社団法人 情報処理学会 研究報告
IPSJ SIG Technical Report

2008－OS－108 (9)
2008／4／23

デバイスドライバのための障害回復機構

井上翔大[†1] 大山恵弘[†1]

コンピュータで多数のデバイスが扱えるようになり，そのデバイスドライバの数も膨大になっている．しかし，デバイスドライバには信頼性が低いものがあり，それらはOSに障害を引き起こす主要因になっている．そこで，本研究では，デバイスドライバのデッドロックや無限ループによる障害を検知し，回復する機構を提案する．我々は，Linuxを対象として提案機構の設計と実装を行った．提案機構では，デッドロックや無限ループの検出にはタイマを使用し，スタックの書き換えによって強制的に障害回復機構へ処理を移す．提案機構は，Loadable Kernel Module (LKM) として実装されているので，デバイスドライバ本体や，カーネルに変更を加えることなく使用できる．我々は実験によって，デバイスドライバに起きた障害から回復できることを確認し，オーバヘッドも非常に小さいことを示した．

Fault Recovery Mechanism for Device Drivers

Shouta Inoue[†1] and Yoshihiro Oyama[†1]

Today we can use many devices on computers and a number of device drivers are available. Unfortunately, some device drivers contain errors, and consequently device drivers are the primary cause of operating system faults. In this work, we propose a mechanism for detecting and recovering from faults of device drivers due to deadlocks and execution of infinite loops. We designed and implemented the mechanism on Linux. The mechanism uses a timer to detect deadlocks and execution of infiniteloops. It forces a misbehaving device drivers to jump to the operation for fault recovery by rewriting a return address in the execution stack. The mechanism is implemented as a Loadable Kernel Module(LKM) and thus can be used without modification to device drivers or the operating system kernel. Experimental results showed that the operating system kernel could recover its control from faulty device drivers, and that the overhead imposed on device accesses was significantly small.

1. はじめに

現在，デバイスドライバの信頼性の低さが，オペレーティングシステム（OS）の障害の主要因になっている．例えば，1) によると，Windows XPにおけるクラッシュの原因のうち85%はデバイスドライバのバグによるものだという．また，Chouらの調査[2]では，Linuxカーネルにおいて，デバイスドライバは，その他の部分に比べて3–7倍のバグがあるという報告がなされている．

こういった信頼性の低さに加えて，デバイスドライバはカーネルモードで動作するため，不正なメモリアクセスやデッドロック等のバグが容易にOSをクラッシュさせたり，フリーズさせたりする．例えば，デバイスドライバが，カーネル空間でスピンロックのデッドロックを起こすと，OS自体がフリーズしてしまい，一切の処理ができなくなることがある．

†1 電気通信大学
University of Electro-Communications

したがって，デバイスドライバの信頼性を高めることは，システム全体の信頼性のためにも非常に重要である．これまでもNooks[3]を始めとして，デバイスドライバ等の追加されたモジュールのメモリ保護については研究が行われてきた．しかし，追加モジュールにおけるデッドロックや無限ループからのカーネルの保護については，対策が取られず，現実的な保護機構は存在しなかった．

そこで，本研究ではデバイスドライバのデッドロックや無限ループによる障害からカーネルを保護することを目的とする．デバイスドライバがデッドロックや無限ループに陥ったことを検知すると，そのドライバを強制的に終了させて，カーネルに処理を戻すようにする機構を構築する．主にスタックの書き換えによって，障害回復を行う．デバイスドライバからのカーネルメモリの保護については扱わない．また，提案機構はLinuxを対象とし，カーネルコードに手を加えることなくLoadable Kernel Module（LKM）として実装し，使用者や開発者が手軽に使えることを目標にする．

```
struct file_operations mydevice_fops = {
    .owner   = THIS_MODULE,
    .llseek = mydevice_llseek,
    .read    = mydevice_read,
    .write   = mydevice_write,
    .ioctl   = mydevice_ioctl,
    .open    = mydevice_open,
    .release    = mydevice_release,
    .aio_read  = NULL,
    .aio_write = NULL,
             :
             :
};
```

図 1 デバイスドライバの関数テーブル
Fig. 1 Function table of a device driver.

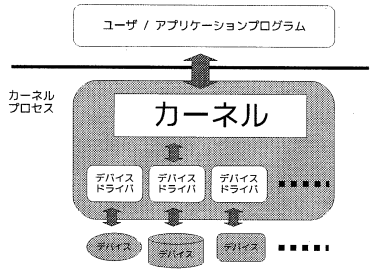

図 2 Linux におけるデバイスドライバ
Fig. 2 Device drivers on Linux.

