システムソフトウェアと
オペレーティング・システム 90－18
（２００２．６．２８）

Lambdaオペレーティングシステムの Linux上での実現

羽山徹* 久住憲嗣† 北須賀輝明‡ 福田晃§

toru.hayama@and.hitachi-sk.co.jp nel@f.csce.kyushu-u.ac.jp, kitasuka@f.csce.kyushu-u.ac.jp,

fukuda@f.csce.kyushu-u.ac.jp

あらまし

オペレーティングシステム (OS) の開発においてデバッグは非常に面倒である。そこで、本研究では開発中のOS Lambdaを Linux上で動かすことが出来るように、Lambdaのハードウェア依存部分をLinuxの機能を利用し、実装を行っている。Lambdaのハードウェア依存部分で求められる機能として、タイマ、割り込み、スレッド管理ブロック、コンテキストスイッチの実装を行う。割り込みにはシグナルを利用し、コンテキストスイッチにはsetjmp, longjmpを利用する。この手法を用い、実装したハードウェア依存部分にLambdaのハードウェア非依存部分を載せて動作させたところ、適切な動作が得られた。

Realization of Lambda Operating System on Linux

Tooru Hayama * Kenji Hisazumi † Teruaki Kitasuka ‡ Akira Fukuda ‡

Abstract

In development of an operating system (OS), debugging is very troublesome. Then, in this research, the hardware dependence portion of Lambda is written by using the function of Linux so that OS Lambda under development can be executed on Linux. Functions called from the hardware dependence portion of Lambda, are a timer, interruption, a thread management block, context switching, and interruption. The signal is used for interruption and setjmp and longjmp are used for context switching. Suitable operation was obtained, when the hardware non-depending portion of Lambda was put on the mounted hardware dependence portion using this technique.

*日立ソフトウェアエンジニアリング（株）
Hitachi software engineering
†九州大学大学院 システム情報科学府
Graduate School of Information Science and Electrical Engineering, Kyushu University
‡九州大学大学院 システム情報科学研究院
Graduate School of Information Science and Electrical Engineering, Kyushu University
§九州大学大学院 システム情報科学研究院
Graduate School of Information Science and Electrical Engineering, Kyushu University

1 はじめに

OSの開発に限らず、ソフトウェアの開発において、コーディングよりもむしろそのコードに対するデバッグに費やす時間が多い。既存の信頼性のあるOS上で開発するOSを動かす事が出来れば、デバッガ等はその信頼性のあるOSの機能を利用できるので、デバッグが非常に楽になる。Linuxにはgdbという優れたデバッガが用意されており、プログラムの追跡が容易である。汎用ＯＳ上で動かすと、テストする際にLinux上のアプリケーションと同様にコンパイルし実行出来る。そこで、本研究では開発中のOS、LambdaをLinux上で動かすことが出来るように、Lambdaのハードウェア依存部分をLinuxの機能を利用し、実装を行う。

2 Lambda OSの特徴

本章では、まず本研究の対象とするOS Lambdaの特徴について述べる[3]。

2.1 Lambdaの特徴

LambdaはOSの保守性や開発効率を向上させるため、マイクロカーネル構成を採用した組み込み向けOSである。

組み込みシステムは、一度出荷したらソフトウェアを変更することがまれである。Lambdaでは柔軟性のあるマイクロカーネル構成で開発した後、出荷時にはOSをモノリシック化することにより、マイクロカーネル構成の欠点である性能の悪さを改善している。

これにより、開発中はマイクロカーネルの開発効率の良さを利用し開発工数を削減し、出荷時にはモノリシックカーネルの性能で実行することが可能となっている。

2.2 マイクロカーネル

マイクロカーネルは以下のモジュールから構成される。

- スレッド・タスク管理機構
- 割り込み管理機構
- タイマドライバ
- メモリ管理機構
- プロセス間通信
- スケジューラ

2.2.1 スレッド管理機構・スケジューラ

タスクは保護の単位であり１個以上のスレッドを持つ。スレッド管理ブロックは、ハードウェア依存部分と非依存部分に分けられる。

- ハードウェア依存部分

  スレッドはそれぞれスタック領域を持つ。その確保を行う関数、スレッドを利用可能な状態に初期化する関数を実装する必要がある。また、コンテキストの保存、回復を行う関数も実装する必要がある。

