システムソフトウェアと
オペレーティング・システム 83－4
（２０００．２．２４）

OS/omicron V4におけるデバッグ支援環境の設計

佐藤元信†、 高野了成†、 早川栄一‡、 高橋延匡‡
†東京農工大学工学部 ‡拓殖大学工学部

〒184-8588 東京都小金井市中町2-24-16
E-mail:motop@os-omicron.org

計算機の応用分野が広がるにつれて、システムソフトウェアへの動的な拡張性が求められている。これに対応するため、現在ではマイクロカーネルアプローチによる資源管理の拡張や、カーネルへの動的なモジュール追加機構などが利用されている。実行時にモジュールをリンクする環境、特にタスクとモジュールや資源管理との関係が柔軟に変更できる環境では、性能上のオーバヘッドが少ない反面、ミドルウェアなどの資源管理プログラムのデバッグが難しい。本稿では、このような環境を備えたOS/omicron V4上でのデバッグ支援環境の設計について述べる。この支援環境では動的拡張の際に任意の手続きの置換えや挿入ができる機構を提供する。この機構を利用してデバッグ用の手続きを挿入することで、対象となるタスクがどのモジュールを実行しているか、どのような資源を利用しているかといった情報を収集できる。また、手続き呼出しをトレースし、どのコンテクストでどのモジュールが呼び出されたかという情報を収集でき、ミドルウェアなどのデバッグを支援する環境を提供できた。

Design of Debugging Environment on OS/omicron V4

SATO Motonobu†, TAKANO Ryousei†,
HAYAKAWA Eiichi‡ and TAKAHASHI Nobumasa‡

†Faculty of Engineering, Tokyo University of Agriculture and Technology
‡Faculty of Engineering, Takushoku University

2-24-26 Nakacho Koganei, Tokyo, 184-8588 Japan
E-mail:motop@os-omicron.org

Dynamic extensibility of system software is the important feature of recent system for adapting diverse application. Microkernel or loadable kernel module approach is mainly utilized the extension of systems software. These approaches, however, make debugging the resource management like middleware difficult because the domain of resource management is changed when configurating the system dynamically. In this paper we describe the debug environment for dynamic extensible system software on the OS/omicron V4.

The feature of the environment is following:
(1) Implementing the mechanism for replacing or inserting any procedures when dynamic extension
(2) Collecting information about the context when the module is running on.
(3) Tracing the procedure and binding the procedure call sequences with task context or modules for debugging middleware.

Debugging the middleware or operating system can be supported in our debug environment.

1. はじめに

計算機の小型化やネットワークの普及に伴い計算機の利用形態が多様化している。PDA や家電への組込みシステムのような利用可能な資源の制限が厳しく特定用途に特化したものから、高性能、高信頼性が求められるサーバシステムのようなものまである。オペレーティングシステムには、このような広範囲なシステムに柔軟に対応できることが期待される。

近年では、システムプログラムの拡張性を確保するために動的にカーネル内にモジュールを追加する機構やマイクロカーネル構成を採用するオペレーティングシステムが多くなっている。しかし、マイクロカーネルではタスクを単位として拡張しているので性能の問題や拡張の単位が大きいなどの問題がある。また、このような機構でモジュールを追加できるのはカーネルに対してであり、アプリケーションプログラムを拡張するには共有ライブラリのような別の機構を使わなければならない。

これらの問題を解決するために筆者らは、同じ枠組みでシステムモジュールもユーザモジュールも拡張していくことができる OS/omicron V4 というシステムを研究、開発している。このシステムでは実行時にモジュールをリンクし、同じモジュールを異なるシステム階層で利用できる環境を提供している。しかし、このような実行環境ではどのモジュールが結合し実行されるかは実行時まで分からないので、ソースコードだけからデバッグをすることが困難となる。

こういった状況は動的な拡張機構を備えたシステムのすべてで起こり得る。資源管理などのミドルウェアが動的に拡張される場合、その資源がどのモジュールからどのように利用されているかといった情報が把握しにくくなり、デバッグが困難になる。本稿ではこのような動的に拡張する機構を備えた OS/omicron V4 におけるデバッグ支援環境について述べる。この支援環境は対象プログラムを書き換えることなく、動的な拡張をタイミングとして任意の手続きを挿入することで対象の状況を調べることができる。この機構を利用することでモジュールやタスクの情報も得ることができる関数トレーサの実現や、メモリ資源のモニタの設計について説明する。