2. Linux におけるデバイスドライバ

デバイスドライバとは，特定の入出力機器を制御し，カーネルに対して統一的なインターフェイスを提供するソフトウェアである．デバイスドライバによって，カーネルは登録された関数を呼び出すだけで，デバイスに対して open，close，read，write といった処理を行うことができる．

Linux において，ユーザプログラムがシステムコールを発行すると，カーネルに処理が移り，デバイスドライバの open，read などのインターフェイス関数を呼ぶ．デバイスドライバはデバイスを制御して，それぞれに対応する動作をさせる．これらのインターフェイス関数は，図 1 のような関数テーブルに登録され，必要に応じてカーネルがこれらの関数を呼び出す．

Linux におけるデバイスドライバを図 2 に示す．ユーザがデバイスへの入出力を伴うシステムコールを発行すると，カーネルに処理が移り，カーネルスレッドが 1 つ作られる．デバイスドライバはそのスレッド上で実行され，他の全てのカーネルスレッドとメモリ空間を共有する．カーネル内における処理では，コンテキストスイッチやプロセス間通信が必要ないため高速だが，一つのカーネルの機能がメモリや CPU 使用時間を通じて，他の全ての機能に影響を及ぼすという問題がある．例えば，バグのあるデバイスドライバがカーネルに組み込まれた場合を考える．もしこのデバイスドライバが，カーネルの重要なデータを持つ構造体を誤って上書きしたり，存在しないメモリ領域へアクセスしたりすると，そのカーネル自体がクラッシュする可能性もある．また，カーネルロックを取って無限ループなどを行うと，CPU 資源を独占して，カーネルの他の全ての機能を停止させることになる．

3. 関連研究

主な関連研究としては，デバイスドライバの実行を監視するもの，デバイスドライバをユーザ空間で実行するもの，デバイスドライバに型安全な言語を使うものなどがある．

3.1 デバイスドライバの実行を監視する研究

デバイスドライバの実行を監視し，異常な動作をしているようであれば，その障害がカーネルに及ばないようにするという研究である．

Nooks[3] はカーネルとデバイスドライバの間の全ての関数呼び出しを監視し，その引数や返り値を検査するシステムである．カーネル中の重要なデータのコピーを持ち，デバイスドライバはそのコピーにしかアクセス出来ないようにすることで，カーネルとデバイスドライバを隔離している．また，Nooks を使って，障害を発生したデバイスドライバを再ロードして，再び使用できるようにする研究もある[4]．Nooks は，デバイスドライバにおける多くの障害を検知できるが，カーネルに大幅な変更を加える必要がある．また，デバイスドライバが無限ループやデッドロックに陥ると回復できない．

David らの研究[5] は，本研究と同じくカーネル内での無限ループやデッドロックから回復するためのものである．タイマーによって障害を検知し，OS を再起動させて回復を図る．その際，メモリやレジスタの内容を再利用することで，OS を障害発生前の状態に復元している．しかし，データの一貫性が崩れる可能性があったり，再起動時にシステムにダウンタイムが生じるといった問題がある．本研究では，強制的に処理をカーネルに戻すことで，障害からの回復を図っており，このような問題は生じない．

Herber らの研究[6] は，MINIX を対象とし，異常を起こしたデバイスドライバだけを再起動させることで，障害から回復するというものである．再起動後に，デバイスドライバ自身やその上位層のコンポーネントによって，状態を復元させ，影響がアプリケーションプログラムに及ばないようにする．上位層のコンポーネントからの通報があった場合や，定期的なシグナルへの応答が滞った場合に，デバイスドライバに異常が起

こったと判断する．ただし，マイクロカーネルの機能を前提としており，Linux などのモノリシックカーネルには直接適用することはできない．

3.2 デバイスドライバをユーザモードで実行する研究

デバイスドライバをユーザモードで実行できれば，デバイスドライバに起因する障害からカーネルを保護することができる．この様な研究としては，デバイスドライバのコードの一部を，ユーザ空間で実行するものや，マイクロカーネルなどがある．