- ハードウェア非依存部分

  スレッド管理機構は、マイクロカーネル内に全スレッドを管理するテーブルを置き、スレッドIDからスレッド構造体を簡単に得ることが出来るように実装する。また、スレッド間通信の待ちのためにスレッド間通信待ちテーブルも置いており、ディスクリプタからスレッド構造体を得ることができる。マイクロカーネル内には簡単なスケジューラが実装されており、必要に応じてスケジューラサーバを協調動作して、スケジューリングを行う。

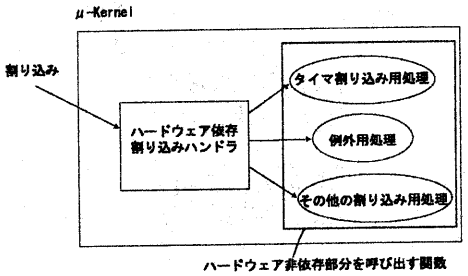


図 1: 割り込み処理の流れ

2.2.2 タイマドライバ

マイクロカーネルは、スケジューラやタイムアウト処理に利用するために、割り込みを発生させることができるタイマをひとつ必要とする。またそのために、ハードウェア依存部分にタイマドライバの実装が必要である。

2.2.3 割り込み処理

割り込みが発生すると、マイクロカーネルは割り込み要因に対応したポートにプロセス間通信で通知する。そして、その通知を割り込み処理スレッドが受け取り、割り込みを処理する。また必要に応じて、割り込み処理スレッド内で割り込みマスクの変更などを行うことができる。割り込み処理の流れは図1のようになる。

3 Lambdaのハードウェア非依存部分と、依存部分とのインターフェース

Lambda のハードウェア非依存部分と、依存部分のインターフェースは以下のようになっている。

3.1 ハードウェア依存部分からみた非依存部分のインターフェース

3.1.1 スレッド管理ブロック周り

・Thread 構造体

それぞれのスレッドに関するさまざまな情報を格納するハードウェア非依存の構造体。

ハードウェア依存部分で利用するのは以下の部分である。

stack : そのスレッドの利用するスタックへのポインタを格納する。

sp : そのスレッドのスタックポインタを格納する。

HWthread 構造体 : スレッド管理ブロックのハードウェア依存部分となる。ハードウェア依存部実装者が自由に定義できる。

3.1.2 コンテキストスイッチ

・Thread *current_context

現在走行中のスレッドが格納されているハードウェア非依存部の変数。

これを参照することで、今どのスレッドが走っているかが分かる。

・Thread *thread_get_current_context()

現在走行しているスレッドを得るハードウェア非依存部の関数。

これを利用すれば、今どのスレッドが走っているかが分かる。(上記の current_context を参照するのと同じ結果が得られる。)

・void sys_thread_sched(Thread *next)

次に走らせるスレッドを選択し、ハードウェア依存部のスレッド切り替え関数を呼び出す。

表 3.1.2 に示すように動作する。この後、登録したスレッドに実際に切り替える関数を呼ぶ。

表 1: sys_thread_sched の動作

引数	処理
NULL	走行中のスレッドをスリープさせる。 最高優先度のスレッドを次に走らせるスレッドとして登録する。
NULL 以外	走行中のスレッドをスリープさせる。 引数で受け取ったスレッドを次に走らせるスレッドとして登録する。

3.1.3 マスク

・glock()

割り込みを禁止する。ハードウェア依存部分の割り込み禁止を行う関数を呼ぶ。

・gunlock()

割り込みを許可する。ハードウェア依存部分のロックを解除する関数を呼ぶ。

3.1.4 初期化周り

・kernel_start()

