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

2004－OS－95 (11)
2004／2／27

仮想マシン環境を用いたOSのテスト環境の開発

†吉田 幸二 ‡早川 栄一

近年ソフトウェアの開発において、開発期間の短縮が求められている。このような要求は、オペレーティングシステム(以下、OS)の開発についても同様に存在する。OS の開発において、最も時間のかかる工程の一つにテスト、デバッグの工程がある。しかし、OS を対象としたテストツールは既存のものがないので、デバッガを用いてテストが行なわれている。そこで、本研究では仮想マシン環境を用い、割込みや入出力の制御、割込みログを用いた OS の動作の再現、テストケース定義スクリプトを用いたテストの自動化といった特徴をもつテスト環境の開発を行った。また、本テスト環境を使用して、動作の再現や OS のセマフォ部分のテストを行ないシステムの評価を行った。

Development of an Operating System Test Environment using Virtual Machine Environment

† Koji Yoshida ‡ Eiichi Hayakawa

Recently, shortening of a development period is required in development of software. It also exists similarly about development of an operating system (OS). The test tool for OS does not exist though testing and debugging processes takes many times in software development. In this research a test environment with the feature of automation of the test using control of interrupt or device input and output replay of OS using the interrupt log, and the test case definition script is developed on virtual machine environment. Moreover, this test environment is evaluated using semaphore and the replay of OS behavior.

1. はじめに

近年、ソフトウェアの開発において、製品をリリースするまでの期間の短縮が求められている。

このような早いリリースへの要求はオペレーティングシステム(以下、OS)の開発においても同様に存在する。OS の開発において、テストやデバッグには非常に時間がかかる。OS を対象としたデバッグ環境としては、シミュレータを用いたもの[1]や GDB[2]などがあるが、OS を対象としたテストツールやフレームワークが既存では存在しない。そのため、現在ではデバッグ環境を利用したテストが行われている。

しかし、GDB[2]などのデバッグ環境は、インタラク

†拓殖大学大学院工学研究科
Graduate school of Engineering,Takushoku University
‡拓殖大学工学部情報工学科
Takushoku University

ティブなインタフェースを用いるものが多い。これらの方法では、テストを行なう際に人が動作を見て結果を判断することになり、カバレッジテストのような系統的なテストを実行することが困難でる。また、割込みや入出力は、OS などのプログラムの動作とは無関係に発生する物であり、その発生するパターンは実行するたびに変化し再現性がない。このような環境では、何度も繰返しテストを行なう場合、テスト時の状態が実行するたびに変化するので、同じ条件でのテストができないという問題がある。この問題は既存の実ハードウェア環境で実行されるような、テストツールやデバッグ環境では解決することができない問題である。シミュレータを用いたデバッグ環境では解決することも可能だが、既存のものでは、こういった問題に対するサポートは行われていない。

そこで、本研究の目的として、割込みや入出力の再現ができ、テストを自動的に実行できるテスト環境の開発を行う。

2. 問題分析

OS をテストする時の問題点について述べる。

2.1 割込みと入出力の問題

割込みには、システムコールのようなプログラムの動作に起因する割込みと、入出力のようなプログラムの動作とは関係なく発生する割込みとがある。タイマや入出力などのプログラムの動作とは関係なく発生する割込みは、いつ発生するかわからず、その発生パターンに再現性がない。そのため、OS のテストを行なう場合に見た目上同じ挙動をしていても、割込みが起こるパターンなどの内部の状態は実行するたびに異なる。

OS が行う多くの処理は、割込みを使用して実行される。そのため、バグの中には、実行中のアドレスや時間で割込みが発生した場合などの特定の条件下で発生するようなバグがある。例えば、排他や同期の処理、入出力において割込み禁止を掛け間違った場合に発生する。このようなバグは割込みが発生したアドレスや時間によって正しい動作をする場合と誤った動作をする場合が出てくるので、原因の特定やバグの再現が困難である。

表示関数を用いたデバッグ方法やデバッガとして GDB を用いた場合には、どちらも実ハードウェア上で OS の実行を行うので、このようなバグのデバッグは困難である。また、実ハードウェア上で OS の実行を行わない仮想マシンを用いたデバッガでは、このようなバグのデバッグも可能だと考えられるが、既存のデバッガにはこのようなサポートは行われていない。

