Vol.2018-OS-142 No.10
2018/2/28

GPUを用いたOSレベルでの障害検知

尾崎 雄一[1]　山本 裕明[1]　光来 健一[1]

概要：システムに障害が発生すると，その上で動作しているすべてのサービスが影響を受けるため，サービス提供者やシステム管理者，ユーザにとって大きな損失となる．そのため，システムの障害をできるだけ早く検知して，障害からの復旧を行う必要がある．システム障害の検知を行うには監視対象システムの外部から監視を行う手法と内部で監視を行う手法が挙げられる．外部から監視を行う場合，障害が発生するまで検知できないことが多く，障害についての詳細な情報を取得するのも難しい．一方，内部で監視を行う場合は障害の予兆を検知するのに十分な情報が得られるが，障害によっては検知システムが影響を受ける．そこで本研究では，監視対象システムの内部にあり，かつシステム障害の影響を受けにくいGPUを用いて障害の検知を行うGPUSentinelを提案する．GPUSentinelでは，GPU上の検知プログラムがメインメモリを参照することによってシステムの状態を取得し，それを基に障害検知を行う．CUDAのマップトメモリ機能を用いてメインメモリ全体にアクセスできるようにするために，Linuxカーネルと GPUドライバに修正を加えた．また，検知プログラムの作成を容易にするために，自動アドレス変換を行うツールを開発した．GPUSentinelを用いて実験を行い，システムのプロセス一覧を取得できることを確認した．

キーワード：障害検知, GPU, LLVM

1. はじめに

近年，情報システムは機能の多様化，大規模システムを開発する技術の発達等により複雑化してきている．その結果，システム障害の発生も増加傾向にある．大きなシステム障害だけでもここ8年間で倍増している [1]．システム障害が発生するとサービスが停止してしまい，サービス提供者やシステム管理者，サービスの内容によってはユーザにも損失を発生させてしまう．そこで，損失を減らすためにシステム障害をできるだけ早く，正確に検知する必要がある．

従来の障害検知手法として，監視対象システムの外部から監視を行う手法と内部で監視を行う手法がある．システムの外部から監視する場合，ホストに対する死活監視を行って障害を検知するのが一般的である．監視対象が仮想マシン（VM）の場合には，VMの外側で仮想ハードウェアの状態を監視することもできる．この手法にはシステム障害時でも監視が行えるという利点があるが，詳細なシステム情報の取得ができないという欠点がある．一方，システムの内部で監視する場合，OS上で検知プログラムを動作させたりOSに検知プログラムを組み込んだりすることになる．この手法は，システム内部の詳細な情報を用いて精度の高い障害検知を行うことができるが，監視対象システム内で検知プログラムを動作させているため障害の影響を受けやすいという欠点がある．

そこで，本稿では監視対象ホストのGPU上で検知プログラムを動作させて障害検知を行うシステムGPUSentinelを提案する．GPUは監視対象システムを動作させているCPUやメインメモリから独立してプログラムを実行することができるため障害の影響を受けにくい．また，監視対象ホスト内部で動作するため，障害検知にOSレベルの詳細な情報を利用可能である．GPUSentinelでは，障害検知のためにGPUからメインメモリを参照し，OSのデータを解析する．例えば，監視対象システムのCPU使用率や空きメモリ容量を取得することで障害やその予兆を検知することができる．システムに関する様々な情報が障害検知には必要となるが，GPUの多数のコアを用いることで障害検知の並列化が可能である．

我々はGPUSentinelをCUDAを用いて実装した．メインメモリをGPUから参照できるようにするために，CUDAのマップトメモリ機能を用い，メインメモリ全体をGPUメモリにマップできるようにするために，Linuxカーネルと NVIDIAのGPUドライバに修正を加えた．また，GPU上で動作する検知プログラムはOSの仮想アドレスをGPUアドレスに変換する必要があるため，LLVMを用いてプロ