ハードウェア非依存部の各種機能を初期化して、カーネルを走らせる。

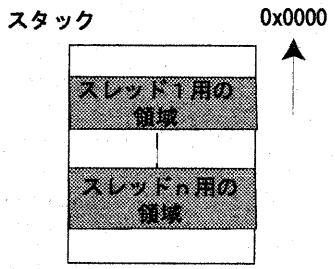


図 2: スレッドごとのスタック領域の様子

3.2 ハードウェア非依存部分からみた依存部分のインターフェース

3.2.1 スレッド管理ブロック

・hw_thread_alloc_stack(*thread)

スレッドごとのスタック領域を図 2 のように確保し、スタックポインタの設定を行う。

・hw_thread_setup_stack(*thread, func)

スレッドを利用できるようにスレッド管理ブロックのハードウェアに依存する初期化を行う。

3.2.2 コンテキストスイッチ

・hw_switch_to_thread(*next)

現在走行中のスレッドの状態を保存し、次に走らせるスレッドに処理を切り替える。

3.2.3 マスク

・hw_interrupt_block()

割り込みを禁止する。

・hw_interrupt_unblock()

割り込みを許可する。

3.2.4 タイマ

・hw_timer_init()

タイマの初期化を行う。

4 ハードウェア依存部分のLinuxへの移植

ハードウェア依存部分で求められる機能は、

・タイマ

・割り込み

・スレッド管理ブロック

・コンテキストスイッチ

・割り込みのマスク

である。

本章ではこれらの機能を実装し、LambdaをLinux上で動作するアプリケーションとして移植するにあたって必要となるLinuxのAPIについて述べる。

4.1 Linux

Linuxとは、UNIX系OSのひとつであり、現在人気を集めている。1991年、Intel80386プロセッサベースのIBM PC互換機用のOSとして、Linus Torvalds氏によって開発された。Linuxが魅力的である理由のひとつとして、ソースコードがGNU一般公共使用許諾契約書(GNU General public License)というライセンスに基づいて公開されており、誰でも自由に調べることが出来るという点が挙げられる。また、豊富なアプリケーションがこのGNU一般公共使用許諾契約書の下で自由に入手できる[5]。

4.2 タイマ

タイマはLambdaのタイマ割り込みによるプリエンプションのために必要である。

Linuxではsetitimer関数というタイマを扱う関数があるのでそれを利用する。

・setitimer関数

設定時間毎にシグナルをプロセスに配送するように設定を行うための関数。

4.3 割り込み

割り込み処理機構のハードウェア非依存部分を動作させるために、割り込み要因を判断して、カーネルのハードウェア非依存部分を呼び出す関数を実装する必要がある。

Linuxにおいてプロセスへの割り込みに使われている機能がシグナルである。

現在走行中のプロセスに対してシグナルが入ると、現在そのプロセスが行っている処理を中断し指定した処理を行う。その処理が終わると、前に処理を中断したところから処理を再開することが出来る。

シグナル発生時の動作は、それぞれのシグナルにデフォルトの動作が設定されているが、その時に呼び出す関数をカーネルに指示しておくことによって、独自に用意した関数で発生したシグナルを処理することができる。これをシグナルを捕捉するといい、捕捉する関数をシグナルハンドラと呼ぶ[4]。

シグナルを捕捉するには次の関数を利用する。

signal(signo, (*func)())

引数で受け取ったsignoでどのシグナルを捕捉するかを指定し、funcでそのシグナルが入ったときに呼び出す関数を指定する。

現在、実際に起こり得る割り込みはコンテキストスイッチの契機として使用するタイマ割り込みだけである。よって、普通ならば割り込みが入ると図1のように動作するが、今回は、タイマ割り込みだけを捕捉することにする。

タイマ割り込みは setitimer で設定したタイマが切れたとき起こり、その時 SIGALRM というシグナルが入る。よって、hw_timer_init() 内で

signal(SIGALRM, コンテキストスイッチを行う関数)