2. OS/omicron V4 の概要

この章ではターゲットシステムである OS/omicron V4（以下 V4）の目的と実行環境の特徴について説明する。また、その特徴から発生するデバッグにおける固有の問題について述べる。

2.1 OS/omicron V4 の特徴

V4 のシステム構成は、システムからアプリケーションまでを同じ枠組みで拡張できるように設計されている [1]。これは性能と信頼性のトレードオフに応じて同一のモジュールを実行時に任意のシステム階層に配置できるようにするためである。この機能を実現するために V4 には次のような特徴がある。

(1) 単一 2 次元アドレス空間

V4 ではデータやモジュール間の関係を単にポインタとして表現できるように単一 2 次元アドレス空間を提供する。この環境ではポインタはセグメント ID とオフセットの組で表現され、セグメント内での境界チェックが行われる。セグメントは一塊のコードやデータごとに割り当てられる。

(2) ダイナミックリンク

実行時にモジュール間の結合を決定できるように、V4 ではダイナミックリンク（DL）を利用している。これはプログラムのコンパイル時にリンケージテーブル（LT）と呼ばれる名前とアドレスの対応を示した表を出力し、実行時にダイナミックリンカがその表の内容を確定するとこで実現されている。ロードされたモジュールの LT をもとに名前空間を構成するので、名前空間にはモジュール名、外部変数名、関数名などの言語処理系内の名前も含まれている。ダイナミックリンクの処理の流れを図 1 に示す。

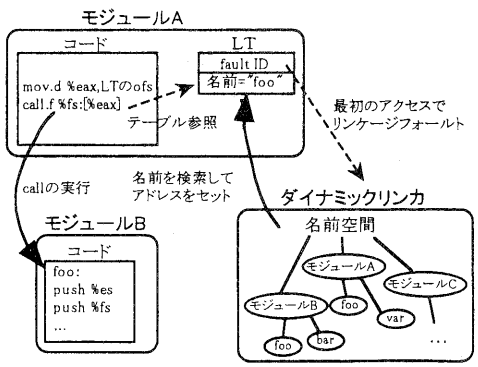

図 1 ダイナミックリンクの処理

(3) Unified Procedure Call（UPC）

V4 ではモジュール間の通信のオーバヘッドを保護の必要性に応じて選択できるように UPC と呼ばれる機構 [2] を提供する。この機構は従来のタスク間通信に代わるもので、記述言語からは手続き呼出しとして利用できるという特徴がある。UPC では特定の手続きを別タスクのコンテクストで実行することができる。

また、UPC ではモジュールのシステム階層が変更されても呼び出す側のプログラムを変更せずに実行でき

るようにモジュール間ゲートという機能も提供する。これは実行時にその UPC 呼出しがシステム階層を横断するかを判断し、横断がある場合はそのままのコンテクストで他のシステム階層の手続きを呼び出すことができる。

(4) タスクと保護、資源の分離

V4 では保護と性能のトレードオフを柔軟に選択できるように Unix の提供するプロセスのような枠組みは提供せず、タスクと保護の単位や資源の管理単位を分離して提供する。このようなモデルでは、資源がタスクに結び付いているのかモジュールに結び付いているのかシステムは判断できないので、資源の解放を自動的に行えるとは限らない。

2.2 デバッグにおける問題点

V4 の特徴に示したように、V4 では実行時に決定される部分が多い。これらのことからデバッグ時には次のような問題点がある。

(1) 実行の追跡

V4 ではシステムとユーザプログラムが密に結合しており、ユーザのコンテクストのままシステムのコードを走行することがある。また、すべてのモジュールは DL により実行時に結合されるので、どのモジュールを利用するかは実行時にならないと分からない。このような環境では、デバッグ対象となるプログラムを停止させて外部から観察するだけではその実行状況を把握することは困難である。

