査範囲とする．

**検査範囲**

物理メモリに展開された LKM を検査データとして，このデータに含まれているテキスト部を検査範囲とした．

**検査データを取得する処理の流れ**

共存 OS のページディレクトリの一部を図 3.3 に示し，以下で説明する．変更されるページテーブルは，カーネルテキスト部にある,「swapper_pg_dir」のエントリにあるアドレスを先頭アドレスとして配置されている．このページテーブルを監視することにより，LKM のロードと LKM のアンロードを監視できる．しかし，ページテーブルを検査対象としても，特定の LKM に対して固有のハッシュ値を取得できない．これは，ページテーブルに LKM の展開先アドレスが追加されるが，追加される場所はランダムなためである．ここで，ハッシュ値算出の流れを図 3.5 に示し，以下で説明する．ページテーブルに追加された LKM 展開先アドレスから物理アドレス上位 20bit を算出し，物理メモリのページを取得する．取得したページを結合し，一つのデータとして変数に格納する．これを検査データとする．

### 3.3.2 LKM のロードによるカーネルテキスト部の変更への対処

LKM のロードが終了した後にカーネルテキスト部のハッシュ値を取得し，もし変更されていれば，正常データとして登録する．

## 3.4 LKM に対応した検査処理

### 3.4.1 検査処理に用いるシステムコール

TwinOS におけるヘルスチェック機能の実現には，カーネル空間での処理が必要である．このため，ヘルスチェック機能を支援するシステムコールが実装されている．本章では，TwinOS におけるヘルスチェック機能のシステムコールについて説明する．システムコールの一覧を表 3.1 に示し，システムコールの詳細について以下で説明する．

(1) netos システムコール

**【形式】** `int sys_netos(const void *bzImage, unsigned int size)`

**【引数】**

表 3.1 システムコール一覧

| システムコール名 | |
|---|---|
| netos | 共存 OS のカーネルイメージをメモリに展開する. |
| os2_trace_os1 | 展開したカーネルイメージをトレースモードで起動する. |
| get_hash | ハッシュ値を算出する. |
| suspend_os | 共存 OS の走行を停止させる. |
| restart_os | 共存 OS の走行を再開させる. |

bzImage：bzImage を展開する場所の先頭アドレス.
size：bzImage のサイズ

**【機能】**

アドレス bzImage から始まる size の大きさの領域にカーネルイメージを展開するシステムコールである.

(2) os2_trace_os1 システムコール

**【形式】**`int sys_os2_trace_os1(void)`

**【引数】**なし.

**【機能】**

先行 OS を破壊する命令をエミュレートしながら，共存 OS を起動するシステムコールである.

(3) get_hash システムコール

**【形式】**
```
int sys_get_hash(int format, unsigned char* start,
                 int length, unsigned char* buffer)
```

**【引数】**

format：buffer に格納する値の形式
(0：生データ 1,2：ハッシュ値)
start ：領域の開始アドレス
length：領域のサイズ
buffer：ハッシュ値の格納先アドレス

**【機能】**

start と length で指定した領域のハッシュ値を取得する．format が 1 の場合，ハッシュ関数を用いてハッシュ値を算出する．format が 2 の場合，MD5 アルゴリズムを用いてハッシュ値を算出する．

(4) suspend_os システムコール

**【形式】** `void sys_suspend_os(void)`

**【引数】** 引数なし

**【機能】**

共存 OS の走行を停止させるシステムコールである．現在のヘルスチェックでは，共存 OS の起動途中に自動でこのシステムコールが発行される仕様となっている．図 3.6 に suspend_os システムコールの処理の流れを示す．

(5) restart_os システムコール

**【形式】** `void sys_restart_os(void)`

**【引数】** 引数なし

**【機能】**

共存 OS の走行を再開させるシステムコールである．(2) で停止した共存 OS を再開する時に手動で発行する．図 3.7 に restart_os システムコールの処理の流れを示す．

### 3.4.2 LKM に対応した検査処理全体の処理の流れ

LKM に対応した検査処理全体の処理の流れを図 3.8 に載せ，以下で説明する．LKM のロードはカーネルの init プロセス後に行われるため，LKM に対応した検査処理はカーネル検査終了後すぐに行う．

(1) 正常 LKM データが存在するかを調べる

正常 LKM データが存在していれば次の処理を実行する．正常 LKM データが存在していなければこの機能を終了する．正常 LKM データとは，安全な LKM のハッシュ値を保存したデータである．

(2) ページテーブルの生データを取得する

get_hash システムコールを発行し，ページテーブルのバイナリデータを取得する．

(3) 全ての LKM をロードするための処理待ちを行う

sleep 命令を用いて 10 秒間プロセスを停止し，LKM がロードされるまで待つ．

(4) ページテーブルの生データを取得する

再度 get_hash システムコールを発行し，ページテーブルのバイナリデータを取得する．

(5) 全ての LKM がロードされているかの判定を行う

(2) で取得したページテーブルと (4) で取得したページテーブルを比較する．(5) の処理を初めて実行された時は，比較結果にかかわらず再度 (3) と (4) の処理を行う．つぎに，(5) の処理の実行が 2 回目以降である時の場合を考える．比較結果で値が同一であれば，次の処理を実行する．値が異なっていれば，再度 (3) と (4) の処理を行う．

(6) LKM のハッシュ値を取得する

LKM のテキスト部のハッシュ値を取得する．ハッシュ値は 1 つの LKM に対して 1 つ算出される．

(7) 取得したハッシュ値と正常データを比較する

(7) で取得したハッシュ値と正常データを比較する．値が同じであれば次の処理を行う．値が異なれば異常終了する．

(8) 結果を返却する

ユーザに結果を表示し，機能を終了する．

## 3.5 各機能の実装

### 3.5.1 正常データ作成機能

正常データ作成機能と正常走行検査機能は，基本的に同じ処理を行う．これら機能の処理の違いは，取得したデータを「正常データとして扱うか」,「検査データとして扱うか」の処理の違いである．図 3.1 に正常データ作成機能の処理の流れを載せ，以下で説明する．正常データ作成機能の起動後に，共存 OS の起動処理が開始する．このため，図 3.1 は共存 OS の

起動の流れと対応づけている．正常データ作成機能では，2.2.2 項で述べたカーネルテキスト部の変更のタイミングにおいて，ハッシュ値を算出する．このハッシュ値を正常データとする．ユーザ空間からカーネル空間のメモリにアクセスすることはできないため，ハッシュ値の取得には，get_hash システムコールによる支援が必要となる．

### 3.5.2 正常起動検査機能

共存 OS を起動処理に合わせて，共存 OS が正常に起動するかを検査する機能である．ただし，検査の処理自体は正常走行検査機能と同じであるため，説明を省略する．ここでは，起動中に行う検査の流れについて述べる．図 3.2 に処理の流れを載せる．正常データ作成機能のハッシュ値算出のタイミングと同じタイミングで，ハッシュ値を算出し，正常データと比較する．算出されたハッシュ値が正常データと異なる場合エラーを出力し，この機能の処理を終了する．

os2_trace_os1 を発行してトレースモードの実行が終了した後は，共存 OS は先行 OS の管理から離れ，独立して動作する．比較処理の間に OS が改ざんされるのを防ぐため，トレースモードの実行が終了するとき，走行を一時停止させた状態で起動するように os2_trace_os1 を変更する．このため，OS による支援には os2_trace_os1 の機能変更がある．os2_trace_os1 発行後は，共存 OS 再開を用いて，任意の契機で共存 OS の走行を再開させる．