前者の研究としては，カーネル内に代理デバイスドライバを置き，デバイスドライバ自体はユーザモードで実行するものがある[7]．これは，ハードウェアと直接やり取りする部分だけをカーネル空間に残し，他の部分をユーザ空間で実行するというものである．互いの通信は代理デバイスドライバを使って行う．また，既存のデバイスドライバを，カーネル空間に残す部分とユーザ空間に移してもいい部分に自動的に分割するという研究[8]もある．これらの研究では，ユーザ空間のライブラリが利用できたり，デバイスドライバをユーザレベルに隔離できるなどの利点があるが，デバイスドライバを再設計する必要がある．また，ユーザ空間とカーネル空間の切替が重く，頻繁に特権命令を実行するデバイスドライバには向かないなどの欠点がある．

後者のマイクロカーネル[9),10)]とは，カーネル空間で実行される処理を限定し，それ以外の処理をユーザ空間上のプロセスとして実行するものである．デバイスドライバは，ユーザプロセスとして実行されるので，その障害の多くからカーネルを保護することができる．しかし，プロセス間の通信が大きなオーバヘッドとなる．本研究におけるカーネル保護は，マイクロカーネル程徹底したものではないが，既存のモノリシックカーネルにも手軽に導入でき，オーバヘッドも小さい．

3.3 デバイスドライバに型安全な言語を使う研究

Java などの型安全な言語でデバイスドライバが書かれていれば，バグによって，デバイスドライバがカーネル内の不正なメモリ領域にアクセスするのを防ぐことができる．SafeDrive[11)]は，コード変換によって，C に型安全性を付加するシステムである．デバイスドライバの開発者が変数やポインタの範囲を明示してやり，コード変換器が変換時に検査を行う．その上でコード変換器は，プログラムに実行時のチェックを挿入する．そして，実行時にそのチェックでエラーになると，異常と見なしてデバイスドライバを再ロードする．このシステムでは，多くの障害から復帰出来る反面，OS のカーネルやデバイスドライバ自体に手を加える必要がある．Singularity[12)]や，JavaOS[13)]は，OS そのものが型安全な言語で書かれている．当然デバイスドライバ自体も型安全な言語で書かれるため，バグを含むデバイスドライバによる不正なメモリアクセスを防ぐことができる．また，これらの OS はマイクロカーネルでもあるため，万が一デバイスドライバに障害が起こったとしても，その影響を最小限に止めることができる．

4. 提案機構

デバイスドライバにおけるデッドロックや無限ループといった障害を検知し，その障害を起こしたデバイスドライバを強制終了する機構を，Linux 上に実装する．強制終了によって，OS がクラッシュしたり，他の異常が起こる可能性もあるが，OS ごとフリーズし続ける事態は避けることができる．提案機構は，デバイスドライバからのカーネルメモリの保護や，デバイスドライバを使った乗っ取りなどの攻撃に対する防御は行わない．また，LKM として実装することで，カーネルやデバイスドライバ自体に変更を加えず，使用者や開発者が提案機構を手軽に使えるようにする．

4.1 提案機構の概要

提案機構は，

(1) デバイスドライバのインタフェース関数のフック関数
(2) デバイスドライバの実行監視部
(3) デバイスドライバの強制終了部

の 3 つからなり，図 3 に示す構成になっている．

1 のフック関数は，デバイスドライバとカーネルの間に置かれ，デバイスドライバに処理が渡ったことや，デバイスドライバから処理が戻ってきたことを，2 の実行監視部に伝える．

2 の実行監視部は，デバイスドライバの実行を監視し，異常を検知した場合は 3 の強制終了部を呼び出す．

3 の強制終了部は，実行監視部から通知があった場合に，異常があったデバイスドライバを強制的に終了させ，カーネルに処理を戻す部分である．なお，デッドロックを検出した場合は，ロックの獲得関数を終了させ，ロックの解放待ち処理を終了させる．

提案機構では，監視するデバイスドライバや，監視を行うか否かをユーザが指定することができる．また，監視を行わないときには，オーバヘッドがかからず，監視を行っているときでも他のデバイスドライバの実行には全く影響を及ぼさない．

4.2 監視対象の指定

提案機構では，ユーザが，proc ファイルシステムを用いて，監視するデバイスドライバや，監視するか否かを指定する．提案機構は図 4 の様にファイルシステムの階層の中に `target` と `watching` という仮想ファイルを作る．ユーザは，これらの仮想ファイルを通して，監視するデバイスドライバや監視するか否かを指定する．

`target` は監視するデバイスドライバを指定するための仮想ファイルである．デバイスの名前をユーザが

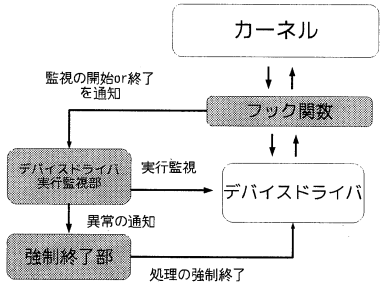

図 3 提案機構の構成
Fig. 3 Architecture of our system.