[1] 九州工業大学
Kyushu Institute of Technology

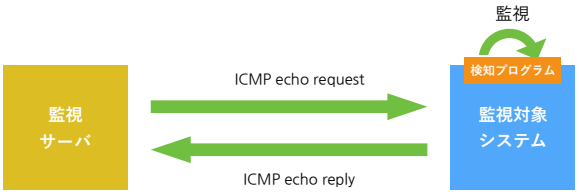

図 1: 従来の監視手法

グラム変換を行うツールを開発した．GPUSentinel を用いて，監視対象システム上で動作しているプロセスの一覧をメインメモリから取得して表示する検知プログラムを作成し，標準的なプロセス数では実用的な実行時間であることを確認した．

以下，2 章で従来の障害検知手法の問題点についてを述べる．3 章で GPUSentinel を提案し，4 章でその実装について述べる．5 章で GPUSentinel を用いて動作する検知プログラムの性能を調べた実験について述べる．6 章で関連研究に触れ，7 章で本稿をまとめる．

2. 障害検知

近年，情報システムはクラウドコンピューティングなどの多様な技術を取り入れることで多機能化を実現しており，システム構造が複雑化している．その結果，システム障害の発生件数も年々増加してきている [1]．システム障害が発生するとサービス提供者やシステム管理者，ユーザに大きな損失が生じてしまう．オンラインバンキングを提供するシステムを例に挙げると，システムに障害が発生した場合，サービス提供者は障害から復旧するまで利益を得られなくなる．また，システム管理者は障害の原因を特定してシステムを復旧しなければならない．システムのユーザにも取引ができないことによる損失が発生する可能性がある．

このような損失を軽減するためには，障害から素早く復旧しサービス提供を再開することが重要である．そのためには障害をできるだけ早く検知する必要がある．できれば，障害によってサービスが提供できなくなる前に予兆を検知するのが望ましい．また，障害をできるだけ正確に検知することも必要である．障害を誤検知してしまうとシステム管理者が発生していない障害の原因を調査しなければならず，サービスを停止させなければならなくなる場合もあるためである．その結果，サービス提供者やユーザの損失に繋がってしまう．

従来のシステムの監視方法として，図 1 のように監視対象システムの外部から監視する方法と内部で監視する方法がある．監視対象システムの外部からネットワーク経由で監視する例として，ICMP を用いたホストの死活監視があげられる．この監視手法により少なくとも監視ホストと監視対象システムの OS の一部は正常に動作していることが

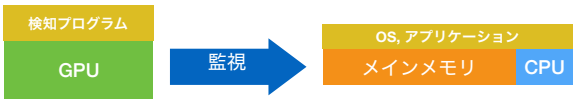

図 2: GPUSentinel のシステム構成

確認できる．また，監視対象システムが提供しているネットワークサービスに定期的に接続することでも死活監視を行うことができる．この場合はサービス単位で正常に動作しているかどうかを確認できる．これらの手法の利点は，監視対象システムの内部に依存することなくブラックボックスとして障害検知ができることである．しかし，監視対象システムの内部状態はわからないため，障害発生時に詳細な情報を取得するのが難しい．そのため，応答がない時にどのような障害が発生しているかがわからないという点が欠点である．

監視対象システムが VM 内で動作している場合は，VM の外側でより詳細な情報を取得することが可能である．検知システムは仮想化システムから VM の仮想ハードウェアの状態を取得することができ，その情報を障害検知に利用することができる．例えば，VM の CPU 使用率，ディスクアクセス量，ネットワークパケット数などを取得することで異常を検知しやすくなる．しかし，それ以上の詳細な情報は得られず，これらの情報を利用できるのは VM を使用している場合に限られる．

一方，監視対象システムの内部で監視する場合，検知プログラムを OS に組み込んで動作させたり，OS 上で動作させて OS から情報を取得したりする．そのため，検知プログラムがシステム内部の詳細な情報を利用することができ，より精度の高い障害検知を行うことができる．しかし，監視対象システムの OS に障害が発生した場合に検知システムが正常に機能しなくなったり，機能を停止したりする．例えば，システムのメモリが足りなくなった場合には検知プログラムが強制終了させられる場合がある．また，監視対象システムと同じホストで検知プログラムを実行するため，システム全体の性能低下を引き起こす可能性がある．