### 3.5.3 正常走行検査機能

共存 OS 起動後に検査を行う機能である．処理流れを図 3.9 に示し，以下に説明する．

(1) ハードディスクまたはメモリから，正常データを取得する．

(2) 共存 OS カーネルメモリ空間からデータを取得する．

(3) 正常データと検査データを比較する．

(4) 結果を判定する．

ユーザ空間からカーネル空間のメモリへアクセスすることはできないため，(2) の「共存 OS カーネルメモリ空間からデータを取得」処理を get_hash システムコールにより支援する必要がある．その他処理はユーザ空間で実現可能であるため，ユーザ空間で実現する．

### 3.5.4 共存 OS 停止，再開機能

共存 OS 停止機能と共存 OS 再開機能は，それぞれ停止処理と再開処理を呼び出す．これらの機能は単純に共存 OS を停止，再開処理を行い，終了する．共存 OS の停止，再開処理はユーザ側で実現できないため，suspend_os システムコールと restart_os システムコールによる支援を行う．

### 3.5.5 正常 LKM 登録機能

正常 LKM 登録機能とは，LKM のロードが終了した時にハッシュ値の取得と保存を行う．この機能は正常データ作成機能の終了後に起動される．

#### 動作

正常 LKM 登録機能の処理の流れを図 3.10 に載せ，以下で説明する．

(1) 共存 OS のページテーブルの取得

共存 OS のページテーブルを取得する．この取得したページテーブルは，共存 OS のページテーブルの監視のために用いる．

(2) 共存 OS のページテーブルの監視

先行 OS 側が共存 OS のページテーブルを監視する．

(3) LKM のロードによる共存 OS のページテーブルの変更

共存 OS 側で LKM をロードすると，ページテーブルが変更される．

(4) 共存 OS のページテーブルの変更後のハッシュ値の算出

先行 OS が共存 OS のページテーブルの変更をトラップすると，ページテーブルの変更点から LKM 展開先アドレスを算出して，物理メモリにあるページのハッシュ値を算出する．このハッシュ値を正常データとして登録し，処理を終了する．

ユーザ空間からカーネル空間のメモリへアクセスすることはできないため，検査範囲の取得処理では．get_hash システムコールによる支援が必要となる．その他の処理はユーザ空間で実現可能であるため，ユーザ空間で実現する．

### 3.5.6 正常 LKM 検査機能

正常 LKM 検査機能とは，LKM のロード時にハッシュ値を取得し，正常データと比較する機能である．この機能は正常起動検査機能の終了後に起動される．

#### 動作

機能の処理流れを図 3.8 に載せ，以下で説明する．(1)〜(4) までは，2 章の正常 LKM 登録機能の処理流れと同じである．

(1) 共存 OS のページテーブルの取得

(2) 共存 OS のページテーブルの監視

(3) LKM のロードによる共存 OS のページテーブルの変更

(4) 共存 OS のページテーブルの変更後のハッシュ値の算出

(5) ハッシュ値と正常データとの比較

(4) で取得したハッシュ値を正常データと比較する．正常データのハッシュ値と一致しない場合は，共存 OS の走行を停止させ，結果を返却し，この機能の処理を終了する．正常データのハッシュ値と一致する場合は，結果を返却してこの機能の処理を終了する．

ユーザ空間からカーネル空間のメモリへアクセスすることはできないため，検査範囲の取得処理では．get_hash システムコールによる支援が必要となる．その他の処理はユーザ空間で実現可能であるため，ユーザ空間で実現する．

## 3.6 ヘルスチェック機能の全体の処理の流れ

実装した各機能を組み合わせた 2 つの処理を実装した．これらについて説明する．

### 3.6.1 正常データ作成処理

図 3.1 と図 3.10 を合わせて，1 つの処理として実装した．これを正常データ作成処理とした．正常データ作成処理の流れを載せ，以下で説明する．

(1) 共存 OS のカーネルイメージを展開する

netos システムコールを発行し，共存 OS のカーネルイメージをメモリ上に展開する．

(2) カーネルテキスト部のハッシュ値を取得する．

(3) ハッシュ値を正常データ (a) とする

(3) で取得したハッシュ値を正常データ (a) とする．

(4) 共存 OS の起動処理を開始する

os2_trace_os1 システムコールを発行し，共存 OS の起動処理を開始する．

(5) カーネルテキスト部のハッシュ値を取得する

共存 OS の起動処理の終了後にカーネルテキスト部のハッシュ値を取得する．

(6) ハッシュ値を正常データ (b) とする

(5) で取得したハッシュ値と正常データ (b) とする．

(7) カーネルテキスト部のハッシュ値を取得し続ける

共存 OS がバスの初期化を行う時に，共存 OS のカーネルテキスト部が変更される．この変更をトラップするために，共存 OS のカーネルテキスト部が変更されるまで取得処理を実行し続ける．

(8) ハッシュ値を正常データ (c) とする

(7) で取得したハッシュ値を正常データ (c) とする．

(9) 結果を表示

カーネルの検査が正常に終了したことをユーザに通知する．共存 OS はここで init プロセスを起動する．

(10) ページテーブルの生データを取得する

get_hash システムコールを発行し，ページテーブルのバイナリデータを取得する．

(11) 全ての LKM をロードするための処理待ちを行う

sleep 命令を用いて 10 秒間プロセスを停止し，LKM がロードされるまで待つ．

(12) ページテーブルの生データを取得する

再度 get_hash システムコールを発行し，ページテーブルのバイナリデータを取得する．

(13) 全ての LKM がロードされているかの判定を行う

(10) で取得したページテーブルと (12) で取得したページテーブルを比較する．(13) の処理を初めて実行した時は，比較結果にかかわらず再度 (11) と (12) の処理を行う．つぎに，(13) の処理の実行が 2 回目以降である時の場合を考える．比較結果で値が同一であれば，次の処理を実行する．値が異なっていれば，再度 (11) と (12) の処理を行う．

(14) LKM のハッシュ値を取得する

LKM のテキスト部のハッシュ値を取得する．ハッシュ値は 1 つの LKM に対して 1 つ算出される．

(15) 取得したハッシュ値を正常データとする

(14) で取得したハッシュ値を正常 LKM データとする．

(16) 結果を返却する

ユーザに結果を表示し，機能を終了する．

(1)，(2) により，正常データ (a) と正常データ (b) と正常データ (c) および正常 LKM データが作成される．正常データ (a) と正常データ (b) および正常データ (c) はカーネルテキスト部の検査に用いる．正常 LKM データは，LKM の検査に用いる．

### 3.6.2 検査処理

図 3.2 と図 3.8 を合わせて，1 つの処理として実装した．これを検査処理とした．検査処理の流れを載せ，以下で説明する．

(1) 正常データが存在するかを調べる

正常データが存在していれば次の処理を実行する．正常データが存在していなければこの機能を終了する．

(2) 共存 OS のカーネルイメージ展開する

netos システムコールを発行し，共存 OS のカーネルイメージをメモリ上に展開する．

(3) カーネルテキスト部のハッシュ値を取得する．

取得したハッシュ値を検査データ (a) とする．

(4) 検査データ (a) と正常データ (a) を比較する

(3) で取得した検査データ (a) を正常データ (a) と比較する．値が同じであれば次の処理を行う．値が異なれば共存 OS 停止後に異常終了する．

(5) 共存 OS 起動処理を開始する

os2_trace_os1 システムコールを発行し，共存 OS の起動処理を開始する．

(6) カーネルテキスト部のハッシュ値を取得する

共存 OS 起動処理の終了後にカーネルテキスト部のハッシュ値を取得する．取得したハッシュ値を検査データ (b) とする．

(7) 検査データ (b) と正常データ (b) を比較する

(6) で取得した検査データ (b) を正常データ (b) と比較する．値が同じであれば次の処理を行う．値が異なれば共存 OS 停止後に異常終了する．

(8) カーネルテキスト部のハッシュ値を取得し続ける

共存 OS がバスの初期化を行う時に，共存 OS のカーネルテキスト部が変更される．この変更をトラップするために，共存 OS のカーネルテキスト部が変更されるまで取得処理を実行し続ける．変更後に取得したハッシュ値を検査データ (c) とする．