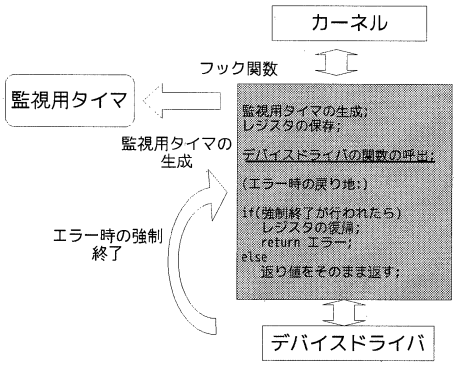

図 5 フック関数
Fig. 5 Hook function.

```
/--bin
 |-local
 |-etc
 :
 |-proc--driver
 :      |-modules
        :
        |-driverwatcher--target
        :              |-watching
```

図 4 ファイルシステム階層中の `target` と `watching`
Fig. 4 Files `target` and `watching` in filesystem hierarchy.

このファイルに書き込むと，提案機構がカーネル空間中から，そのデバイスに対応するデバイスドライバの情報を探し出す．一方，**watching** は，実行を監視するか否かを指定するための仮想ファイルである．実行監視時には，デバイスドライバとカーネルとの間にフック関数が挿入され，監視の必要が無くなると，フック関数が除去される．フック関数を動的に挿入したり，除去することにより，監視が不要になった際には全くオーバヘッドがかからずに済む．

4.3 フック関数

フック関数の主な役割は以下の 3 つであり，図 5 の様な構成になっている．

(1) デバイスドライバを呼び出す直前に，実行監視のためのカーネルタイマを生成する．
(2) 異常が無かった場合は生成したカーネルタイマを削除する．
(3) 異常発生時には障害回復処理を行う．

カーネルがデバイスドライバのインタフェース関数を呼び出した時は，まずフック関数が呼び出される．フック関数では最初，監視用カーネルタイマを設定（監視回数，監視のインターバルなど）する．次に，監視用カーネルタイマを生成し，デバイスドライバの監視を始める．カーネルタイマとは，カーネルが用意している機構で，タイマ割り込みを利用して，指定した時刻にハンドラ関数を呼び出すものである．ハードウェア割り込みを利用しているので，デバイスドライバが無限ループに陥っても，ハンドラ関数に処理を移すことができる．

監視用カーネルタイマを生成したら，その時点の全てのレジスタの値を保存しておく．提案機構では，異常発生時にデバイスドライバからフック関数に強制的にジャンプさせることがある．そのため，フック関数に戻ってきた時には，レジスタの値がデバイスドライバの処理途中のままで，フック関数において処理を続けることができない可能性がある．そこで，デバイスドライバの呼び出し前にレジスタの値を保存しておき，障害回復機構によって強制的にジャンプさせられた場合には，保存しておいたレジスタの値を復帰させる．

これらの前処理をした後に，元々のデバイスドライバのインタフェース関数を呼び出す．

呼び出したデバイスドライバの関数から処理が返ってきた時，以下の 2 つの可能性が考えられる．

(1) デバイスドライバの処理が正常に終了し，呼び出し元のフック関数に戻ってきた．
(2) 提案機構の障害回復機構が働き，デバイスドライバの処理が強制的に終了させられ，フック関数に戻ってきた．

1 の場合は，監視用カーネルタイマを削除し，デバイスドライバからの返り値をそのままカーネルに返して終了する．

2 の場合は，呼び出し前に保存しておいたレジスタの値を復帰させて，処理が続けられる状態にする．そして，問題の起こったデバイスドライバを使用できないようにした上で，カーネルにエラーを返す．監視対象のデバイスドライバが使用できないようになると，提案機構のフック関数がデバイスドライバの関数を呼び出さずに，カーネルへエラーを返して，障害を起こしたデバイスドライバには，二度と処理が渡らないようになる．

4.4 実行監視部

デバイスドライバの実行を監視する部分である．カー

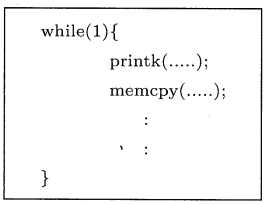

図 6 カーネル関数を呼び出している無限ループ
Fig. 6 An infinite loop that calls kernel functions.