そのため、OS の挙動が一定ではなく実行するたびにその挙動が変化してしまい、同じ条件下でのテストが行えないという問題がある。

2.2 テストの結果判断の問題

従来のデバッガが提供するコマンドラインインタフェースを用いてテストを行なった場合、テスト対象が正しい動作を行っているかの判断を、人がトレースしながら行うことになる。

このようなテスト方法では、テストを行なう人がプログラムの動作を停止して確認するため、テストに時間がかかる。また、テストケースが増えた場合や系統的なテストを行なうことが面倒である。そのため、既存のアプリケーションプログラムのテストを対象とするようなテストツールでは、テストの自動化を行なって、手間を軽減している。

そこで、OS のテストについても、テスト結果の判定を自動化し、テストを容易に行える環境が必要である。

3. システムの設計方針

(1) OS の挙動を再現可能な環境の提供

割込みや入出力の挙動も含めた OS の動作を再現可能な環境を提供する。テストやデバッグにおいて、一番重要なことはターゲットの動作やバグを再現することである。そのバグや動作が再現できないような環境では、バグが起こった時にその原因を特定することができない。バグの原因は、何度もバグや動作を再現し、その動作を解析することで特定することができる。

OS のほとんどの処理は、割込みをトリガとして動作する。例えば、プロセスを切替えるディスパッチの処理はタイマから発生する割込みによって動作を行い、アプリケーションプログラムから OS の処理を呼び出すシステムコールもソフトウェア割込みを使用する。このように OS の動作には割込みや入出力がいつ発生したかということが関係している。しかし、割込みや入出力がいつ発生するかはプログラムを実行するたびに変化する。例えば、キーボードからの入力によって発生する割込みを考えた場合、ユーザが行ったキー操作をユーザ自身が、キーを押した時間も含めて再現することはできない。そのため、既存のハードウェア上では、ある特定の割込みや入出力のパターンが来た時だけ起こるようなバグをテスト、デバッグすることが困難である。

そこで、本研究ではこの問題の解決方法として、仮

想マシン環境を用いる方法を提案する。

テスト対象の OS を仮想マシン上で動作させ、仮想マシン上で発生する割込みや入出力の発生をログとして取得する。そのログを基に割込みや入出力を再現することで、OS の挙動を再現する環境を提供する。

(2) スクリプトを用いたテストの自動化

現在のテストは正しく動作しているかどうかの判断を人が行っている。printf などの表示関数を用いた方法では、表示関数の出力を見て動作が正しいかを判断しているし、デバッガを使った方法では対話型のインタフェースを使用し、人がテスト対象の実行を操作、レジスタやメモリの値をダンプして見ることで正しく動作しているかを判断している。GDB には実行するコマンドを順番に記述したスクリプトを使用することもできるが、このスクリプトでは各行に GDB のコマンドが記述されているだけで、コマンドを実行した結果を利用することができないのでテストを自動で行うことはできない。

そこで、テスト結果の判定条件とテストケースを記述するスクリプト言語を提供することで、テストを自動的に行えるようにする。テストの自動化を行うことで、繰返し実行するテストや系統的なテストを容易に実行できるようになる。

(3) バイナリ互換なテスト環境の提供

テストを行うために特別なコードをソースコードに加えてしまうと、実際に動作する時のソースコードとテストを行う時のソースコードが異なってしまう。この変更によってプログラムの動作が変化する場合がある。そのため、本システムでは、対象となる OS のソースコードを変更することなくテストを行える環境を提供することで、実際に動作する時と同じ状態でテストを行えるようにする。

4. システムの設計

4.1 システムの全体構成

本システムは、問題分析で挙げた問題に対して仮想マシンを用いた環境を提供することで解決する。テスト対象の OS を動作させる仮想マシンと仮想マシンの制御、テスト結果の解析などを行うテストツールから構成されている。図 1 に本システムの全体構成を示す。

仮想マシンが提供している、ハードウェアエミュレータ上でテスト対象となる OS が動作する。テストツールは、仮想マシン内の通信・制御インタフェースを用いて、ハードウェアエミュレータの実行制御やメモリ、レジスタ値を取得しテストを行なう。