(2) 資源管理

V4 ではモジュール間の結合を実行時に行ったり、資源をプロセスのような枠組みで管理しないことを述べた。通常のライブラリを用いて実現されたアプリケーションプログラムでは、これらの資源はプログラム終了時にランタイムが自動的に解放するので問題ない。しかし、サーバやドライバのようなミドルウェアでは直接資源の操作を行うことがあるので、この場合資源の解放が自動的に行われない。解放はサーバプログラムを記述するユーザの責任になるが、正しく動作しているかを確認するのは困難である。

V4 ではシステム全体が DL により拡張可能なので、すべてのシステム階層でこのような問題が発生する。このような状況は V4 でなくとも、カーネル内にモジュールを追加する機構やダイナミックリンクライブラリを利用できる OS でも発生する。特に資源の生成、消去に関わるようなミドルレイヤを共有したり動的に拡張する場合、その構成や資源の利用状況を把握するのは困難であり、それに対応できるようなデバッグ支援が必要である。

3. 設計方針

先に述べた問題点を解決するために次のような設計方針を設定した。

(1) 手続き呼出しを捕まえる

モジュールの構成が動的に変化する環境では通常記述言語の手続き呼出しを機に他のモジュールへの遷移が発生する。そこで、実行時にどのモジュールからどの手続きが呼び出されたかという情報を取得することで、実際にモジュールがどのように結合されているのかを調べられるようにする。

(2) 同じコンテクストでの動作を追う

V4 では DL により同じコンテクストのまま様々なモジュールのコードを実行するので、デバッグの対象を一つのモジュールに制限できない。そこで、同じコンテクストで実行されたモジュールを追跡したり、逆に異なるコンテクストで実行された同一のモジュールでの実行を追跡することでどのタスクがどのモジュールや資源を利用しているかが分かるようにする。こうすることでバグが他のモジュールに起因するものであっても問題個所の特定が容易になる。

(3) デバッグ環境自体の拡張性を提供する

デバッグ対象となるモジュールが動的に他のモジュールと結合されるので、デバッグ支援環境もこれに合わせて動的に対応しなければならない。また、実行時にどのような情報が見たいかは、バグの状況によって変化する。これに対応するためにデバッグ環境自体を容易に拡張、変更できる機構を提供する。

4. 設計

4.1 デバッグ支援環境の構成

プログラムを再コンパイルせずにデバッグできるように、DL を利用して実行時に対象となるモジュールの制御を奪うことでデバッグ機能を提供する。ここでは制御を奪う関数をスタブ関数と呼び、デバッグ支援環境の提供するスタブ関数の機能を変更することでデバッグ機能を追加できる。ユーザが容易に機能を変更できるよう、支援環境やデバッグ情報の解析ツールはユーザレベルプログラムとして実現される。この構成を図 2 に示す。

(1) マイクロカーネル

マイクロカーネルでは例外の発生を支援環境に通知したり、対象となっているタスクのコンテクストの読

み書きを提供する。また、デバッグ情報の保護や時系列の一貫性を保つために、デバッグ情報を蓄えるためのバッファを提供する。

(2) ダイナミックリンカ

ダイナミックリンカではリンケージフォールトを受け取り、名前により手続きのリンクを行う。指定されたモジュールから発生したDLや特定の手続きへのDLを変更する機能を提供する。リンケージフォールトが発生すると、これらの指定がないか検索し、あった場合はリンク先をスタブ関数に変更する。

(3) デバッグ支援環境

デバッグ支援環境では例外を受け取ることでステップ実行やブレークポイントなどの機能を提供する。また、様々なスタブ関数を提供することでプログラムの変更なしにこれらの機能を使えるようにする。デバッグ支援環境の取得した情報はすべてマイクロカーネルのバッファに保存される。

(4) 解析、表示プログラム

マイクロカーネルに保存された情報を取得し、ユーザに見やすい形で表示するためのツールとなるアプリケーションプログラムである。このプログラムはデバッグしたいユーザが自由に作成することができる。