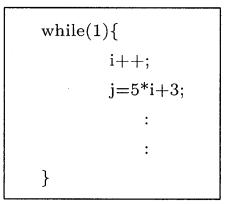

図 7 カーネル関数を呼び出していない無限ループ
Fig. 7 An infinite loop that does not call kernel functions.

ネルタイマを使い，一定時間経ってもデバイスドライバからの処理が戻ってこない場合は，強制的にデバイスドライバを終了させる．

ほとんどのデバイスドライバは，一定時間後には必ず終了するものであり，長時間終了しない場合は，異常を起こしている可能性が高い．本研究では，異常かどうかの判断に，その経験則を利用している．

4.5 強制終了部

実行監視部によって，異常が発生していると判断されると，強制終了部が，異常が発生したデバイスドライバを終了させる．強制的にデバイスドライバを終了させる処理は，以下の 5 つの処理から適当なものを選んでいる．

(1) スリープしているデバイスドライバを終了させる
(2) カーネル関数を呼び出している無限ループを終了させる
(3) カーネル関数を呼び出していない無限ループを終了させる
(4) スピンロック獲得関数を終了させる
(5) セマフォ獲得関数を終了させる

カーネル関数を呼び出している無限ループとは，図 6 のように無限ループ内でカーネル関数を呼んでいるもののことである．一方，カーネル関数を呼び出していない無限ループとは，図 7 のように無限ループ内で計算だけを行っているか，関数を呼び出していてもデバイスドライバのものだけという無限ループのことである．後者の無限ループを強制的に終了させる処理は，関数の実行途中でも，フック関数へジャンプさせてしまう．そのため，この処理を，`memcpy` の様な重要なデータを変更するかもしれないカーネル関数に適用して，関数の実行途中で強制終了させると，カーネル内の状態の整合が取れなくなってしまう恐れがある．そこで，無限ループを強制終了させる処理を，カーネル関数を呼び出しているものと，そうでないものとで，安全のために分けている．

提案機構では，表 1 の様な判断によって，適切な処理を選んでいる．表中の番号は，上の強制終了処理の説明の番号と対応している．

強制的にデバイスドライバを終了させると決断した時に，デバイスドライバのスレッドが CPU において実行中である場合（表 1 の Yes の列）を考える．その時に，デバイスドライバのモジュールの関数が実行中であれば，カーネル関数を呼び出していない無限ループにあると見なして，3 の処理を行う．一方，デバイスドライバのスレッドでカーネル関数が実行中であれば，カーネル関数を呼び出している無限ループにあると判断して，2 の処理によって，デバイスドライバを終了させる．スピンロックの獲得関数が実行中であれば，スピンロックの獲得関数を終了させる専用処理である 4 を実行すればよい．セマフォの獲得関数はデッドロックを起こすと，スリープし続けるので，CPU によって実行中であることは考えられない．

逆に，デバイスドライバのスレッドがスリープしている時（表 1 の No の列）を考える．デバイスドライバ中の関数や，カーネル関数がスリープしているスレッドで実行中であれば，終了させる処理は，1 のスリープしている関数を終了させる処理になる．スピンロックの獲得関数は，常に CPU で実行され，スリープすることはないので，この場合は考えない．また，セマフォの獲得関数がスレッドで実行中であれば，その関数を終了させる専用の処理である 5 を実行する．

4.6 強制終了処理の詳細

デバイスドライバの強制終了は，図 8 の様にスタックに保存されている帰り番地を書き換え，強制終了処理にジャンプさせることで行う．図 9 がその概要である．スタックを書き換え，`terminator` という中継用の関数に飛ばす．この `terminator` は，提案機構で定義された関数で，あらかじめ保存しておいたフック関数のエラー時の帰り番地にジャンプさせるという中継処理を行っている．

以下，各場合における処理について述べる．

- デバイスドライバがスリープしている場合
  カーネルスレッドがスリープしている場合，重

表 1 終了処理の選択
Table 1 Selection of a termination operation.

ドライバが実行中の関数	ドライバのスレッドが実行中である	
	Yes	No
ドライバの関数	3	1
カーネルの関数	2	1
スピンロックの関数	4	-
セマフォの関数	-	5

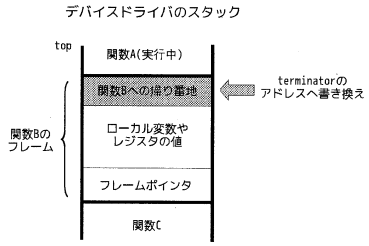

図 8 スタックの書き換え
Fig. 8 Rewriting a stack.