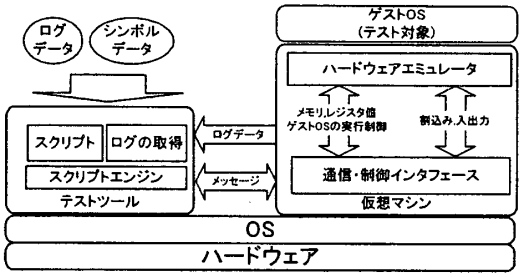


図 1：システムの全体構成

4.2 仮想マシンの設計

次に、本システムで使用する仮想マシンの機能を示す。

(1) ハードウェアエミュレーション機能

ゲスト OS が動作するハードウェア環境のエミュレーションを行う。提供する機能としては、機械語命令、例外、割込み、MMU、レジスタ、タイマ、入出力のエミュレーションを行う。

(2) 通信・制御インタフェース

テストツールとの間でメッセージを送受信する通信インタフェースと、そのメッセージを基にハードウェアエミュレータを制御するインタフェースを提供する。制御可能な機能としては、テスト対象 OS の実行の制御、割込み、入出力の制御、メモリやレジスタの値の取得がある。

4.3 テストツールの設計

本システムで提供するテストツールは次に挙げる機能を提供する。

4.3.1 通信機能

仮想マシンが提供する通信インタフェースを用いて、仮想マシンに接続し、仮想割込みと仮想入出力の発生要求やメモリの値、レジスタの値、ログデータの送信要求とログデータの受信を行う。

4.3.2 テストケース定義スクリプト

テストケースを記述するスクリプト言語を提供することで、ユーザのテストケースの作成を容易にする。このスクリプトで、メッセージとテスト結果の判定条件を記述する。次にスクリプトで記述する情報を示す。

- 発生させる例外，割込みの種類
- 例外，割込みを発生させる位置
- 仮想割込み，仮想入出力のログデータ
- テスト結果の判定条件

4.3.3 シンボル情報の取得

OS は一般的に、C 言語のような高水準言語で記述される部分とアセンブリ言語で記述される部分がある。デバッグやテストを行う時には、C 言語で書かれた部分についてはソースコードのシンボルレベルの情報が使いたい。また、アセンブリ言語で書かれた部分については、メモリのアドレスやレジスタ値などのレベルでの情報を必要としている。

そこで、ソースコードの関数名や変数名などといった、シンボル情報を、テストケース定義スクリプトから使用できるようにすることで、各レベルに対応した抽象度の情報を扱えるようにする。

4.3.4 仮想マシンとスクリプトの実行時間

割込みや入出力を制御するにあたり重要なこととして、何を基準として割込みや入出力を発生させるかということがある。ブレイクポイントならば、プログラムのどこが実行されたかを知るために、実行されたアドレスを基準としている。しかし、割込みや入出力ではプログラム中のどこで発生したかという情報に加えて、いつ発生したかという時間情報が必要となる。本システムで扱える時間は

- 現実の時間
- 仮想マシン内の時間

の二つが存在する。この時間の関係について、図 2 に示すスクリプトと仮想マシンの実行時間を使って説明する。

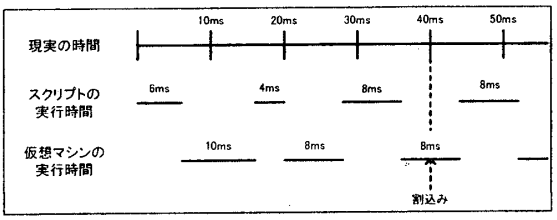


図 2：仮想マシンとスクリプトの実行時間

例えば、図 2 に示すようにスクリプトと仮想マシンがそれぞれ実行していた時、現実の時間で実行開始から 40ms の時に割込みが発生したとする。しかし、仮想マシンの実行時間では、22ms の時に割込みが発生したとなり、スクリプトを実行していた 18ms 分のずれがあることになる。つまり、この割込みを再現を現実の時間で行なった場合は、スクリプトの実行時間の変化で、割込みが発生する時間が変化することになる。

したがって、現実の時間を基準とした場合、ブレイクポイントやスクリプトの変化などで仮想マシンの実行が停止する時間が変化した場合割込みや入出力が起こった時間の整合性を取ることができなくなってしまう。仮想マシン内の時間は、ブレイクポイントなどによって仮想マシンの実行が停止した場合、時間が停止することになる。したがって仮想マシンが停止した場合でも割込みや入出力起こった時間の整合性を取ることが可能である。このことから、割込みや入出力の基準とする時間には仮想マシン内の時間を使用する。また、この時間は仮想マシン内の時間としてだけではなく、実時間とも対応付けられなければならない。