を実行し、SIGALRM を捕捉するための設定を行う。

4.4 コンテキストスイッチ

コンテキストスイッチを行うには、各種レジスタ、スタックポインタなどのハードウェアに依存したコンテキストの保存、回復を行う必要がある。Linux には setjmp, longjmp というコンテキストの保存、復元を行うことができる関数がある。

4.4.1 setjmp, longjmp

引数で受け取る jmp_buf 型の変数に setjmp で状態を保存、longjmp で状態を復元する。

jmp_buf 型

コンテキストスイッチを行うにあたって必要十分なレジスタの値を保存出来る型。

longjmp の概要

引数の jmp_buf 型の変数をもとに、レジスタの値を復元する。

プログラムカウンタも復元されるのでプログラムは setjmp が呼ばれた位置に戻る。

setjmp の概要

表 4.4.1 のように動作する。

表 2: setjmp の概要

	普通に呼ばれた	longjmp で戻ってきた
処理	各種レジスタの値を保存する。	何もしない。
返り値	0	longjmp の第2引数（非0）

実際にはシグナルマスクも保存できる sigsetjmp, siglongjmp という関数を利用する。

4.4.2 スレッド管理ブロック

スレッド管理ブロックのハードウェア依存部分である HWThread 構造体は以下のものをメンバとする。

・sigjmp_buf 型の変数（j_buf）

・そのスレッドの処理が実装された関数へのポインタ

4.4.3 シグナルハンドラ

以下のプログラムが SIGALRM を捕捉する関数のプログラムである。この関数の中では sys_thread_sched 関数を呼ぶだけである。

```
void hw_sig_alrm(int signo)
{
  sys_thread_sched(NULL);
}
```

シグナルハンドラが呼ばれると同時に補足されたシグナルは自動的にマスクされ、この関数を抜けると同時に解除される。

```
void hw_switch_to_thread(Thread *next)
{
 if (sigsetjmp(current_context->hw.j_buf, 1) != 0 ) {
  return;
 }
 else {
  current_context = next;
  siglongjmp(current_context->hw.j_buf, 1);
 }
}
```

図 3: hw_switch_to_thread()

表 3.1.2 にあるように sys_thread_sched 関数を引数 NULL で呼び出すと、スケジューリングした後、ハードウェア依存の実際にスレッドを切り替える関数を呼ぶ。その関数が hw_switch_to_thread 関数である。hw_switch_to_thread 関数は図 3 のように実装する。割り込みが入ってスレッドが切り替わる処理の流れをフローチャートで図 4 で表す。

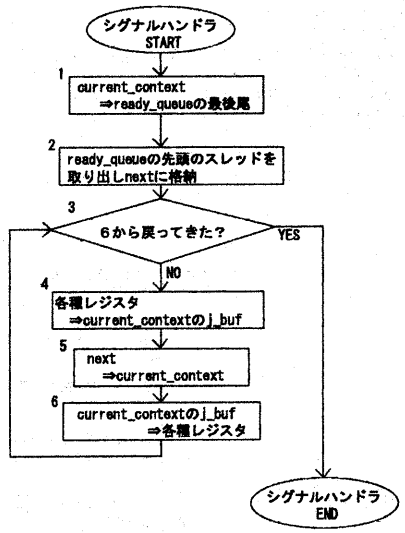
フローチャートの 6 の処理で現在のスタックポインタを、次に走行させるスレッドの会うタックをさすようにする。シグナルハンドラを抜けるときにはスタックポインタがさしているスタックがシグナルが入ったときとは異なる。こうすることによって、スレッドを切り替える事ができる。

4.5 割り込みのマスク

割り込みはハードウェアに依存するので、その割り込みのマスクもハードウェアに依存する。よって Linux 上でも実装する。

Linux にはシグナルのマスクを操作する関数、sigprocmask 関数を利用する。

割り込みはタイマ割り込みのみとしているので、割り込みのマスクの設定はこの関数を利用して SIGALRM のマスクを操作してやればよい。



1, 2:sys_thread_sched内の処理
3, 4:setjmp内の処理
5:hw_switch_to_thread内の処理
6:longjmp内の処理