3. GPUSentinel

本稿では監視対象システムに搭載された GPU 上で検知プログラムを動作させてシステムを監視し，障害を検知するシステム GPUSentinel を提案する．システムの概要図を図 2 に示す．GPUSentinel では検知プログラムが GPU を占有して自律的に動作する．検知プログラムは障害が発生する前のシステム起動時に起動され，その後 GPU 上で動作し続ける．GPU からシステムを監視するために，検知プログラムはシステムのメインメモリ上のデータを解析する．これにより，OS レベルの詳細な情報を利用して精度の高い障害検知を行うことが可能となっている．

GPU は監視対象システムを動作させている CPU やメインメモリから独立して動作するため，システム障害が発生しても GPU 上の検知プログラムは動作し続ける可能性が高い．監視対象システムが GPU を使う場合でも，別途，GPU を用意することで GPUSentinel が動作可能である．GPUSentinel では，GPU 上の検知プログラムはメインメモリに格納されているデータから見つけられる障害は検知可能である．ただし，GPU はメインメモリ以外にはアクセスできないため，その他のハードウェア障害の検知は難しい．

GPU は多数の演算コアを有しているため，同時に複数の監視処理を行うことが可能である．この特徴により，さまざまな障害の予兆を並列して調べることができる．また，それぞれの監視処理についても複数のコアを用いて並列化することで高速化することができる．GPU でシステムの監視を行うことにより，メモリや PCI Express の帯域への影響はあるものの CPU 性能には影響を与えないようにすることができる．

GPUSentinel で検知可能な障害の例として，システムリソースの枯渇による障害が挙げられる．例えば，メモリを大量に使用するプロセスがあったり，メモリリークを起こしている場合にはメモリを確保できなくなる．また，プロセスが大量に生成されると様々なリソースが不足する．このような障害を検知するために，GPUSentinel の検知プログラムはメモリの空き領域とページキャッシュの容量から利用可能なメモリ残量を求めたり，プロセスリストをたどることでプロセス数を調べたりすることができる．これらの値が閾値を下回ったり超えたりした場合に障害として検知することができる．

4. 実装

我々は Linux 4.4 と NVIDIA ドライバ 375.66 に修正を加え，CUDA 8.0 と LLVM 5.0 を用いて GPUSentinel を実装した．

4.1 メインメモリのマッピング

メインメモリの内容を GPU から参照する際には，DMA を用いてメインメモリのデータを GPU メモリに転送するのが一般的である．例えば，システム側で/dev/mem の内容を読み出して OS のデータを DMA 転送することができる．しかし，システム障害が発生してシステムが正常に動作しない状況に陥ると，このような DMA 転送を正常に行うことができなくなる可能性がある．

そこで，GPUSentinel では CUDA が提供するマップトメモリ機能を用いて GPU 側から自律的にメインメモリにアクセスできるようにする．マップトメモリはプロセスのメモリページを GPU のアドレス空間にマップして，GPU 上のプログラム（GPU カーネル）から参照可能にする機能である．マップした領域に GPU カーネルがアクセスすると透過的に DMA 転送が行われる．障害が発生する前にメインメモリを GPU アドレス空間にマップしておくことにより，障害によりシステムが機能しなくなっても GPU カーネルはメインメモリを参照することができる．