そこで、実行された機械語命令から実行時間をクロック数として算出する方法をとる。このクロック数は RISC のように 1 クロックで 1 命令実行する物や CISC のように命令によって実行にかかるクロック数が決まっているものがあるが、実ハードウェア、仮想マシンにかかわらず一定である。そこで、このクロックを仮想マシンの起動からカウントした数を本システムの時間の基準として用いる。

4.4 システムの詳細設計

本システムの詳細な設計について述べる。

4.4.1 初版の全体構成

本システムの初版においては、既存のソフトウェアを利用し、足りない機能を拡張することで実現する。仮想マシンのハードウェアエミュレータ部分にはオープンソースソフトウェアとして提供されている、IA-32 PC エミュレータの Bochs [3]、通信・制御インタフェースとして GDB スタブ、テストケース定義スクリプトのスクリプトエンジンとして Ruby のスクリプトエンジンを用いる。

割込みや入出力の制御は Bochs や GDB スタブに拡張する形で実装を行う。また、テストに必要な機能は Ruby のライブラリとして実装し、テストケースの定義を Ruby で記述する。図 3 に初版におけるシステムの構成を示す。

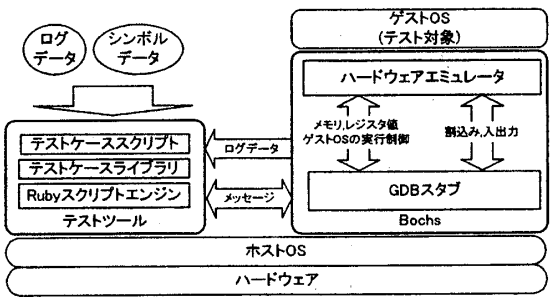


図 3：初版の全体構成

Bochs で提供されている、ハードウェアエミュレータと GDB スタブには、割込みと入出力を制御できるように拡張する。テストツール側は、テストケースの定義、GDB スタブとのメッセージおよびログデータの送受信をするための Ruby で記述されたライブラリを提供する。

テスト対象となる OS は、Bochs 内のハードウェアエミュレータ上で動作を行う。テストは、ライブラリを用いて書かれた Ruby のスクリプトと GDB スタブとが通信を行い、OS の実行制御や割込み、入出力の制御、メモリ、レジスタ値の取得、割込みと入出力のログデータの取得を行う。

4.4.2 仮想マシンの詳細設計

次に示す四つの部分について拡張を行う。

(1) 機械語命令インタプリタの拡張

機械語命令インタプリタ内では、機械語命令を実行する前に割込みを発生や GDB スタブで設定されたブレイクポイントを実行を行うか判断している。そこで、この判断の部分を拡張して、GDB スタブから設定された割込みを発生させるかの判断を追加する。そうすることで、1 命令ごとにどこででも割込みを発生させられるようにする。

(2) 割込みコントローラの拡張

割込みコントローラでは割込みが発生した時の処理を行っている。本システムでは、この部分に何の割込みが発生したかのログを GDB スタブを通してテストツールに伝えるためのプローブを追加する。これによって Bochs 内で発生するすべての割込みや例外をログとして取得することができるようになる。

(3) I/O デバイスエミュレータの拡張

ここでは、キーボードやマウス、HDD、FD、ディスプレイ、イーサネットなど様々なデバイスがエミュレータを通してゲスト OS に提供されている。

そこで、GDB スタブから入力や出力を与えてやることができるようにエミュレータ部分に拡張を行う。

(4) GDB スタブの拡張

GDB スタブでは、gdb 本体と通信を行いリモートデバッグの機能を提供している。本システムでは、GDB スタブを介して Bochs の制御を行う。提供する機能は次に示す八つである。

- 継続実行
- ステップ実行
- ブレイクポイント
- 任意のレジスタ値の取得
- 任意のアドレスに格納されている値の取得
- 割込みの制御
- 入出力の制御
- 割込み，入出力ログの送信

GDBstub でサポートされていない機能は次の三つである。

- 割込みの制御
- 入出力の制御
- 割込み，入出力ログの送信

この機能については GDB スタブの機能拡張と機械語命令インタプリタ、I/O デバイスエミュレータ、仮想割込みコントローラに対して行った拡張と連携して実現する。GDB スタブに追加するコマンドを表 1 に示す。