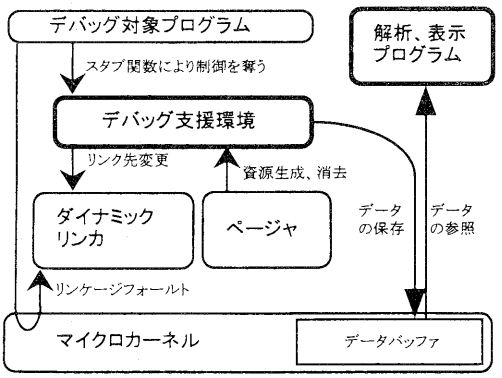

図2 デバッグ支援環境の全体構成

4.2 本方式の特徴

スタブ関数を使うことで実現される本方式では、デバッグ機能を提供する手続きはデバッグ対象プログラムのコンテクストでそのまま実行されるので容易に対象を制御することができる。通常のデバッガでは対象を停止させて情報を盗み見るような形になるが、この方式ならば情報取得に必要なコードを挿入することができるので、必要な情報だけを容易に取り出すことができる。

また、システムプログラムではデバッガで実行を停止させること自体が危険な場合がある。例えば、割込みを禁止しているクリティカルセクションやデバイスを制御している部分で情報を取得するために実行を停止してしまうと正しく動作しなくなることがある。本方式では関数呼出しを行っている部分にしかコードを挿入できないという制限はあるが、対象のコンテクストでそのまま実行されるので停止させずに情報を取得することができる。

4.3 プロセッサ命令レベルの支援

先に述べたようにスタブ関数による支援では関数呼出しの部分での解析しかできないという限界がある。そこでより細かな情報を取得するために、プロセッサ命令レベルでの実行の追跡機能を提供する。この機能はプロセッサの提供するステップ実行例外を利用し、そのハンドラを登録することで実現される。ハンドラでは実行された命令を解析し、そのインストラクションと実効アドレスをバッファに出力する。ユーザはバッファを読み取ることで実際にどのように命令が実行されたかを追うことができる。

また、実行を追うだけでは不十分な場合のために、メモリへのアクセスを監視する機能を提供する。監視したいアドレスの指定を実行時に指定するのは困難なので、この機能を利用する場合はユーザがプログラム中で監視したいアドレスをデバッグ環境に渡す手続きを呼び出す。

トレース機能をどこから利用するかをプログラム中で指定することもできるが、再コンパイルの必要があるので簡単なスタブ関数を提供する。トレース開始、終了、指定した関数だけをトレースするスタブ関数を提供することで、ユーザは任意の関数間のトレースを指示することができる。トレース実行の様子を図3に示す。

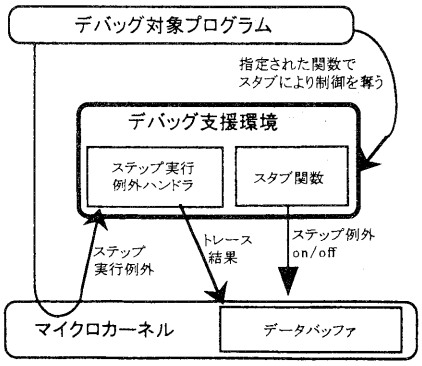

図3 命令トレーサの処理の流れ

4.4 メモリ資源監視ツールの設計

ページャの提供する資源の生成、消去の関数に対してスタブ関数を登録することで、監視ツールのコードを挿入する。監視ツールでは引数を解析することで何の資源が生成、消去されたかを記録する。このとき、どのモジュールがその関数を呼び出したか、どのタスクが呼び出したかも記録することで、モジュール別、タスク別の資源の利用状況を把握する。また、より詳しいメモリの利用状況を知るために、ページフォールト時に呼び出される手続きも監視ツールへ振り替えることで物理メモリの割当て状況も記録する。

これらの状況はユーザレベルのアプリケーションである可視化ツールで見ることができる。可視化ツールでは監視機構に情報を問い合わせることで状況を把握し、その情報に基づいてグラフィカルな表示を行う。このイメージを図4に示す。