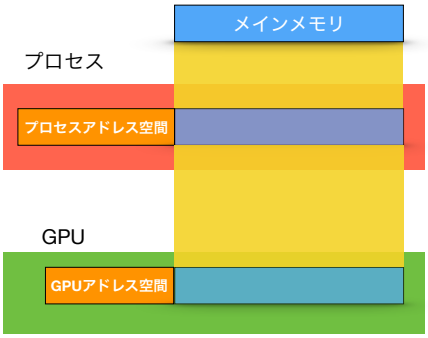

図 3: マップトメモリを用いたメインメモリのマッピング

マップトメモリ機能を利用するために，GPUSentinel はまず，図 3 のようにメインメモリをプロセスのアドレス空間にマップする．これは CUDA ではプロセスのメモリページしか GPU アドレス空間にマップできないためである．マップしたメインメモリを GPU アドレス空間にマップする際には，ページアウトされないように CUDA によってすべてのメモリページがピン留めされる．そのため，単純にメインメモリ全体をプロセスにマップするとシステムの空きメモリがなくなる．プロセスにマップしたメモリページがすべてロックされて使用中になるためである．ちなみに，/dev/mem を mmap システムコールでマップすれば，メモリ領域を使用中にしないようにマップすることができる．しかし，このようにしてマップしたメモリ領域は特殊な扱いがなされており，ピン留めすることができない．

そこで，GPUSentinel では Linux カーネルのメモリ管理に修正を加え，/dev/pmem という特殊なデバイスファイルを用意した．/dev/pmem は mmap システムコールでマップされる時に，メモリページの参照カウンタを増やさないようにすることで，メモリが使用中にならないようにする．GPUSentinel では，プロセス自身がマップされたメモリを使用するわけではないので，メモリページの参照カウンタが 0 でも問題にはならない．また，このようにしてマップされたメモリページをピン留めする際にもロックを行わないようにする．一方，/dev/pmem をマップすることで GPUSentinel 以外のプロセスがメインメモリを参照できるとセキュリティ上の問題が生じるため，メモリ保護を設定することでプロセスからは参照できないようにしている．

このようにしてマップした後でピン留めされたメモリページに対して，ピン留めを正常に解除し，アンマップで

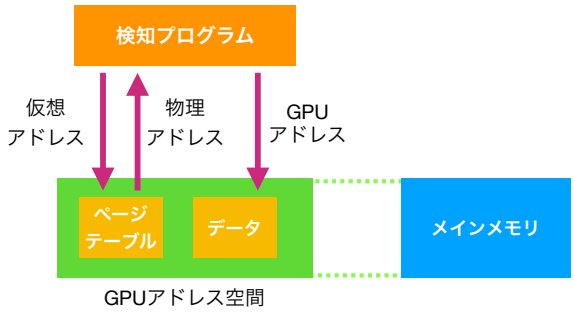

図 4: アドレス変換の流れ

きるようにするために Linux カーネルだけでなく NVIDIA ドライバにも修正を加えた．通常，ピン留めを解除する際にはメモリページがアンロックされるため，/dev/pmem をマップした場合にはアンロックしないようにする．また，アンマップする際には参照カウンタが減らされるため，同様に参照カウンタを変更しないようにする．NVIDIA ドライバの大部分はソースコードが非公開であるが，メモリをピン留めする部分は公開されているため，その部分だけ修正した．

CUDA のマップトメモリ機能の制限を回避するために，GPUSentinel は sysinfo システムコールの偽装を行う．CUDA ではメインメモリ全体のサイズより少し小さなサイズのメモリしか GPU アドレス空間にマップすることができない．これはすべてのメモリページをピン留めした結果，システムが動作しなくなるのを防ぐためだと考えられるが，GPUSentinel は実際にはメモリページのピン留めを行わないため不要な制限である．そこで，CUDA プログラムを起動する時にだけ LD_PRELOAD を用いて sysinfo システムコールをフックし，メインメモリのサイズとして少し大きい値を返すことでメインメモリ全体をマップ可能にした．

4.2 自動アドレス変換

GPU 上で動作する検知プログラムはメインメモリから OS のデータを取得する際にアドレスを変換する必要がある．検知プログラムは仮想アドレスを用いて OS のデータにアクセスするが，仮想アドレスのままでは GPU アドレス空間にマップされたメインメモリにはアクセスできないため，GPU アドレスに変換してからアクセスを行う．そのために，図 4 のように，まずメインメモリ上にある OS のページテーブルを用い仮想アドレスを物理アドレスに変換し，それを GPU アドレスに変換することになる．このアドレス変換を検知プログラムがデータを取得するたびに記述するのは冗長かつ煩雑である．