表 1：拡張する機能

コマンド	説明
vintpoint xx,*addr,clock*	仮想割込みの設定
Vintpoint *addr,clock*	仮想割込みの削除
vintlog:*filename*	ログからの割込みの設定
Vintlog	ログの削除

4.4.3 テストツールの詳細設計

(1) テストケース定義スクリプト

テストケース定義スクリプトの機能は、Ruby のライブラリとして実装される。表 2 に提供するライブラリを示す。

これらのライブラリを使用して、キーボード入力部分のテストを行なうスクリプトの例を図 4 に示す。

図 4 に示したスクリプトでは、入力のデータ列を与え、0x02c9e60 クロック目から入力が開始される。入力される時間間隔を乱数(0~9999 クロックの間)で変化させながら、テストを行なっている。データ列を変えることで、様々なデータや入力の時間間隔でテストを行なうことができる。バグが発生した場合は、その時のログ"io_test.log"を使用して、同じ入力と時間間隔を再現することができる。

このようなスクリプトを記述し、実行することで、

テストを行なうことができる。

表 2：提供するライブラリ

関数名	説明
run()	ゲスト OS を実行
stop()	テストの終了
set_breakpoint(addr)	ブレイクポイントの設定
step(start_addr,end_addr)	ステップ実行
set_interrupt(addr)	割込みの設定
set_memory_value(addr, value)	メモリの値を設定
get_memory_value(addr)	メモリの値を取得
set_register_value (reg_name,value)	レジスタの値を設定
get_register_value (reg_name)	レジスタの値を取得
assert(condition)	判定条件の指定
start_intlog(host_name, port_num, file_nema)	割込みログの取得開始
stop_intlog()	割込みログの取得停止

```
require "TestCase"

target = TestCase.new("localhost",1234)
target.start_intlog("localhost","1235","io_test.log")

keyboard_vector = 0x21        #キーボードからの入力割込みのベクタ番号
test_string = "test key data" #テスト用のキー入力データ
test_address = 0              #キー入力を発生させるアドレス
test_time = 0x02c9ce60        #キー入力を開始する時間
i = 0

#入力の設定
target.set_interrupt(keyboard_vector,test_address,test_time,test_string[i])
target.set_breakpoint(0x00111c98) #ブレイクポイントの設定

loop{
  target.run()
  #アドレス0x0012fb39に入力された値が格納される(と仮定)
  input_val = target.get_memory_value(0x0012fb394)
  #与えた入力とテスト対象が処理したの結果の比較
  target.assert(input_val == test_string[i])
  #割込みを削除
  target.remove_interrupt(keyboard_vector,test_address,test_time)
  test_time = test_time + rand(10000) #次に入力を発生させる時間を設定
  i++
  if(i >= test_string.length)
    break #用意したデータをすべて処理し終わったら終了
  end
  #次の入力を設定
  target.set_interrupt(keyboard_vector,test_address,test_time,test_string[i])
}
```

図 4：スクリプトの例

(2) 仮想割込み，仮想入出力イベントログ

仮想割込み,仮想入出力が発生した時のログを取る。ログは、テキスト形式で一つのイベントが一行に記述され、イベントの種類、発生した時のアドレス、時間、仮想入出力の場合にはその値の順番で記録される。ログデータの中の時間とは、仮想マシンが動作し始めてからのクロック数である。図 5 にログの例を示す。

```
≀
int:20,00117946,0320a56a,00000000,
int:0e,6000a308,03252a80,00000000,
int:20,001098b7,04439293,00000000,
int:21,0011ef7a,04439d20,00000072,
int:20,0010a73d,0443a61b,00000000,
int:80,600331ec,0443a9aa,00000000,
≀
```

図 5：割込みログの例

5. システムの実現

5.1 仮想マシンの実現

仮想マシン部分は、4.4.2 で述べた設計にそって実装を行った。実装した機能は、次に示す四つである。

- 割込みの制御
- 割込みログの送信
- GDB リモートプロトコルへのコマンドの追加
- キーボードからの入力の制御

入出力の制御機能は現在のところ、キーボード入力についてだけ対応している。

5.2 テストツールの実現

テストツールは 4.4.3 で述べた詳細設計にそって、GDB リモートプロトコルを処理する GdbProtocol クラスとテストケースを定義するための TestCase クラスの二つを Ruby のクラスライブラリの実装を行った。