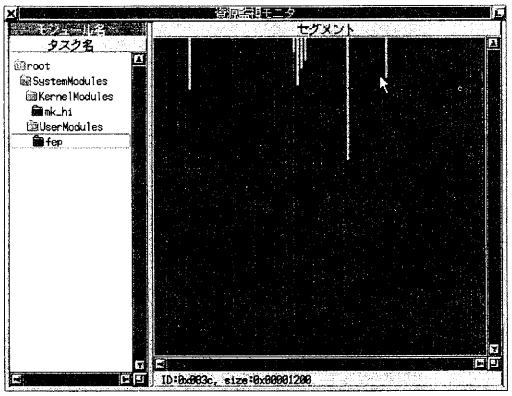

図4 メモリ資源監視ツールのイメージ

4.5 フォールト原因を推定するヒント情報の提供

V4 では一般保護例外などでタスクが停止した場合、ランタイムでそのときの全レジスタなどを表示して情報を提供している。しかし、これだけではバグの原因を追求するには不充分なので、セグメント ID を利用してより詳しいヒントを提示する。

V4 では 2 次元アドレスを採用しているのでポインタはすべてセグメント ID とオフセットの組として表現される。フォールト発生時に実効アドレスを求め、そのセグメント ID について 4.4 の監視ツールに問い合わせることで、どのタスクの生成したセグメントに対する不正アクセスなのか調べることができる。また、スタック、コード、LT、静的データのセグメントであればどのモジュールのものかを特定することができ、どのようなポインタの操作を誤ったのか判定することができる。

5. 実現

設計に基づき、PC/AT 互換機用の V4 上でデバッグ支援環境を実現した。この章ではその詳細と、実行例を示す。

5.1 ダイナミックリンカのフック機構とスタブ関数

従来の V4 のダイナミックリンカでは、木構造で表現された名前空間に対して指定された枝の中から名前を探すことでリンクを行っていた。この枝の指定方法に二種類の指定を追加することで、設計で示した DL 時に指定した関数にリンク先を切り替える機能を実現した。また、リンク先を完全に変更してしまうと本来呼び出すはずだった関数が分からなくなってしまうので、LT のフィールドを一つ利用してその関数を記憶しておくことにした。

スタブ関数では本来呼び出す関数を呼び出すためにスタックフレームのコピーを行う。また、呼び出す前後で指定された手続きを呼び出せるようにすることで簡単にユーザが必要とする機能のスタブを実現できるようにした。このコードの簡略図を図5に示す。

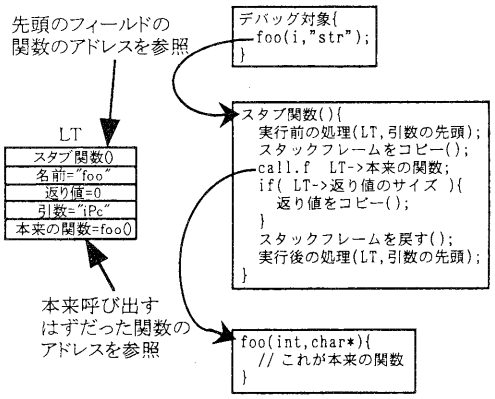

図5 スタブ関数のコード

5.2 プロセッサ命令レベルの支援の実現

x86 系プロセッサの提供するステップ実行例外を利用することで命令トレーサを実現した。デバッグ支援環境ではこの例外を受け取るタスクを生成し、そのタスクの中で実行した命令を解析することで実効アドレスなどの記録を行う。実効アドレスの計算は、セグメンテーションフォールトをデバイス管理が利用するために既に実現されているので、その計算ルーチンをそのまま利用した。

また、x86 系プロセッサの提供するデバッグレジスタを利用することで、指定されたメモリへのアクセスを監視する機能を実現した。x86 系のプロセッサでは、デバッグレジスタで指定した仮想アドレスへのアクセ

スがあった場合に例外を発生する。デバッグ支援環境ではこの例外を受け取り、指定されたアドレスにアクセスが発生した場所を記録する。このときコードの位置だけでなく、どのモジュールのどの手続きでどのタスクが、といった情報も保存しているので実行時の構成が理解でき、プログラマの勘違いなどの発見に役立つ。