(9) 検査データ (c) と正常データ (c) を比較する

(8) で取得した検査データ (c) を正常データ (c) と比較する．値が同じであれば次の処理を行う．値が異なれば共存 OS 停止後に異常終了する．

(10) 結果を表示

検査が正常に終了したことをユーザに通知する．

(11) ページテーブルの生データを取得する

get_hash システムコールを発行し，ページテーブルのバイナリデータを取得する．

(12) 全ての LKM をロードするための処理待ちを行う

sleep 命令を用いて 10 秒間プロセスを停止し，LKM がロードされるまで待つ．

(13) ページテーブルの生データを取得する

再度 get_hash システムコールを発行し，ページテーブルのバイナリデータを取得する．

(14) 全ての LKM がロードされているかの判定を行う

(11) で取得したページテーブルと (13) で取得したページテーブルを比較する．(14) の処理を初めて実行した時は，比較結果にかかわらず再度 (12) と (13) の処理を行う．つぎに，(14) の処理の実行が 2 回目以降である時の場合を考える．比較結果で値が同一であれば，次の処理を実行する．値が異なっていれば，再度 (12) と (13) の処理を行う．

(15) LKM のハッシュ値を取得する

LKM のテキスト部のハッシュ値を取得する．ハッシュ値は 1 つの LKM に対して１つ算出される．

(16) 取得したハッシュ値と正常データを比較する

(15) で取得したハッシュ値と正常データを比較する．値が同じであれば次の処理を行う．値が異なれば共存 OS 停止後に異常終了する．

(17) 結果を返却する

ユーザに結果を表示し，この機能を終了する．

検査で異常が検出された場合，共存 OS を停止させ，結果を管理部へ返却する．共存 OS の停止には共存 OS 停止機能を使用する．

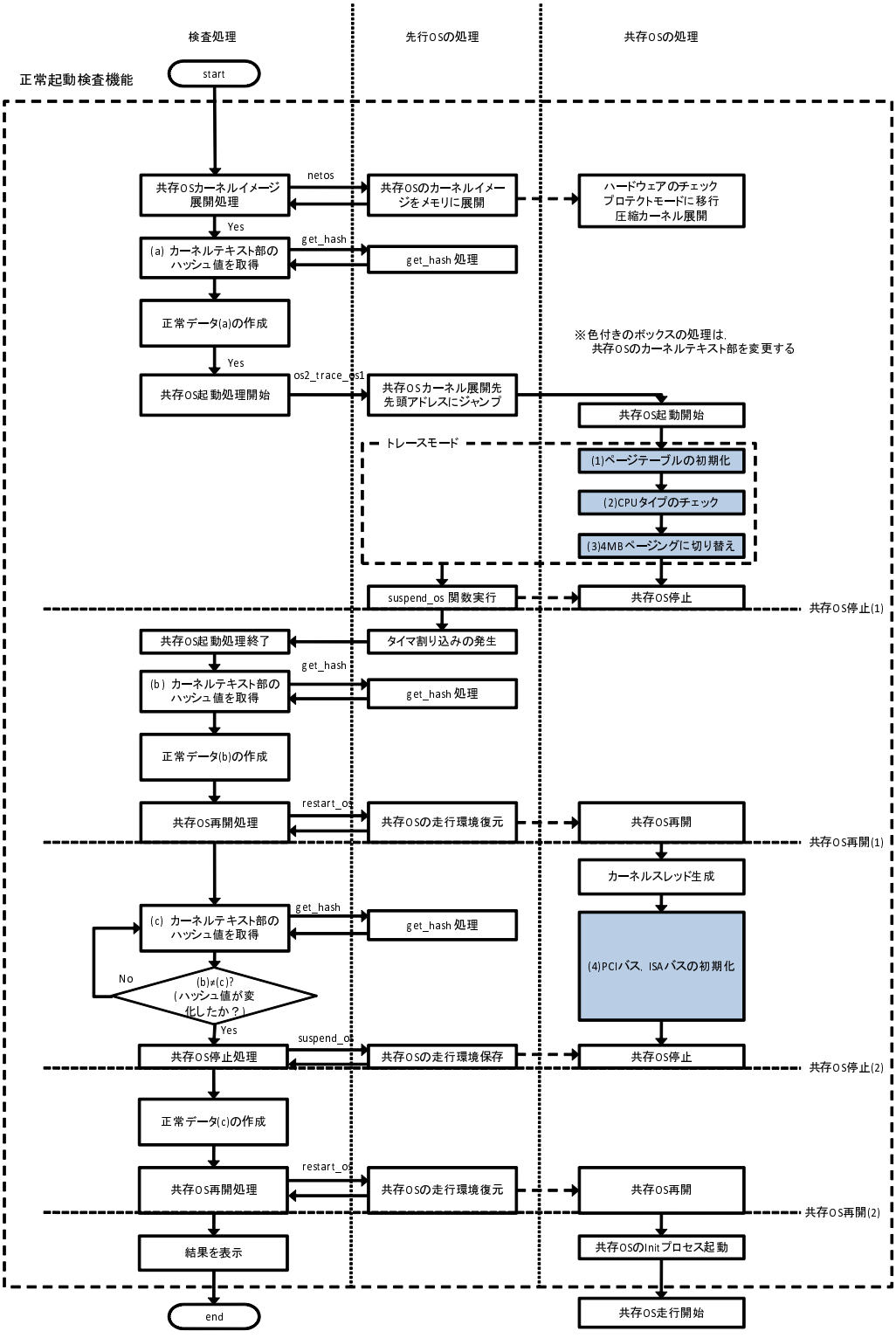


図　3.1　従来の正常データ作成処理の流れ (カーネル検査)