図 4: シグナルハンドラの流れ

5 発生した問題

sigsetjmp, siglongjmp でスレッドを切り替える場合に、sigsetjmp で保存した状態を siglongjmp で復元するが、初めて処理を行うスレッドに切り替わる場合は sigsetjmp で保存していないので siglongjmp を行ってもスレッドが切り替わらない。

siglongjmp で初めて処理を行うスレッドに切り替えるには以下の情報が必要である。

・そのスレッドの行うプログラムが書かれてあるアドレス。

・そのスレッドの利用するスタックポインタの値。

これらの情報を図 6 のように sigjmp_buf 型の変数に入れてやり、siglongjmp を行えばよい。

Linux では次の定数が用意されている。

・JB_SP：sigjmp_buf 型の変数のスタックポインタが格納されているアドレスへのオフセット値

```
sigjmp_buf j_buf;
buf [JB_PC] =
  そのスレッドが実行するプログラムが書いてあるアドレス;
buf [JB_SP] =
  そのスレッドが利用するスタックの底のアドレス;
siglongjmp(j_buf);
```

図 5: JB_PC, JB_SP の利用

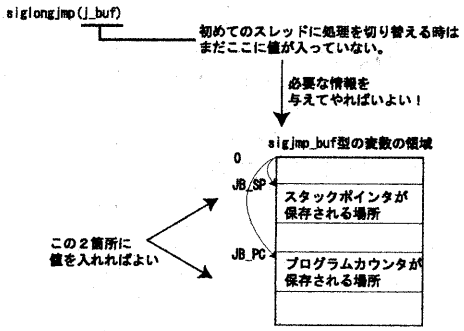


図 6: JB_PC, JB_SP

・JB_PC : sigjmp_buf 型の変数のプログラムカウンタが格納されているアドレスへのオフセット値

これらを利用し、図 5 のように実装することによって初めて実行するスレッドへ処理を切り替えることができる。sigjmp_buf j_buf;

6 まとめ

本稿では、組み込みシステムの開発効率を向上させるために、組み込み用 OS を既存の信頼性のある OS 上で開発することが有効であることを述べた。そこで、開発中の組み込み用 OS、Lambda を Linux 上で動かすことが出来るように、Lambda のハードウェア依存部分を Linux の機能を利用することによって、実装を行った。

Lambda のハードウェア依存部分は以下の部分で構成されていた。

・タイマ
・割り込み
・コンテキストスイッチ
・マスク

本研究で実装を行った Lambda のハードウェア依存部分上で Lambda を実行したところ、正確な動作が得られた。

謝辞

本研究の一部は日本学術振興会・科学研究費補助金（課題番号 : 12480099）の助成を受けている。

参考文献

[1] 福田晃, 最所圭三, 片山徹郎, 中西恒夫: 組込システム向け実行環境の自動生成 – δ プロジェクトの構想 –, 電子情報通信学会技術研究報告, No.726, CPSY 99-125, pp.17 22, 2000.

[2] 久住憲嗣, 中西恒夫, 福田晃 : 組み込み用マイクロカーネル OS の高速化, SWEST3 予稿集, pp.146-149, 2001

[3] 久住憲嗣, 中西恒夫, 福田晃 : 組み込み向けマイクロカーネル OS Lambda の保護機構, コンピュータシステムシンポジウム論文集, No.16 pp.81-88, 2001

[4] W. リチャード・スティーブンス（大木敦雄訳）: 詳解 UNIX プログラミング, pp.170-314, 2000

[5] D.P.BOVET, MARCO CESATI（高橋浩和, 早川仁監訳, 岡島順治郎, 田宮まや, 三浦広志訳）: 詳解 LINUX カーネル, pp.1, 2001