5.3 スタブ関数の例

フック機構を使ってスタブ関数を登録することで様々な機能を実現できる。前述の命令トレーサの開始、終了を指示するスタブ関数の他に、モジュールのすべての関数にスタブ関数を登録することで関数トレーサのようなものを実現した。このトレーサでは関数の呼出し、とリタンがあった部分で引数や返り値の情報などを記録する。また、その関数がどのモジュールのものか、引数に指定されたセグメントが何のセグメントかを示すことでデバッグの手がかりを提示する。このデータの出力例を図6に示す。図中の関数名で「@」より後ろの部分がモジュール名である。

```
call WinInit@viewer(0,2,STACK:fffffe8c);
  call ウィンドウを生成する@viewer(STACK:fffffe0c,1,320,200);
    call strcpy@no_res.lib(STACK:fffffc18,DLOCAL:138);
    return strcpy@no_res.lib();
    call strcat@no_res.lib(STACK:fffffc18,STACK:fffffe0c);
    return strcat@no_res.lib();
    call キャンバスの厚みを得る@viewer(void);
    return キャンバスの厚みを得る@viewer();
    call malloc@viewer(64000);
    return malloc@viewer();
    call __UIM_ウィンドウを生成する@hitsuji();
      call __KER_UFMを登録する@hitsuji();
        call strcpy@no_res.lib(STACK:ffff989c,DLOCAL:19c04);
        return strcpy@no_res.lib();
        call strcat@no_res.lib(STACK:ffff989c,DLOCAL:2380);
        return strcat@no_res.lib();
        call __KERNEL_SVC_CREATE@hitsuji();
          call _UFMを登録する@hitsuji();
            call UFM管理表をcreateする@hitsuji();
              call strcmp@no_res.lib(HEAP:3f900,STACK:ffff7de4);
              return strcmp@no_res.lib();
```

図6 関数トレーサの実行例

5.4 実行速度の評価

実現した機能の実行速度を測定した。各測定項目とその結果を次に示す。

(1) ダイナミックリンクのオーバヘッド

ダイナミックリンカに新たな機能を追加したことでオーバヘッドが増加することが考えられる。そこでダイナミックリンクのオーバヘッドを測定した。測定の結果、名前の探索部分で31.2 μ s実行時間が増えていることが分かった。ダイナミックリンクは最初の一度しか発生しないことや、ダイナミックリンクの処理全体の3%程度のオーバヘッドであることから問題になるほどのオーバヘッドではないと判断できる。

(2) スタブ関数のオーバヘッド

スタブ関数ではスタック中の引数領域のコピーを行うので、そのオーバヘッドは引数の大きさに依存する。スタブ関数を登録したときとしていないときの実行時間の差を測定した結果、引数がないときで2.3 μ s、引数が256バイトのときで20.9 μ sであった。

(3) 関数トレーサの実行速度

プログラム中でどれだけ関数呼出しを行っているかに依存するが、関数トレースを行うとこの機能を使わない場合に比べて実行時間が8倍程度になることが分かった。この値は常にトレースを行うには時間がかかりすぎるが、他のスタブと組み合わせて特定の部分だけをトレースすることで充分実用できると考える。

6. まとめ

本稿ではV4におけるデバッグ支援環境について説明した。このデバッグ支援環境では、システムやユーザモジュールが動的に拡張されるというV4の実行環境に着目している。本研究の成果は次のとおりである。

(1) プログラムの変更を必要としないデバッグ支援環境を提供した

ダイナミックリンク発生時にスタブ関数を挿入することで、対象となるプログラムを変更せずに情報を取得できるようになった。

(2) モジュールをまたいでも実行を追跡できる機能を提供した

同一のタスクが異なるモジュールへ移行した場合でも追跡できるようにすることで、システムモジュール内の実行も追跡できるようになった。

今後の課題としては、可視化ツールの未実装部分を実現すること、現在簡単なコマンドとして実装されているので使いやすいGUIの実現などが挙げられる。

参考文献

[1] 佐藤元信, 早川栄一, 並木美太郎, 高橋延匡: "OS/omicron 第4版におけるシステム拡張機構の設計と実現", 情報処理学会研究報告, 98-OS-77, pp.185-190, 1998

[2] 森永智之, 早川栄一, 並木美太郎, 高橋延匡: "単一2次元アドレス空間を提供する拡張可能なマイクロカーネルの開発", 情報処理学会論文誌, Vol.38, No.5, pp.1016-1025, 1997

[3] J.B.Rosenberg: "デバッガの理論と実装", アスキー出版局, 1998

[4] A.S.Tanenbaum: "Modern Operating Systems", Prentice-Hall International, Inc., 1992