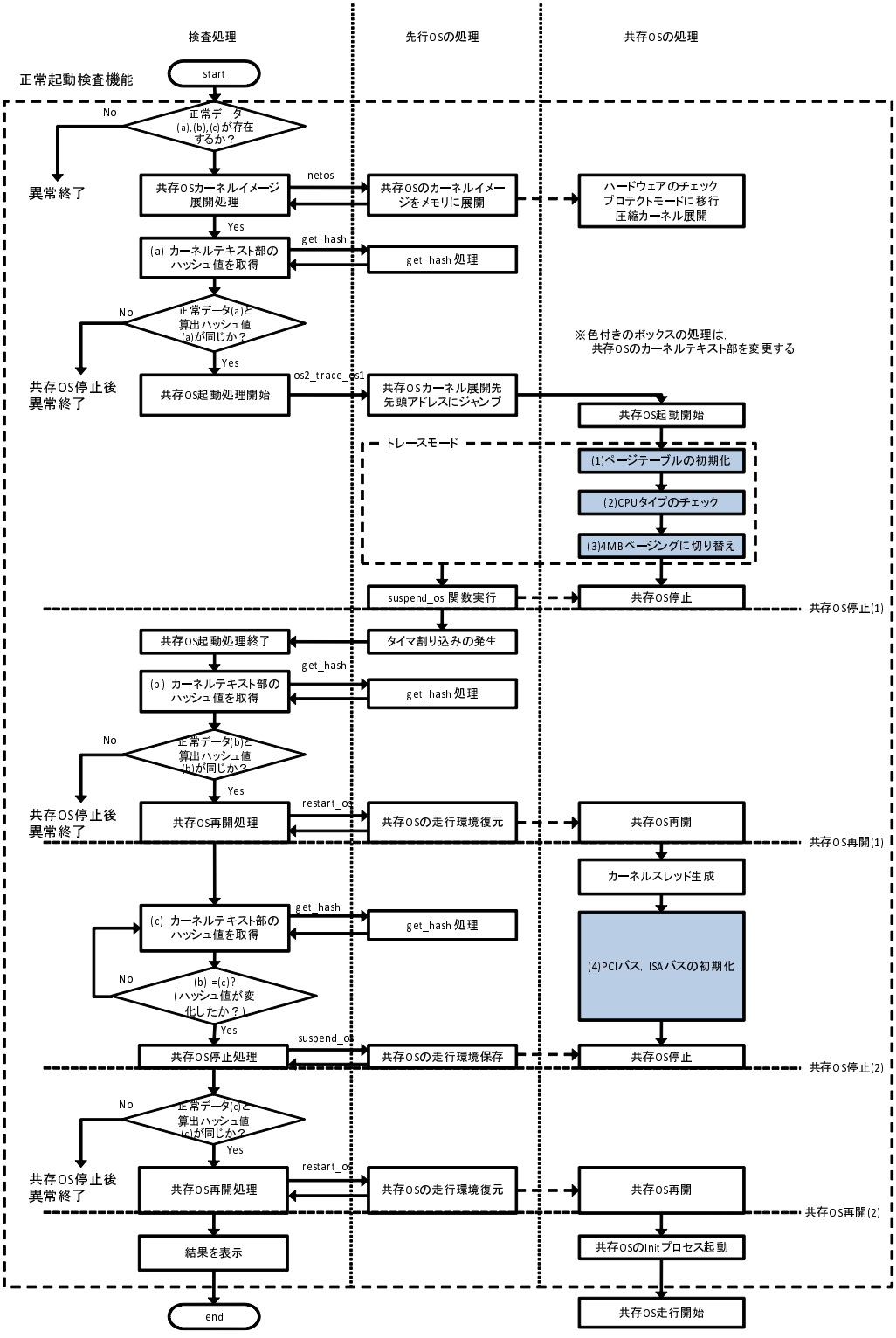


図　3.2　従来の検査処理の流れ (カーネル検査)

| ADDRESS | 00 | 01 | 02 | 03 | 04 | 05 | 06 | 07 | 08 | 09 | 0A | 0B | 0C | 0D | 0E | 0F |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 00001C50 | E3 | 01 | 00 | 05 | E3 | 01 | 40 | 05 | E3 | 01 | 80 | 05 | E3 | 01 | C0 | 05 |
| 00001C60 | E3 | 01 | 00 | 06 | E3 | 01 | 40 | 06 | E3 | 01 | 80 | 06 | E3 | 01 | C0 | 06 |
| 00001C70 | E3 | 01 | 00 | 07 | E3 | 01 | 40 | 07 | E3 | 01 | 80 | 07 | E3 | 01 | C0 | 07 |
| 00001C80 | 00 | 00 | 00 | 00 | 00 | 00 | 00 | 00 | 67 | D0 | 3C | 01 | 00 | 00 | 00 | 00 |