そこで，このアドレス変換を透過的に行えるようにするフレームワークである LLView を開発した．LLView は検知プログラムを LLVM を用いてコンパイルし，生成された中間表現を変換することで自動アドレス変換を実現する．LLVM の中間表現では，メモリからデータを読み込む箇所で load 命令が使用される．この load 命令の直前でアドレス変換を行う関数を呼び出し，変換されたアドレスを用いて load 命令を実行するように命令を置き換える．アドレス変換を行う関数は引数のアドレスが OS カーネルのアドレスではない場合には渡されたアドレスをそのまま返す．これにより，マップされたメインメモリにアクセスする時以外は GPU のローカルメモリにアクセスすることができる．

例えば，検知プログラムをコンパイルして次のような中間表現が得られたとする．

```
%1 = load i64, i64* %jiffies
%2 = udiv i64 %1, 250
```

この中間表現では OS カーネル内の 64 ビットのグローバル変数 jiffies の値をローカル変数%1 に読み込み，250 で割って%2 に格納している．LLView を用いるとこの中間表現は次のように変換される．

```
%1 = bitcast i64* %jiffies to i8*
%2 = call i8* @g_map(i8* %1)
%3 = bitcast i8* %2 to i64*
%4 = load i64, i64* %3
%5 = udiv i64 %4, 250
```

この中間表現では jiffies のアドレスを 8 ビット整数のポインタにキャストして%1 に格納し，それを引数として g_map 関数を呼び出してアドレス変換を実行する．g_map 関数から返されたアドレスを元の 64 ビット整数のポインタにキャストし，そのアドレスにあるデータを%4 にロードしている．

LLView は中間表現の変換に Pass を用いる．Pass は中間表現対して最適化を施すための LLVM の仕組みである．LLView は中間表現中に load 命令を見つけると，メモリからの読み込みを行おうとしている変数とその型を取得する．そして，それらの情報を用いて bitcast 命令と g_map 関数を呼び出す call 命令を生成し，load 命令の直前に挿入する．その後，変換後のアドレスを用いてデータを読み込む load 命令を新たに生成し，元の load 命令を削除する．その際に，元の load 命令によってデータが格納されたローカル変数を使っている命令をすべて書き換え，新しい load 命令によってデータが格納されたローカル変数を使うように変更する．上の例では，udiv 命令が使っていた%1 が%4 に置き換えられている．

アドレス変換を行う g_map 関数は基本的には，OS のページテーブルを参照して仮想アドレスを物理アドレスに変換する．対象が Linux の場合の最適化として，以下の場合にはページテーブルを参照せずに変換を行う．メインメモリがダイレクトマッピングされている領域内のアドレスの場

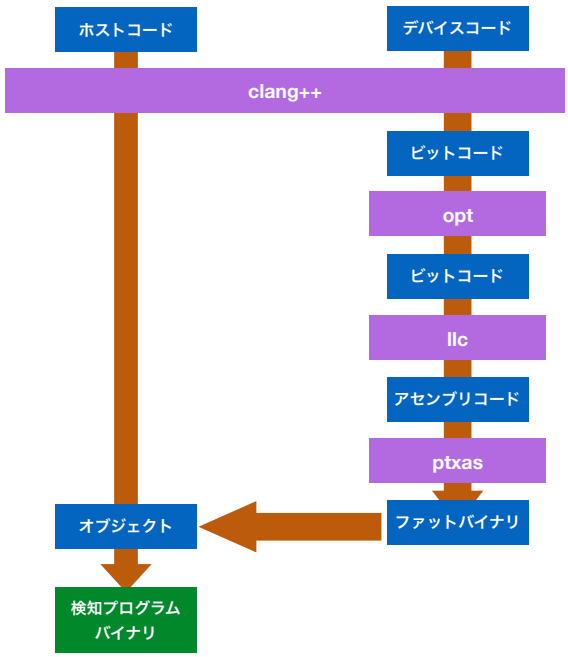

図 5: LLView を用いたコンパイルの流れ

合は，仮想アドレスからその領域の先頭のアドレスを引くことで物理アドレスに変換する．カーネルのテキスト領域がマップされている領域内のアドレスの場合も同様にして変換する．物理アドレスから GPU アドレスへの変換は，メインメモリがマップされた GPU アドレス空間のアドレスに物理アドレスを足すことで行う．