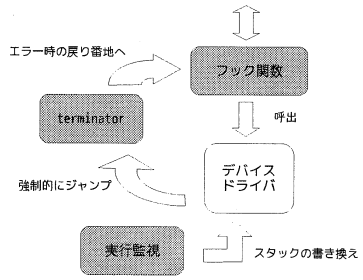

図 9 デバイスドライバの強制終了
Fig. 9 Enforced termination of device drivers.

要なレジスタはプロセスの情報を持つ構造体の中に保存されている．そこでスタックトップのアドレスを持つ esp レジスタの値をこの構造体の中から取得する．そして，そのスタックを順に辿り，監視対象のデバイスドライバへの帰り番地を `terminator` のアドレスに書き換えるようにする．

カーネルスレッドがスリープしている場合，関数呼び出し履歴の先頭には，スレッドの状態を復元する処理が入る．そのため，`while(1){}`のような無限ループがデバイスドライバにあっても（他の関数を呼び出さず，絶対に `return` しない関数があっても），この方法で強制的に終了させることができる．

- カーネル関数を呼び出している無限ループにある場合

  `printk` や `memcpy` などのカーネルが用意した関数は，十分にデバッグが行われている．そのため，処理が終了しない可能性は極めて低く，デバイスドライバの無限ループに戻ってくることが期待できる．そこで，スタックを辿っていき，帰り番地がデバイスドライバのところがあれば，そこを `terminator` に書き換えてしまう．すると，カーネル関数が終了する際に，`terminator` へジャンプするので，この無限ループから抜けることができる．

- カーネル関数を呼び出していない無限ループにある場合

  カーネル関数を呼び出していない無限ループにあると，カーネル関数を呼び出している無限ループと同じ方法では終了させることができないことがある．図 7 のループの様に，スタックの帰り番地を書き換えても，その帰り番地に `return` せず，同じ関数内でループするだけという可能性があるからである．そこで，この場合は，別の方法を使って終了させる．

  カーネルタイマが実行されるのは，タイマ割り込みのコンテキストであった．割り込みが実行された際，x86 系プロセッサでは，CPU が自動的に `eflags`，`cs`，`eip` レジスタを順にスタックに退避させる．また，Linux においてカーネル空間では，`cs` レジスタの値は一定である．そこで，提案機構では `cs` レジスタの値を目印にして，スタック中の `eip` レジスタの値のある場所を探す．そして，その値を書き換えることによって，カーネルタイマ終了時に，`terminator` にジャンプさせ，無限ループから抜けている．

- スピンロックのデッドロックにある場合

  カーネル関数を呼び出していない無限ループを終了させる処理とほぼ同じである．ただし，`eip` レジスタの値を探すところで，その値がスピンロックの獲得関数中かどうかを確認した上で，書き換えるようにしている．

- 割り込み可能なセマフォのデッドロックにある場合

  提案機構では，割り込み可能なセマフォのデッドロックにも対応している．セマフォはデッドロックが起こっても，スリープするだけなので，OS 自体がフリーズすることはない．しかし，プロセスが終了できなくなってしまうので対応することにした．セマフォのデッドロックが起こっていると実行監視部が判断した場合，獲得関数が終了するように提案機構がシグナルを送っている．そのため，割り込み不可能なセマフォでデッドロックが起こると，シグナルを送っても無視されるので，この方法では終了させることができない．割り込み不可能なセマフォによるデッドロックから回復させるためには，プロセスの状態を `sleep` から `wait` に強制的に書き換えるなどの処理が必要になると考えられる．

5. 性能評価

5.1 実験環境

全ての実験は，CPU: Intel Pentium4 3.0GHz，メモリ: 1G バイトのマシンを使用した．OS は，Debian etch，カーネルのバージョンは Linux 2.6.20 である．

5.2 障害からの回復

ブロック型デバイスとキャラクタ型デバイスのドライバ，それぞれに5つのバグを埋め込み，提案機構により障害から回復できるかどうかを実験した．

ブロック型デバイスとして，Linuxの標準のループバックデバイスを，キャラクタ型デバイスとして，自作の仮想デバイスを使った．埋め込んだバグは以下の様なものである．