図　3.3　swapper_pg_dir のエントリ付近

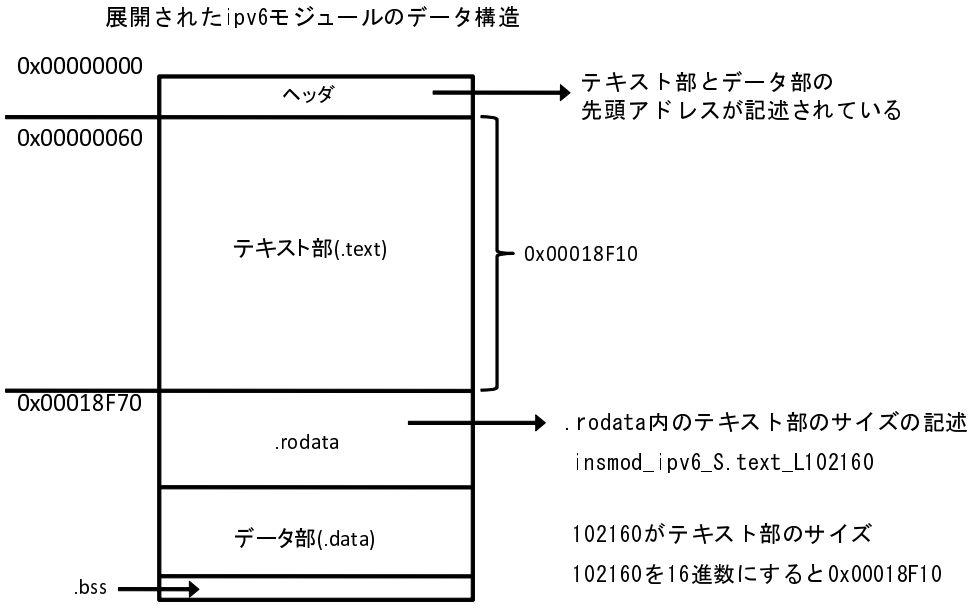


図　3.4　物理メモリに展開された LKM のデータ構造

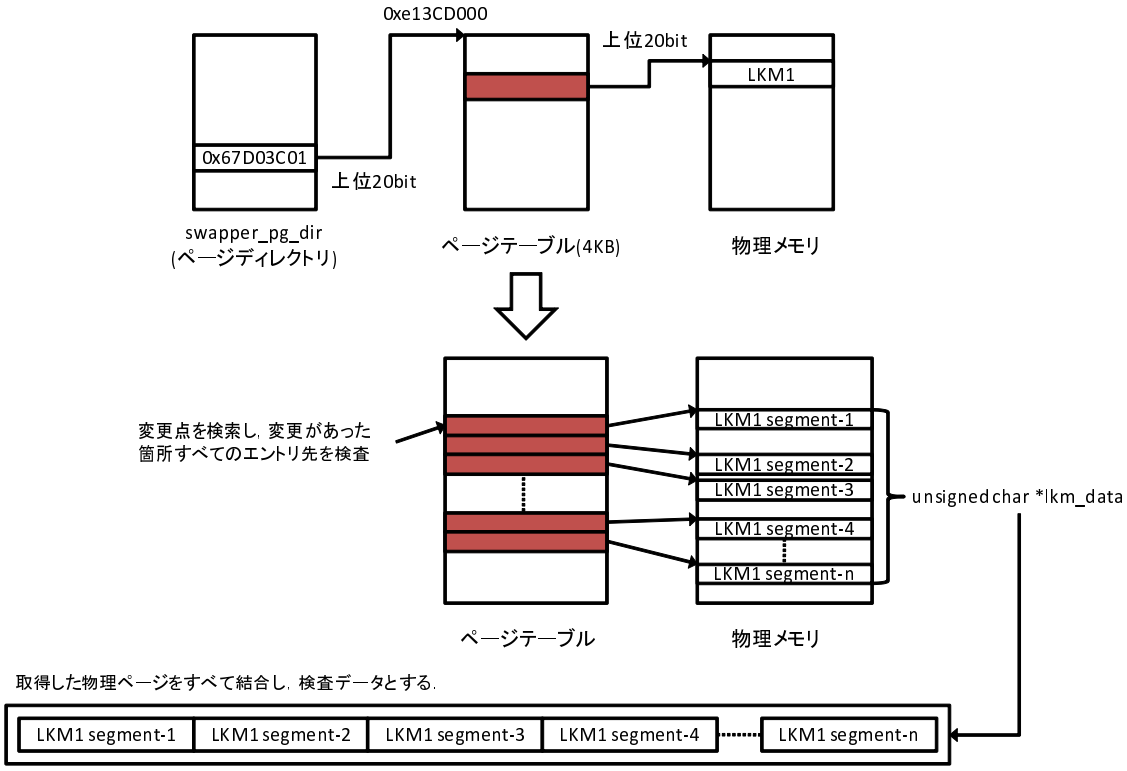


図　3.5　検査データを取得する処理の流れ

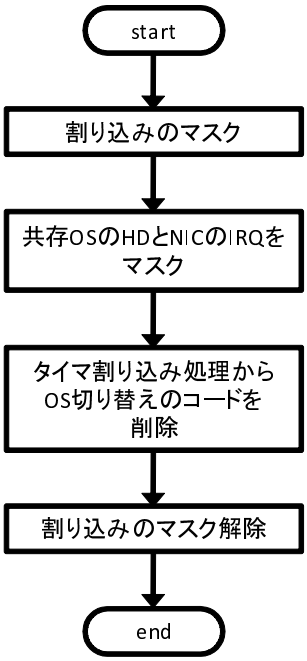


図　3.6　suspend_os システムコールの処理の流れ

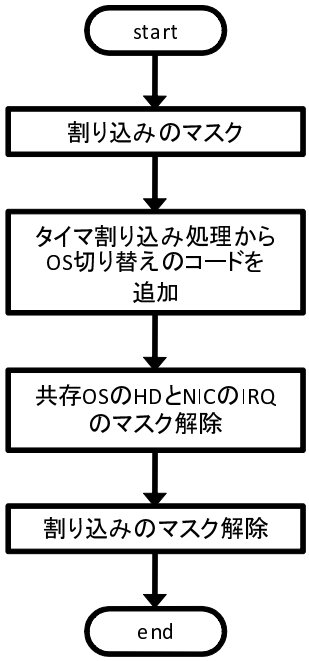


図　3.7　restart_os システムコールの処理の流れ

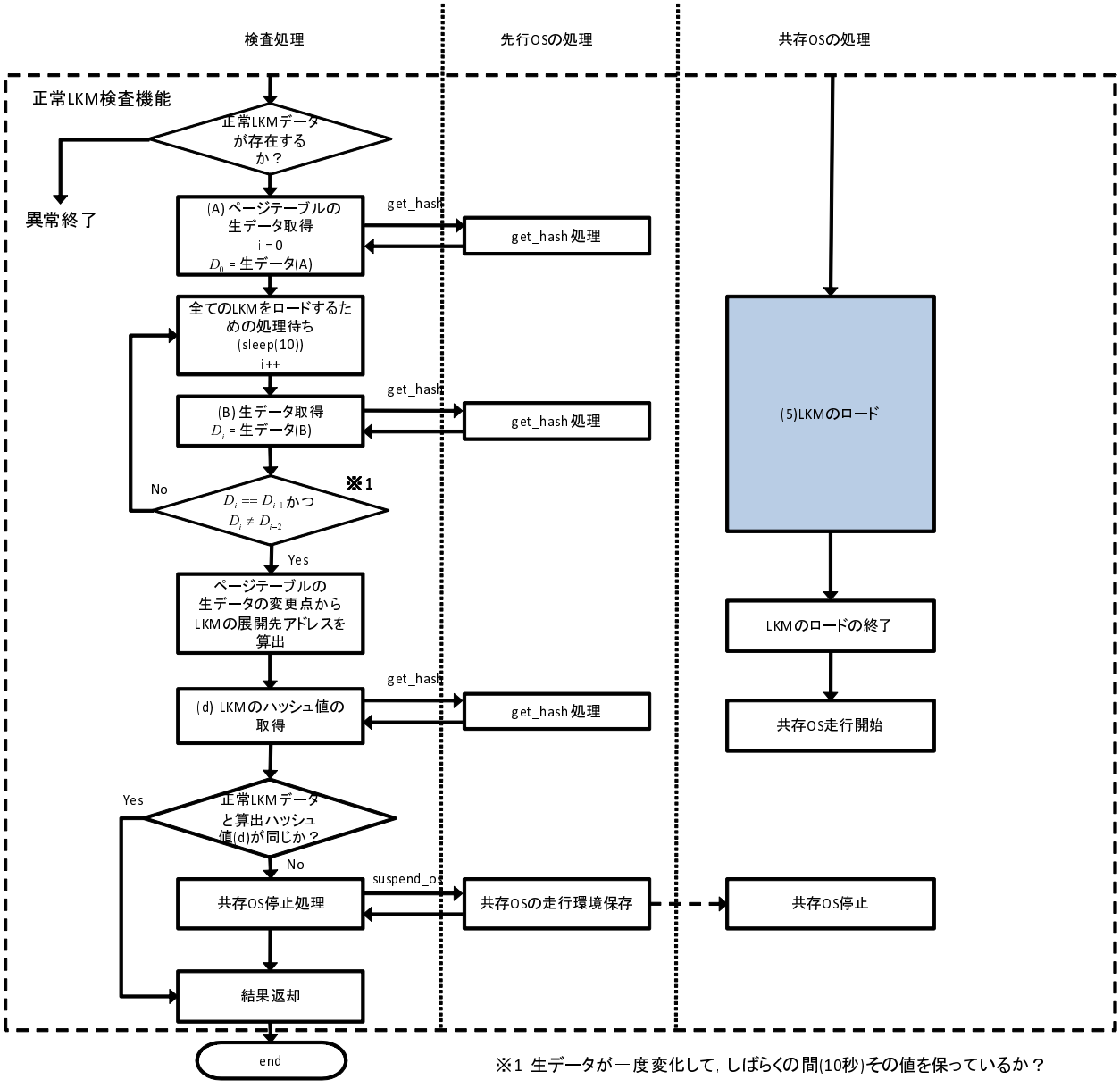


図　3.8　検査処理の流れ (LKM 検査)