検知プログラムが OS カーネル内のグローバル変数にアクセスできるようにするために，LLView は中間表現におけるカーネル変数をカーネル内で用いられている仮想アドレスに置換する．カーネル変数と仮想アドレスの対応は System.map から取得できるため，`jiffies` に対する `bitcast` 命令は Pass を用いて以下のように置換される．

```
%1 = inttoptr i64 -2113892352 to i8*
```

このアドレスは 16 進数で `ffffffff82009000` である．仮想アドレスは 64 ビット整数として扱われるため，`bitcast` 命令が `inttoptr` 命令に変更される．

LLView を用いて CUDA プログラムをコンパイルする流れは図 5 のようになる．検知プログラムは CPU 上で動作するホストコードと，GPU 上で動作するデバイスコードに分けて記述されるため，まずホストコードを LLVM の clang++でコンパイルしてオブジェクトファイルを生成する．次に，デバイスコードを clang++でコンパイルし，中間表現が格納されたビットコードを生成する．このビットコードに対して opt で Pass を適用してプログラム変換を行う．続いて，llc を用いて変換後のビットコードからアセンブリコードを生成し，CUDA の ptxas および fatbinary を用いてファットバイナリと呼ばれる埋め込み可能なバイナリを生成する．最後に，ホストコードから生成されたオブジェクトファイルにファットバイナリを埋め込み，CUDA バイナリを生成する．

```
__global__
void kernel(void *d_mapped, unsigned long size,
     unsigned long phys_base, unsigned long pgd)
{
    g_init(d_mapped, size, phys_base, pgd);

    while (1) {
        //障害検知処理
    }
}
```

図 6: 検知プログラムの雛形

```
#include <linux/sched.h>

struct task_struct *p;

p = &init_task;

do {
    printf("pid: %d\n", p->pid);
    printf("comm: %s\n",kstr(&p->comm));

    p = list_entry(p->tasks.next,
                   struct task_struct,
                   tasks);
} while (p != &init_task);
```

図 7: プロセスリストをたどる検知プログラム

4.3 検知プログラムのプログラミング

障害検知を行う GPU カーネルは図 6 のように記述する．GPU カーネルの引数には，メインメモリがマップされている GPU アドレス，メインメモリのサイズ，OS カーネルがロードされている物理アドレス，ページテーブルのアドレスが渡される．これらの情報は GPU カーネルでは取得できないため，プロセスが OS カーネルから取得して GPU カーネルに渡す．これらの情報は `g_init` 関数に渡され，自動アドレス変換の初期化処理に使われる．そして，無限ループする while 文の中に障害検知処理を記述する．

GPUSentinel では，検知プログラムは Linux カーネルのヘッダファイルを用いてカーネルモジュールのように記述する．図 7 はプロセスリストをたどって情報を取得する検知プログラムの例である．このコードはカーネル変数の `init_task` を開始点として循環リストをたどりながら，プロセス ID とプロセス名を表示している．ここで，プロセス名は `kstr` 関数を用いて明示的にアドレス変換を行っている．これは，文字列が文字列型のポインタであるため，LLView では自動アドレス変換が行われないためである．

CUDA プログラムは C++としてコンパイルされるため，GPUSentinel は C 言語で記述された Linux のヘッダファイルを用いた検知プログラムをコンパイルできるようにするためのマクロを提供する．このマクロは Linux の変数名

```
pid: 0
comm: swapper/0
pid: 1
comm: systemd
pid: 2
comm: kthreadd
pid: 3
comm: ksoftirqd/0
```