5.3 システムの評価

5.3.1 オーバヘッドの測定

Bochs 上で Linux を動作させ、getpid システムコールの実行にかかる時間と 200 桁の円周率の計算にかかる時間の測定を行った。測定結果を表 4 に、測定に使用した実機のスペックを表 5 に示す。

表 4 の測定結果から、実機での実行と比較して、getpid システムコールで約 14 倍、π の計算時間で約 200 倍の時間がかかることがわかった。

表 4：オーバヘッドの測定

実行を行ったマシン	getpid()の実行時間(μs)	πの計算時間(μs)
Bochs	3.8834	2158.79
実機	0.2887	10.73

表 5：測定環境

CPU	AMD Athlon 1.0GHz
メモリ	256MB(内 32MB を Bochs に割当てた)
OS	Vine Linux 2.6r3

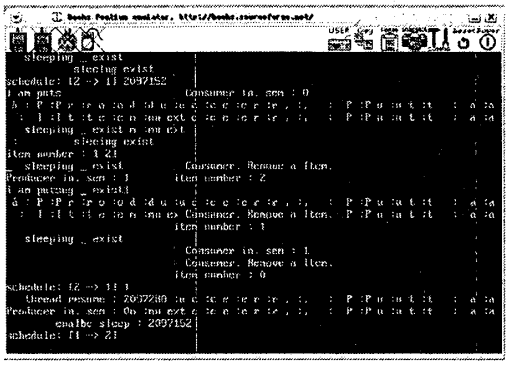
5.3.2 セマフォのテスト

本研究室、で開発した OS である YamanekoOS[5] のセマフォの実装について、本システムを用いてテストを行なった。テストプログラムとして、Thread を使用して生産者消費者問題のプログラムを作成した。同期、排他の処理はセマフォを使ってを行った。手順は、次の通りである。

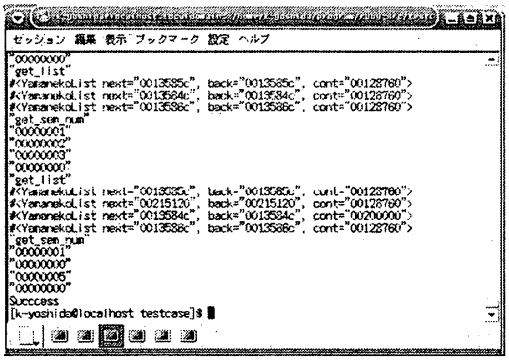
① セマフォ内の割込み禁止区間をずらしてバグを作る
② スクリプトを作成しテストを行い失敗することを確認する
③ バグを修正する
④ 同じスクリプトでテストを行ない合格することを確認する。

図 6 にその実行結果を示す。このテストでは、タイマ割込みを発生させるアドレスを変化させ、セマフォ値が正常な範囲内で変化するかを、チェックしている。もし、セマフォ値が正常な範囲を越えた場合は、その時点で実行を中断し、その時のアドレス(プログラムカウンタの値)を表示するスクリプトを記述し、テストを行なった。

図 7 はテストに合格した時と失敗した時の例を示している。図の中では、上から各セマフォの Wait リストの中、セマフォ値が排他、生産者、消費者の順に表示されている。このテストでは、生産者のバッファのサイズを 5 としているので、正常に動作している限り排他のセマフォ値は 0~1 の間、同期のセマフォ値 0~5 の間で変動するはずである。成功の例を見ると、正しい範囲内でセマフォ値が変化していることがわかる。失敗の例では、セマフォ値が正しい範囲を越えたために実行が[0000fff0]番地で停止されていることがわかる。