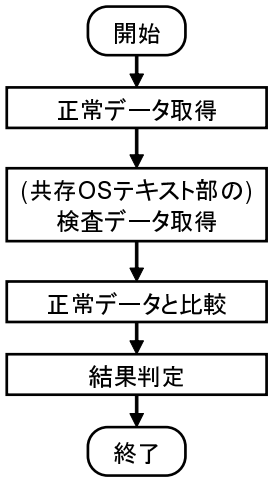


図　3.9　正常走行検査機能処理流れ

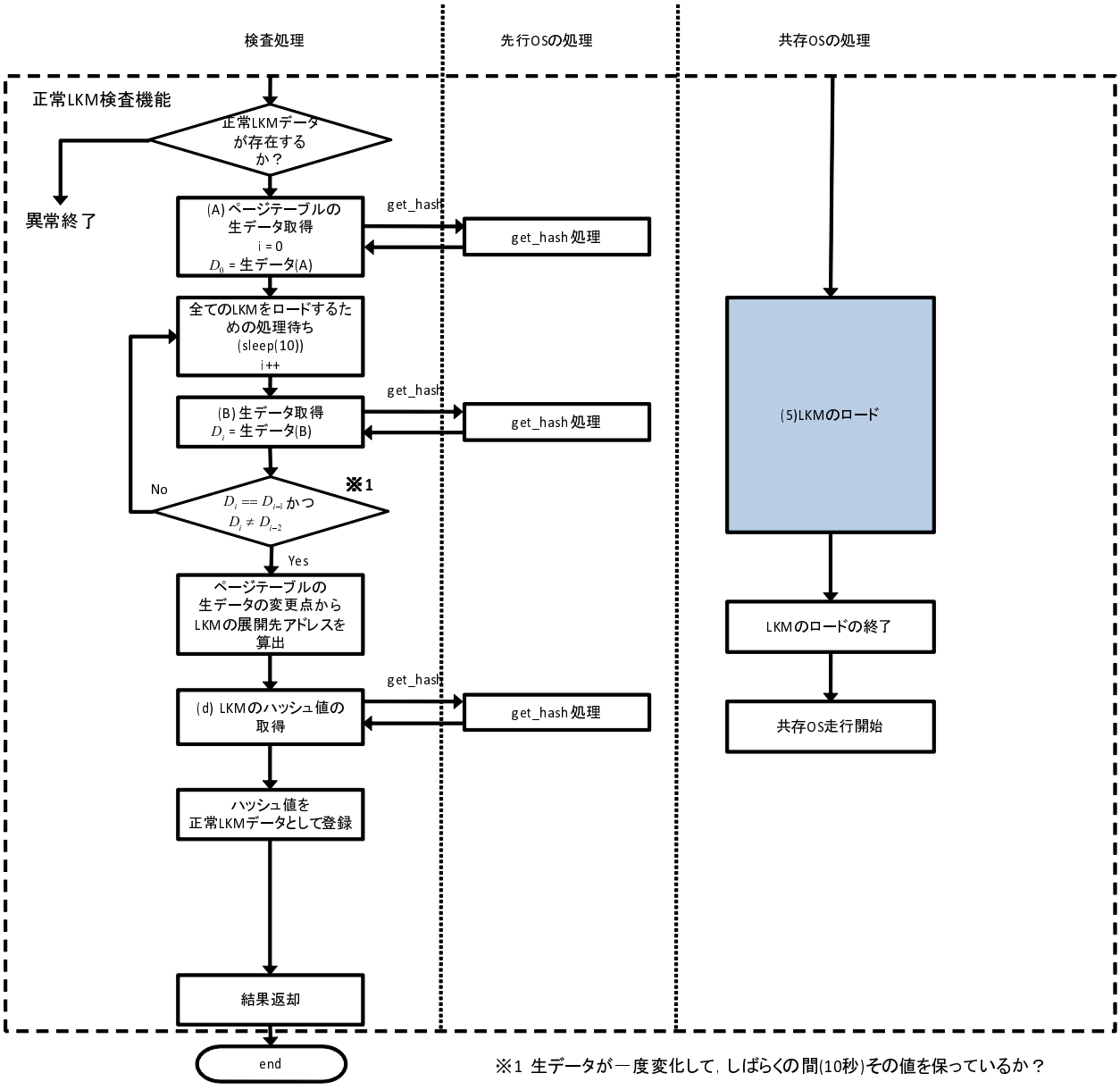


図 3.10　正常データ作成の処理の流れ (LKM 検査)

# 第 4 章

# ヘルスチェック機能の評価

## 4.1　評価環境

今回の測定に用いた CPU は Pentium4 3.0GHz である．

## 4.2　評価の目的

(1) 検査方法の違いによるオーバヘッドの調査

以下の 2 つの方法による get_hash システムコールのオーバヘッドをそれぞれ比較し，(b) の方がオーバヘッドが小さいことを示す．オーバヘッドの値は CPU クロックサイクル数とする．

(A) 検査範囲のバイナリデータ (以降，検査データ) を取得し，正常データと直接比較する方法

(B) 検査範囲のバイナリデータのハッシュ値 (以降，検査データのハッシュ値) を算出し，正常データと比較する方法

以降，(a) の方法をバイナリ検査，(b) の方法をハッシュ検査とする．

(2) 検査処理のオーバヘッドの定量化

ヘルスチェック機能の検査処理には，カーネルテキスト部の検査と LKM の検査がある．検査するカーネルテキスト部は一つしかないが，検査する LKM は複数存在する．複数の LKM がロードされても，オーバヘッドが明確に分かるようにするため，検査にかかる合計時間を定量化する．

## 4.3　評価の方法

### 4.3.1　二つの検査方法のオーバヘッドの測定方法

バイナリ検査では，get_hash システムコールを用いて検査データを取得し，正常データと１バイトずつ比較する．ハッシュ検査では，get_hash システムコールを用いて検査データのハッシュ値を取得し，正常データと１バイトずつ比較する．バイナリ検査とハッシュ検査のオーバヘッドを，それぞれ $T_1, T_2$ とすると，以下の式で表せる．

$$\begin{aligned} T_1 = &(\text{検査データの取得処理のオーバヘッド}) + \\ &(\text{比較処理のオーバヘッド}) \end{aligned} \tag{4.1}$$

$$\begin{aligned} T_2 = &(\text{ハッシュ値生成処理のオーバヘッド}) + \\ &(\text{検査データのハッシュ値取得処理のオーバヘッド}) + \\ &(\text{ハッシュ値比較処理のオーバヘッド}) \end{aligned} \tag{4.2}$$

以下に，取得処理と比較処理の具体的な測定方法を示す．

(1) get_hash システムコールによる処理のオーバヘッド

図 4.1 に，評価プログラムの処理の流れを載せ，以下で説明する．get_hash システムコールを発行する直前に，クロックサイクル数を取得し保存する．次にシステムコールの処理を実行する．そして，AP 側に処理が戻った直後に，クロックサイクル数を取得し保存する．クロックサイクル数の取得には，rdtsc 命令を用いる．最後に，(図 4.1 の time2) と (図 4.1 の time1) との差分を取ることにより，システムコールのオーバヘッドを測定する．

(2) 正常データと取得データとの比較処理のオーバヘッド

図 4.2 に，評価プログラムの処理の流れを載せ，以下で説明する．比較処理を実行する直前に，クロックサイクル数を取得し保存する．次に比較処理を実行する．比較処理とは，検査データを 1 バイトずつ比較する処理である．そして，比較処理の終了直後に，クロックサイクル数を取得し保存する．クロックサイクル数の取得には，(1) と同様に rdtsc 命令を用いる．最後に，(図 4.2 の time2) と (図 4.2 の time1) との差分を取ることにより，比較処理のオーバヘッドを測定する．

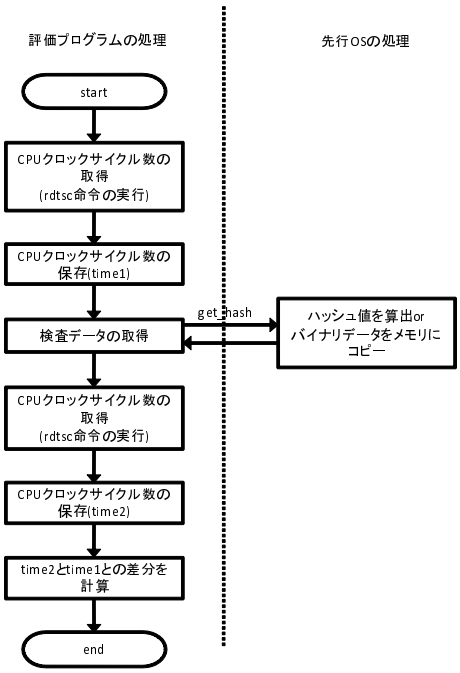


図 4.1 評価処理の流れ (システムコールの処理)

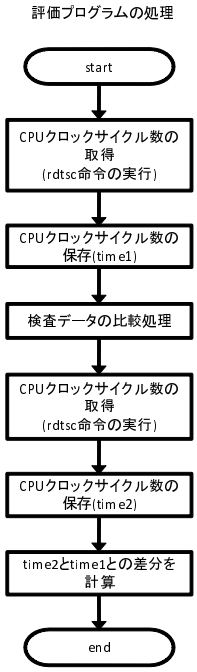


図 4.2 評価処理の流れ (比較処理)