(1) `while(1){}` の様に，他の関数を呼び出さずに，同じ関数内でループし続ける．
(2) `while(1){printk(...);}` の様に，同じカーネル関数を呼び出し続ける．
(3) ドライバをスリープさせ続ける．
(4) スピンロックのデッドロックを起こす．
(5) セマフォのデッドロックを起こす．

表2に実験の結果をまとめた．1列目の番号は，上の説明のバグの番号を表す．結果より，キャラクタ型デバイスでは，全てのバグから回復できたが，ブロック型デバイスでは，幾つか回復出来ない場合があった

障害から回復できない理由を調査してみたところ，バグを埋め込んだ関数の役割が原因であることが分かった．バグをブロック型デバイスのドライバのread, writeのインタフェース関数であるrequest関数や，その関数から呼び出されている関数に埋め込んだ場合は，障害から回復出来なかった．たまたま，2，5のバグがrequest関数に埋め込まれていたため障害回復に失敗したようである．更に実験してみたところ，全てのバグで，request関数以外の部分に埋め込んだ場合は，障害回復に成功したが，request関数に埋め込むと障害回復に失敗した．これは，request関数が呼び出される際は，カーネルタイマが無効になっていて，実行監視部として使っていたカーネルタイマが実行されず，障害回復が行われなかったためだと考えられる．カーネルタイマが無効になっている理由は調査中であり，request関数への対応は今後の課題である．

5.3 提案機構によるオーバーヘッド

バグの無いデバイスドライバを監視させ，その実行時間を測定することによって，提案機構によるオーバヘッドを測定した．

5.3.1 仮想デバイスのドライバのオーバヘッド

Linuxの標準となっているramdiskのデバイスドライバを監視させ，そのオーバヘッドを測定した．

表2 障害からの回復
Table 2 Recovery from faults of device drivers.

バグ	障害回復の成否	
	キャラクタ型	ブロック型
1	成功	成功
2	成功	失敗
3	成功	成功
4	成功	成功
5	成功	失敗

表3 仮想デバイスのオーバヘッド
Table 3 Overhead imposed on operations to a virtual device.

繰り返し (万回)	監視有り (sec)	監視無し (sec)	オーバヘッド (%)
1	0.067	0.066	-
5	0.316	0.286	10
10	0.626	0.569	10
50	3.14	2.84	16.7
100	6.27	5.68	10.4
500	31.3	28.7	8.94
1,000	62.9	56.5	11.3
5,000	312	290	7.78
10,000	629	570	10.4

ramdiskは，ハードディスクの様なデバイスとは異なり，メモリアクセスのみで高速に動作する．そのため，提案機構によるオーバヘッドの上限を測定するのに最適であると考え，評価に使用した．

測定のために，ramdiskの同じ領域に対して256バイトのread，5バイトのwriteを繰り返すプログラムを作った．提案機構による監視がある場合と無い場合，それぞれについて，繰り返し回数を1万回から1億回まで変化させ，プログラムの実行時間を測定した．その結果が表3である．

表より，繰り返し回数が1万回の場合を除いた時のオーバヘッドの平均は，11%となった．これが，提案機構による監視のオーバヘッドの上限であると考えられる．しかし，本プログラムの様にキャッシュを最大限に生かせるようなデバイスアクセスは稀であり，一般的なデバイスドライバは，遅い物理デバイスの制御をする必要もある．そのため，この結果は使用者の体感的なオーバヘッドとは一致しないはずである．そこで，次の実験では，より一般的なデバイスドライバに近い，物理デバイスを制御するドライバのオーバヘッドを測定した．

5.3.2 実デバイスのドライバのオーバヘッド

LinuxのCD-ROMの標準デバイスドライバを監視させ，そのオーバヘッドを測定した．

CD-ROMのデータを10Mバイトから100Mバイトまで読み込むテストプログラムを作り，それぞれの場合における実行時間を測定した．実験は，最初の1回を除く，2回目から4回目の読み込み時間の平均をとった．これは，2回目以降のデバイスアクセスは，メモリキャッシュの機構が働くため，1回目に比べて，高速にアクセスできるようになるからである．

結果を表4にまとめた．結果より，実デバイスのドライバの場合は，デバイスの処理時間が支配的になり，ほとんどの場合で，オーバーヘッドは観測できなかった．一部，実行時間に差が出たところも，メモリキャッシュのヒット率やデバイスのシーク時間などによるものではないかと推測される．

表 4 実デバイスのオーバヘッド
Table 4 Overhead imposed on operations to a real device.