図 8: 検知プログラムの出力結果

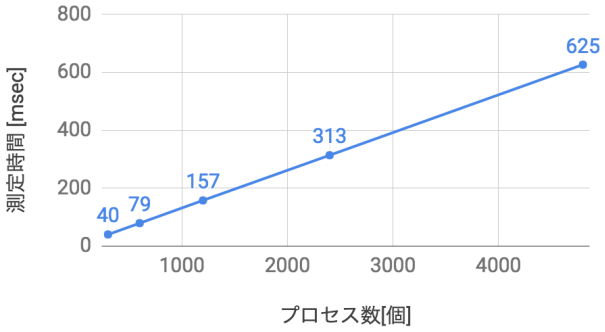

図 9: 検知プログラムの実行時間

と予約語の衝突を回避するために変数名を置換する．例えば，Linux で用いられている `new` という変数名は C++の `new` 演算子と衝突するため，変数名を `_new` に置換する．また，標準 C ライブラリとも Linux の変数名や型名が衝突するため，これもマクロを用いて置換することで衝突を回避する．一方，マクロで回避できない問題については Linux のヘッダファイルを書き換えることで対処した．例えば，C++で必要とされるキャストがヘッダファイルにない場合にはキャストを追加し，`void` 型のポインタ演算は `char` 型のポインタ演算に変更した．

5. 実験

GPUSentinel を用いて，GPU 上の検知プログラムが正常に動作するかどうかを確認し，OS の情報の取得にかかる時間を調べた．実験には，Intel Core i7，16GB のメモリ，GeForce GT610 の GPU を搭載したマシンを用いた．このマシンでは Linux 4.4.67，NVIDIA ドライバ 375.66，CUDA 8.0.61 を実行させた．まず，OS のプロセスリストをたどって，プロセス ID とプロセス名を表示する図 7 の検知プログラムを実行した．その実行結果の一部を図 8 に示す．この図に示すように，プロセス ID が 0 のプロセスから順にすべてのプロセスの情報を表示されることが確認できた．

次に，監視対象システムで動作させるプロセス数を変化させながらプロセス情報の取得にかかる時間を測定した．その結果を図 9 に示す．この結果より，プロセス数に比例して情報取得に時間がかかることがわかる．プロセス数が極端に多いと情報取得にかなりの時間がかかっているが，数百程度の標準的なプロセス数では実用的な時間で済んでいる．

6. 関連研究

OS 内で障害を検知するカーネルレベル障害検知機構 [2] が提案されている．障害検知をカーネル空間で動作させることにより，オーバヘッドを軽減しつつ多様な障害検知を可能にしている．この機能はカーネルタイマを使って定期的に障害検知処理を行い，カーネルデータから CPU 使用率，メモリ使用率，プロセス情報を取得する．しかし，OS 内が正常に動作しなくなると障害検知が行えなくなる．

カーネル間メモリ監視による障害検知機構 [3] では，Orthros を用いて 1 台の計算機上で 2 つの OS を動作させ，監視対象となる ActiveOS のメモリ上に存在するカーネルパラメータをもう一方の BackupOS から監視する．そのために，2 つの OS 間に共有メモリを用意し，ActiveOS が書き込んだ監視対象のパラメータを BackupOS が読み取る．ActiveOS に障害が発生しても BackupOS で障害を検知できるが，障害の種類によっては BackupOS も機能しなくなる．また，監視対象 OS を修正する必要もある．

これまでに，攻撃の影響を受けずにシステムを安全に監視するための様々な機構が研究されてきた．それらの機構を用いることで，システム障害の影響を受けにくい障害検知システムを構築することも可能である．Copilot[4] は専用の PCI カードを用いて DMA でカーネルメモリの内容を取得し，リモートホストで監視を行う．SPE Observer[5] は Cell/B.E. プロセッサを用いて，OS が動作する PPE から隔離された SPE 上で監視システムを実行し，DMA でカーネルメモリを取得する．しかし，いずれも専用ハードウェアや普及していないプロセッサを用いるため，一般的な利用は難しい．