実装上では，不一致が見つかれば，その時点で比較処理が中断される．しかし，今回の評価では，正常データと検査データが同一であるとする．このため，比較処理のオーバヘッドは，常に最大値となる．

get_hash システムコールのオーバヘッドと比較処理のオーバヘッドの測定することで，ハッシュ検査の方がオーバヘッドが小さいことを示す．

### 4.3.2 検査処理のオーバヘッドの定量化方法

検査処理のオーバヘッド $T$ は以下のようにして表せる．

$$T = (\text{カーネルテキスト部の検査におけるオーバヘッド}) + (\text{LKM の検査におけるオーバヘッド}) \tag{4.3}$$

カーネルテキスト部の検査処理におけるオーバヘッドを求め，LKM の検査におけるオーバヘッドを求める．2 種類のオーバヘッドの和により，検査処理のオーバヘッドを求めることができる．具体的なオーバヘッドの算出の過程は，4.4.2 節で説明する．

表 4.1 評価結果のまとめ (バイナリ検査)

| データサイズ ($m$) | 取得処理オーバヘッド ($t_1$) | 比較処理オーバヘッド ($c_1$) | オーバヘッドの合計 ($T_1$) |
|---|---|---|---|
| 0 | 6,216 | 164 | 6380 |
| 0x100(256B) | 8,100 | 3,716 | 11816 |
| 0x1000(4KB) | 31,804 | 52,664 | 84468 |
| 0x10000(64KB) | 402,796 | 1,197,364 | 1600160 |
| 0x100000(1MB) | 5,009,080 | 15,876,112 | 20877816 |

表 4.2 評価結果のまとめ (ハッシュ検査)

| データサイズ ($m$) | 取得処理オーバヘッド ($t_2$) | 比較処理オーバヘッド ($c_2$) | オーバヘッドの合計 ($T_2$) |
|---|---|---|---|
| 0 | 13532 | 88 | 13620 |
| 0x100(256B) | 21,772 | 88 | 21860 |
| 0x1000(4KB) | 49,964 | 88 | 50052 |
| 0x10000(64KB) | 636,776 | 88 | 636864 |
| 0x100000(1MB) | 10,102,096 | 88 | 10102184 |

## 4.4 評価の結果

### 4.4.1 バイナリ検査とハッシュ検査のオーバヘッドの比較

2 章で述べた評価方法を用い，評価した結果を表 4.1 と表 4.2 にまとめる．

表 4.1 と 4.2 のオーバヘッドの合計を見ると，データサイズが 4KB よりも大きい場合，ハッシュ検査の方がオーバヘッドの合計が小さいことが分かる．しかし，検査データのサイズが 0B と 256B のように小さい場合，バイナリ検査の方がオーバヘッドが小さいことが分かる．

### 4.4.2 検査処理におけるオーバヘッドの定量化

検査処理のオーバヘッド $T$ は，4.3.2 項で以下のようにして表せると述べた．

$$T = \text{(カーネルテキスト部の検査におけるオーバヘッド)+}\\ \text{(LKM の検査におけるオーバヘッド)}$$

カーネルテキスト部の検査には，ハッシュ検査を行う．また，LKM の検査には，バイナリ検査とハッシュ検査を行う．このため，検査処理のオーバヘッドを定量化するには，バイナ

リ検査とハッシュ検査のオーバヘッドを定量化する必要がある．よって，4.4.1 の評価により出力されたデータから，バイナリ検査とハッシュ検査におけるオーバヘッドの定量化を行う．オーバヘッドの定量化において，使う記号を定義する．まず，検査範囲を $m$ とする．次に，3 章の (1) で発生するオーバヘッドのうち，バイナリ検査におけるオーバヘッドを $t_1$ とし，ハッシュ検査におけるオーバヘッドを $t_2$ とする．最後に，3 章の (2) で発生するオーバヘッドのうち，バイナリ検査におけるオーバヘッドを $c_1$，ハッシュ検査におけるオーバヘッド $c_2$ とする．これらを用いると，式 (1) と式 (2) は以下の式になる．

$$T_1 = t_1 + c_1$$

$$T_2 = t_2 + c_2$$

ここで，表 4.1 と表 4.2 から，$T_1$ と $T_2$ は，検査範囲 $m$ に比例すると予測される．また，$a$ を get_hash システムコールの呼び出しにおけるオーバヘッドとすると，$T_1$ と $T_2$ は，それぞれ以下の式で表せる．

$$T_1(m) = a + b_1 m$$

$$T_2(m) = a + b_2 m$$

ここで，$b_1$ と $b_2$ は，傾きとする．

検査範囲 $(m)$ は 0B から 1MB まで，256B づつ変化させた．このため，要素数は 4096 個であり，定量化には十分な要素数が存在する．この 4096 個のデータに最小 2 乗法を用いることで，近似式を求める．今回の評価では，Excel の LINEST 関数を用い，近似式を求めた．結果として，以下の式が得られた．

$$T_1(m) = 19.052m + 5632.6 \tag{4.4}$$

$$T_2(m) = 9.6393m + 9020.4 \tag{4.5}$$

上記の式をグラフ化して，図 4.3 に載せる．直線の周りに要素の分布を示している．また，$T_1(m)$ の直線と $T_2(m)$ の直線の交点を図 4.4 に載せる．

図 4.4 のグラフによって，ほとんどの場合，ハッシュ検査のオーバヘッドである $T_2$ の方が，バイナリ検査のオーバヘッドである $T_1$ よりも小さいことが示される．

バイナリ検査とハッシュ検査のオーバヘッドから，カーネルテキスト部の検査処理におけるオーバヘッドを求める．カーネルテキスト部の検査はハッシュ検査で行われる．カーネルテキスト部のサイズを $m_0$ とすると，その検査処理のオーバヘッドは，$T_2(m_0)$ と表される．

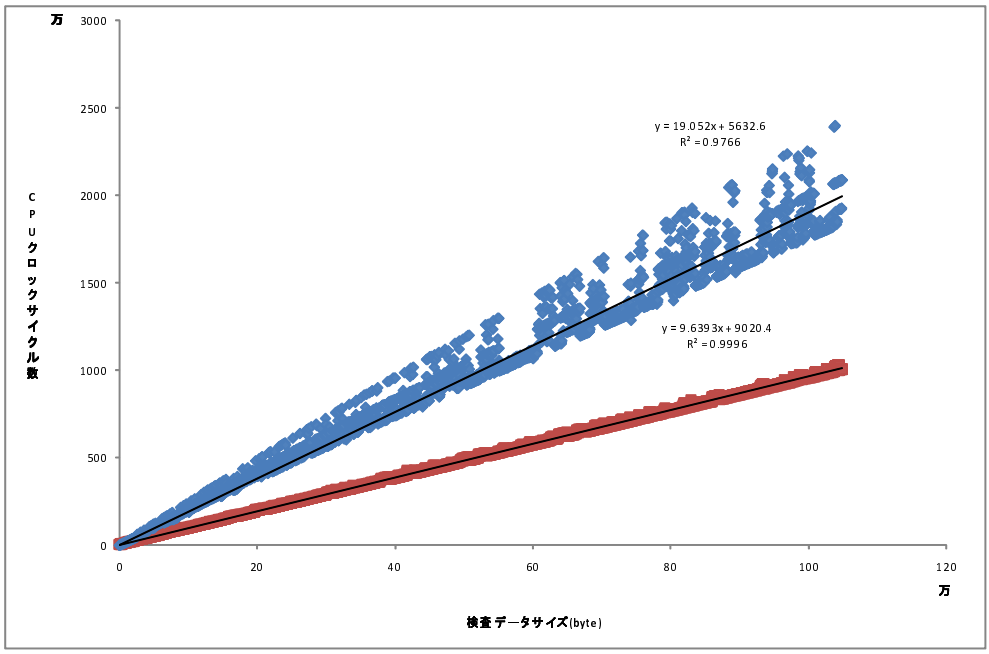


図　4.3　バイナリ検査とハッシュ検査のオーバヘッド (0〜1200000byte)