データ (Mbyte)	監視あり (sec)	監視無し (sec)	オーバヘッド (%)
10	3.69	3.70	0.0
20	6.72	6.72	0.0
30	9.28	9.71	-4.3
40	12.4	12.6	-1.27
50	15.4	15.4	0.05
60	17.9	14.5	23.0
70	21.0	20.6	2.09
80	23.4	23.4	0.00
90	25.8	26.0	-0.82
100	28.3	28.7	-1.47

6. おわりに

本研究では，デバイスドライバのデッドロックや無限ループといったバグを検知し，その障害から回復する機構を設計し，実装した．

提案機構はLKMとして実装したため，既存のデバイスドライバやカーネルには手を加えることなく，手軽に使用することができる．また，監視対象のデバイスドライバにだけオーバヘッドがかかり，それ以外のデバイスドライバには影響を与えない．

性能評価では，提案機構におけるオーバヘッドは高々11%程度であり，実際的なデバイスドライバであれば，無視できる程であることを示した．また，障害回復機能の実験では，ほとんどの場合で目的を達成できることを確認した．

一方で，改善の余地があることも明らかになった．ブロック型デバイスのrequest関数では提案機構が働かず，障害回復が出来なかった．これは，カーネルタイマが無効になっていて，実行監視部が働かなかったことが原因であり，タイマ割り込み以外の割り込み（キーボードやマウス等）を用いれば，対応できるはずである．また，割り込み禁止になっているような所で，無限ループやデッドロックが起こった場合も，タイマ割り込みが無効になるため，提案機構では回復できないと考えられる．この場合は，キーボードやマウスなど一切の割り込みが禁止になっているので，割り込みに頼った提案機構では対応できない．このような場合の対応については別の手法を組み合わせることが必要であり，今後の課題としたい．

参考文献

1) R. Short: Vice President of Windows Core Technology, Microsoft Corp. private communication, 2003.
2) A. Chou, J. Yang, B. Chelf, S. Hallem, D. Engler: An Empirical Study of Operating System Errors. In *Proceeding of the 18th ACM Symposium on Operating Systems Principals*, October 2001.
3) M. M. Swift, B. N. Bershad, H. M. Levy: Improving the Reliability of Commodity Operating Systems. *ACM Transactions on Computer Systems*, 22(4), November 2004.
4) M. M. Swift, M. Annamalai, B. N. Bershad, and H. M. Levy: Recovering Device Drivers. In *Proceedings of the Symposium on Operating Systems Design and Implementation*, December 2004.
5) F. M. David, J. C. Carlyle, and R. H. Campbell: Exploring Recovery from Operating System Lockups. In *Proceedings of 2007 USENIX Annual Techinical Conference*, June 2007.
6) J. N. Herber, H. Bos, B. Gras, P. Homburg, and A. S. Tanenbaum: Failure Resilence for Device Drivers. In *Proceedings of the 37th Annual IEEE/IFIP International Conference on Dependable Systems and Networks*, June 2007.
7) G. C. Hunt: Creating User-Mode Device Drivers with a Proxy. In *Proceeding of the USENIX Windows NT Workshop*, August 1997.
8) V. Ganapathy, A. Balakrishnan, M.M. Swift and S. Jha, Microdriver:. A New Architecture for Device Drivers, In *Proceeding of HotOS XI*, May 2007.
9) M. Accetta, R. Baron, W. Bolosky, D. Gohrb, R. Rashid, A. Tevanian, and M. Young: Mach: A New Kernel Foundation for UNIX Development. In *Proceedings of the Summer 1986 USENIX Conference*, July 1986.
10) BeOS MAX Homepage: http://www.beosmax.org/
11) F. Zhou, J. Condit, Z. Anderson, I. Bangrak, and R. Ennals: SafeDriver: Safe and Recoverable Extensions Using Language-Based Techniques. In *Proceedings of the Symposium on Operating Systems Design and Implementation*, November 2006.
12) G. Hunt, J. Larus, M. Abadi, M. Aiken, P. Barham, M. Fahndrich, C. Hawblitzel, O. Hodson, S. Levi, N. Murphy, B. Steensgaard, D. Tarbiti, T. Wobber and B. Zill: An Overview of the Singularity Project. Microsoft Research Technical Report MSR-TR-2005-135, 2005.
13) J. Mitchell: JavaOS: Back to the Future. In *Proceedings of the Symposium on Operating Systems Design and Implementation*, October 1996.