Intel の汎用 CPU を用いて，システムから隔離された環境で監視システムを動作させることもできる．HyperGuard[6] は CPU のシステムマネジメントモード（SMM）を用いて，メインメモリから隔離された SMRAM 上で監視システムを実行する．HyperCheck[7] は SMM で NIC のドライバを動作させ，メインメモリの内容をリモートホストに転送して監視を行う．HyperSentry[8] は SMM を用いることでハイパーバイザ内で安全に監視システムを実行する．一方，Flicker[9] は Intel TXT を用いて安全に監視システムを実行する．しかし，SMM は低速であり，監視中はシステムを停止させる必要がある．TXT でも監視システムを実行する CPU コア以外を停止させる必要がある．

VM を用いたシステムの場合には，VM イントロスペクション [10] を用いることで，VM の外側からシステムの詳細な情報を取得することができる．これにより，障害に強く，検知能力の高い障害検知システムを構築することができる．しかし，VM を用いない場合には利用することが

できず，ハイパーバイザやホスト OS の障害検知には利用できない．GPUSentinel は VM イントロスペクションを GPU に適用したものである．

7. まとめ

本稿では，監視対象システムの GPU 上で検知プログラムを動作させて障害検知を行うシステム GPUSentinel を提案した．GPUSentinel では，GPU からメインメモリ上の OS データを監視することにより障害を検知する．Linux カーネルおよび GPU ドライバに変更を加えることでマップトメモリ機能を用いてメインメモリ全体をロックすることなく GPU にマップできるようにした．また，検知プログラムを作成する際の負担を軽くするために，LLView と呼ばれる自動アドレス変換のためのフレームワークを開発した．GPUSentinel を用いて検知プログラムを実行し，OS の情報が取得できること，および，実用的な性能が得られることを確認した．

今後の課題は，実際に障害を検知する検知プログラムの作成と，検知処理の並列化による高速化である．また，検知した障害を外部に通知する機構も必要である．将来的には，OS のデータを書き換えることで障害からの復旧を行えるようにすることも検討している．

参考文献

[1] 松田晃一，目黒達生：情報システムの障害状況 2017 年前半データ，SEC journal, Vol. 13, No2，pp. 52–58, 2017.
[2] 岩間 響子，毛利 公一，齋藤 彰一 ：多様な障害へ対応したカーネルレベル障害検知機能の提案と実装，情報処理学会研究報告，Vol.2016-OS-136，No.7，pp. 1–9，2016.
[3] 松下 馨，岩間 響子，瀧本 栄二，毛利 公一，齋藤 彰一：多重 OS 実行環境に置けるカーネル間メモリ監視による障害検知機構の実装，情報処理学会研究報告，Vol.2017-OS-139，No.2，pp. 1–8，2017.
[4] N. Petroni, Jr., T. Fraser, J. Molina, and W. Arbaugh: *Copilot - a Coprocessor-based Kernel Runtime Integrity Monitor,* in Proc. Conf. USENIX Security Symp., 2004.
[5] K. Kourai and T. Nagata: *A Secure Framework for Monitoring Operating Systems Using SPEs in Cell/B.E.,* In Proc. Pacific Rim Int. Symp. Dependable Computing, pp.41–50, 2012.
[6] J. Rutkowska, R. Wojtczuk, and A. Tereshkin: *Xen 0wning Trilogy,* Black Hat USA, 2008.
[7] J. Wang, A. Stavrou, and A. Ghosh: *HyperCheck: A Hardware-Assisted Integrity Monitor,* in Proc. Int. Symp. Recent Advances in Intrusion Detection, pp. 158-177, 2010.
[8] A. Azab, P. Ning, Z. Wang, X. Jiang, X. Zhang, and N. Skalsky: *HyperSentry: Enabling Stealthy In-context Measurement of Hypervisor Integrity,* in Proc. Conf. Computer and Communications Security, pp. 38-49, 2010.
[9] J. McCune, B. Parno, A. Perrig, M. Reiter, and H. Isozaki: *Flicker: An Execution Infrastructure for TCB Minimization,* in Proc. European Conf. Computer Systems, pp. 315-328, 2008.
[10] T. Garfinkel and M. Rosenblum, *A Virtual Machine Introspection Based Architecture for Intrusion Detection,* in Proc. Network and Distributed Systems Security Symp., pp. 191-206, 2003.