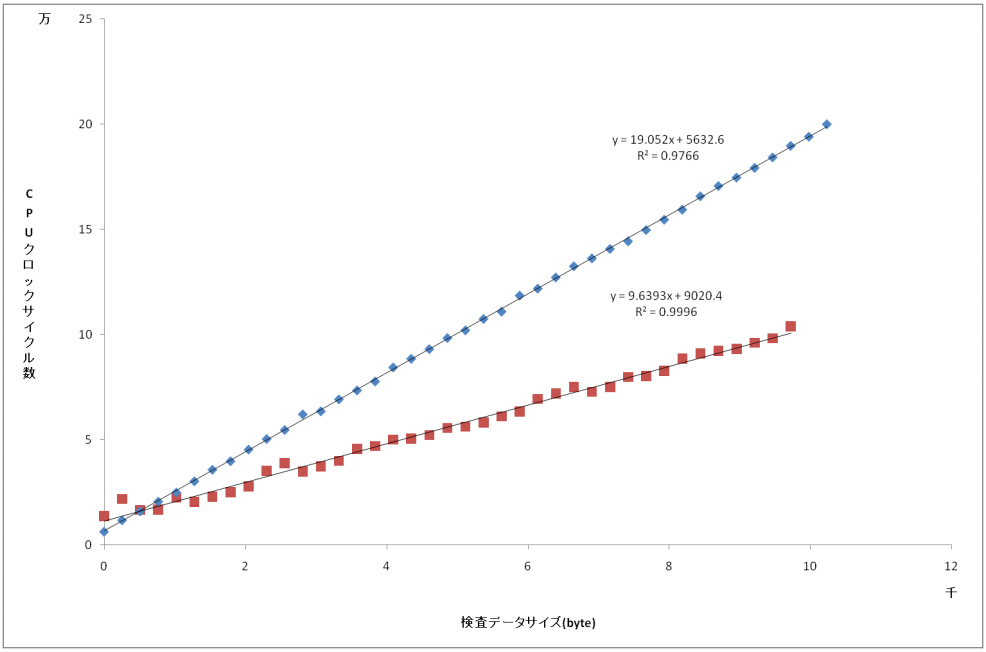


図　4.4　バイナリ検査とハッシュ検査のオーバヘッド (0〜12000byte)

次に LKM の検査処理におけるオーバヘッドを求める．2 章の (2) において，LKM は複数検査されるため，その合計時間を求める必要がある．ここで，i 番目の検査データのサイズを $m_i$ とすると，i 番目の検査におけるオーバヘッドは，以下の式で表される．

$$T_1(m_i) = a + b_1 m_i \qquad (i = 1, 2, \ldots, n)$$
$$T_2(m_i) = a + b_2 m_i \qquad (i = 1, 2, \ldots, n)$$

LKM の検査には，ハッシュ値の取得と比較を行うため，ハッシュ検査を用いる．また，ページテーブルの生データの取得と比較を行うため，バイナリ検査も用いる．このため，i 番目の LKM の検査処理におけるオーバヘッドを $L_i$ とすると，そのオーバヘッドは以下の式で表される．

$$L_i = T_1(4096) + T_2(m_i)$$

ここで $T_1(4096)$ の 4096 は，ページテーブルのサイズ (4KB) である．よって，LKM が $n$ 個ロードされた場合の検査処理におけるオーバヘッド $T(n)$ は，以下の式で表される．

$$T(n) = T_2(m_0) + \sum_{i=1}^{n} L_i$$

これを変形して，

$$T(n) = T_2(m_0) + \sum_{i=1}^{n} \{T_1(4096) + T_2(m_i)\} \tag{4.6}$$

この式が，ヘルスチェック機能の検査処理におけるオーバヘッドである．

## 4.5　考察

4.4 節の結果から，以下の 2 つが考察の結果となる．

(1) 検査処理のオーバヘッドは小さい．

ハッシュ検査において，検査範囲が 1MB の時は，式 4.6 で計算すると，CPU クロックサイクル数は 10102184 である．これは 3GHz の CPU で，およそ $3\mu$ 秒に相当する．1MB 以上の検査データで，ハッシュ検査を頻繁に行うと，検査処理のオーバヘッドは非常に大きくなる．しかし，1MB 以上の検査範囲で検査を行う場合は，カーネルテキスト部の検査だけである．検査処理において，カーネルテキスト部は 1 回しか検査さ

れない．また，LKM のサイズは，ほとんどが 100KB 以下である．つまり，1MB 以上の検査範囲での検査は，ほとんど行わない．よって，検査処理のオーバヘッドは，十分に小さいといえる．

(2) 検査処理にはハッシュ検査を用いる

図 4.4 のグラフにより，検査データのサイズが 360 バイトより小さければ，バイナリ検査を行う方がオーバヘッドが小さいことが示される．しかし，カーネルテキスト部は約 1MB であり，LKM のサイズは 1KB を超えるものがほとんどである．このため，検査処理において，512 バイト以下のデータを検査する処理が頻発することはない．よって，検査処理においては，ハッシュ検査を用いる方が，オーバヘッドが小さくなる．

# 第 5 章

# おわりに

本論文では，TwinOS におけるヘルスチェック機能の課題と対処を明確にし，それを基にヘルスチェック機能の実装について述べた．

まず，TwinOS におけるヘルスチェック機能の課題を明確にした．具体的には，課題として，不正な LKM のロードへの対処と LKM のロードによるカーネルテキスト部の変化への対処があることを示した．

次に，TwinOS を用いた場合の OS が支援すべきインタフェースの設計を示し，どのように実装したかを述べた．具体的には,「正常 LKM 登録機能」「正常 LKM 検査機能」という 2 つの機能を実装し,「正常データ作成機能」と「正常 LKM 登録機能」を用いた正常データ作成処理と「正常起動検査機能」と「正常 LKM 検査機能」を用いた検査処理を実装した.「共存 OS 停止機能」「共存 OS 再開機能」は，上記の二つの処理において，適宜呼び出される．また,「正常走行検査機能」は共存 OS の走行中に，任意の契機で検査を行う．

最後に，ヘルスチェック機能の評価について述べた．具体的には，評価結果の考察により，検査処理にはハッシュ検査を用いることでオーバヘッドを低減できることを述べた．

残された課題として，検査の常駐化と実際のネットワークを介した攻撃に対して，ヘルスチェック機能が有効であることの証明がある．

# 謝辞

本研究を進めるにあたり，懇切丁寧なご指導をしていただきました
谷口秀夫教授，乃村能成准教授，ならびに後藤佑介助教に心より感謝の意を表します．また，日頃の研究活動において，お世話になりました研究室の皆様に感謝いたします．

最後に，本研究を行うにあたり，経済的，精神的な支えとなった家族に感謝いたします．

# 参考文献

[1] 田渕正樹，伊藤健一，乃村能成，谷口秀夫, “ 二つの Linux を共存走行させる機能の設計と評価，” 電子情報通信学会論文誌 (D-I)，Vol.J88-D-I，No.2，pp.251-262，2005.

[2] 池内達也, “ ヘルスチェック機能を支援する TwinOS の実現，” 岡山大学工学部情報工学科修士論文，2009.

[3] 上口祐也, “ TwinOS のヘルスチェック機能の設計，” 岡山大学工学部情報工学科卒業論文，2009.