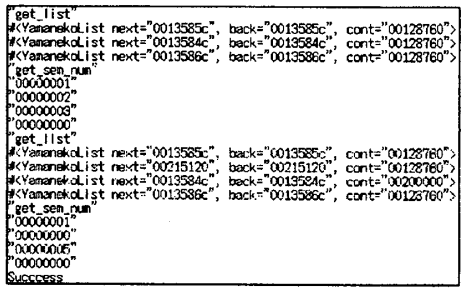


Bochsの画面

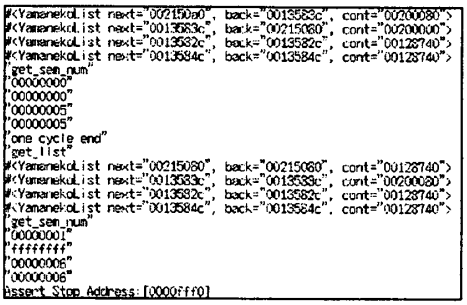


スクリプトの実行結果

図 6：セマフォのテストの実行結果



成功



失敗

図 7：テストの成功と失敗の例

6. 関連研究

関連研究として、シミュレータを用いたデバッガ[1]について述べる。

OS が動作するすべてのハードウェア環境をシミュレートする仮想マシンを用いたデバッガの研究が行われている。この研究では、仮想マシンとして、Simics[4]を用いたデバッグ環境を提案している。図8にこの研究で提案されているデバッグ環境の構成を示す。

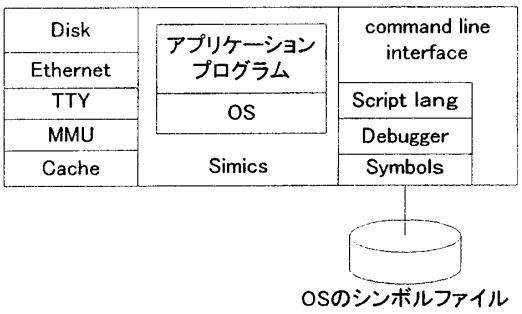


図8：Simics を用いたデバッグ環境の構成

このデバッグ環境では、Simics とコマンドラインインタフェースを持ったデバッガとを組合わせることで、OS のデバッグを行っている。このデバッガは、ブレイクポイントやステップ実行など GDB と同等な機能のほかに、Simics 内の時間をデバッガから操作できる機能も提供している。

このデバッグ環境では、OS が動作するすべてのハードウェア環境をシミュレータで提供しているので、GDB ではトレースができなかった割込み処理の内部でもトレースやデータの取得が可能である。

本システムは、入出力デバイスのエミュレーションも含めた仮想マシンを使用するところは同じだが、

- 割込みや入出力の制御
- 動作の再現
- 割込みなどの発生パターンを含むテストケースの定義
- テストケース定義スクリプトによるテストの　自動化

などの機能により、デバッグよりもテストを対象としたシステムになっている。

7. おわりに

7.1 本研究の成果

次に本研究の成果を述べる。

(1) 割込みと入出力の制御

ハードウェアエミュレート機能を持った仮想マシンを用い、割込みや入出力の制御を行うことで、OS の動作の再現や割込み、入出力が発生するパターンを変化させてテストを行なうことができるようになった。これによって、再現性がない割込みや入出力の関係するバグのテストも行なうことができるようになる。

(2) テストの自動化

テストケースをスクリプトとして定義し、実行を行うことでテストを自動で行なえるようになった。これにより、カバレッジテストのようなテストケースの数が多いテストにおいて、容易にテストを行なえるようになる。

7.2 今後の課題

(1) スナップショット

仮想マシン内のある時点での、すべてのメモリやレジスタなどのイメージをスナップショットとして保存する。保存したスナップショットから実行を再開することで繰返しテストを行なう場合でも、必要な部分だけで実行を繰り返すことができるようになる。

(2) デバッガとの協調

ある一定時間ごとにスナップショットを取得する。テストによってバグが発見された場合、その前で最も近いスナップショットの状態からデバッガを実行できるようにすることで、テストからデバッグへの工程の移行がスムーズになり、デバッグ/テストの効率が向上ガ見込める。

参考文献

[1]Lars Albartsson,Peter S Magnusson,
Using Complete System Simulation for Temporal Debugging of General Purpose Operating Systems and Workloads, MASCOTS 2000
[2]Debugging with GDB,
http://www.asahi-net.or.jp/~wg5k-ickw/html/online/gdb-5.0/gdb-ja_toc.html
[3]Bochs, http://bochs.sourceforge.net/
[4]Simics, http://www.virtutech.com/
[5]山本 茂樹・早川 栄一:仮想マシン環境を指向したオペレーティングシステムの設計と試作, 情報処理学会 FIT2003 B-071, 2003