序　文

ディジタル家電や携帯電話など，組み込みシステムの複雑化にともない，組み込み機器用の次世代OSとして，Linuxに対する興味が高まってきている．これからの組み込みシステムはインターネット接続が標準となり，無線ネットワークや電源ネットワークを介してお互いに通信可能となると考えられる．それにより個々に稼動していた各種のサービスが統合され，従来とくらべてはるかに高度な機能を提供する可能性が期待される．あらゆる機器接続の実現を考えた場合，複雑なシステムを構築するためにはLinuxのような高度な基盤OSが不可欠となってきていている．

組み込みLinuxが注目を集めている理由として，以下の4点を考えることができる．

第1は，Linuxがファイルシステムやネットワークシステムをはじめからコンポーネントとして含んでいることである．従来のリアルタイムOS(以下RTOS)の多くは，ファイルシステムやネットワークシステムをミドルウェアとして提供しており，原則として開発者は開発システムに適したファイルシステムやネットワークシステムを選択できる．しかし実際には，これらのソフトウェアを開発システムの特性に応じて変更することはたいへん困難なため，RTOSベンダから提供されるものをそのまま利用しているのが現状である．この現状をふまえると，ファイルシステムやネットワークシステムをミドルウェアとして提供する従来のRTOSに対し，それらを標準でサポートしているLinuxのほうが，一貫したシステムを構成しやすい．

第2には，Linuxのカーネルインターフェースが，UNIXをベースにした国際標準のカーネルインターフェースであるPOSIXに準拠していることである．POSIXは世界的にもっとも普及しているカーネルインターフェースであり，POSIXに基づく多くの研究・教育が存在し，多くの書籍や刊行物も出版されている．このため，POSIXインターフェースをマスタしているプログラマが世界中に多数存在する．開発コスト削減が大きな課題である現在，組み込みシステムでPOSIXに基づくOSを用いることは，教育コストの削減やグローバルな開発体制を容易に実現可能という点で，たいへん魅力的である．

第3には，Linuxがオープンソースアプローチという方式により開発されていることである．設計図にあたるソースコードが公開されているため，開発者が世界中の地理的に離れた場所に点在していても事実上共同開発が可能となる．この方式では1人のコーディネータが他の開発者から提示された変更をチェックし，変更が好ましい場合はマスターソースコードに変更を追加する．開発されたソフトウェアは無償で入手可能であり，興味がある人ならば誰でもテストできる．テスト結果はコーディネータにフィードバックされ，さらに誰かほかの開発者により改良されていく．潜在的に多数のプログラマがテストやバグの修正を行うため，従来に比べてすばやく安定したソフトウェアの構築が可能となる．一般に基盤ソフトウェアは開発にコストがかかるわりには利益が少なく，従来は少数の機関で基盤ソフトウェアを開発していたため莫大な開発費が必要だった．一方，オープンソースアプローチでは，前述したように異なる組織に属する多くのプログラマが無償でソフトウェア開発を行う部分が多い．オープンソースアプローチの利用により，ほかの人が開発したソフトウェアの利用に対価を要求することは難しくなるが，開発コストの削減がもっとも重要な項目である場合は，きわめて好ましいアプローチといえる．

第4には，Linux上には多くのライブラリやミドルウェアが存在し，それらのインターフェースがデファクトスタンダードとなっているため，移植性の高いアプリケーションやミドルウェアを構築することが可能である．たとえば，マルチメディア処理のためのインターフェースやGUIのためのインターフェース，それらに基づいたディジタルテレビ用のインターフェースなど，さまざまな標準インターフェースが提供され，それらのインターフェースを用いたアプリケーションやインターフェースの明細を記述したドキュメントも完備されつつある．また，世界中のオープンコミュニティにサポートされているため，時代に応じた新しいテクノロジが導入される可能性が高い．

以上の理由により，組み込みLinuxは今後複雑化する組み込みシステムを支えるOSとして最適なOSであり，将来の組み込みシステムを大きく飛躍させる可能性をもっているということができる．

中島 達夫　早稲田大学 理工学部 情報学科 教授

第1章
リアルタイムOSを使った開発とどう変わるのか
Linuxを使った組み込みシステム開発

木内 志朗

本章では，組み込みシステムを開発するとき，Linuxを採用することで何が変わるのかを明らかにする．スケジューリングの実行単位やデバイスドライバ，リアルタイム性，メモリサイズ，開発手順/開発環境，オープンソースの考え方などについて，Linuxを使った場合と一般的なリアルタイムOSを使った場合でどう異なるのかを解説する．

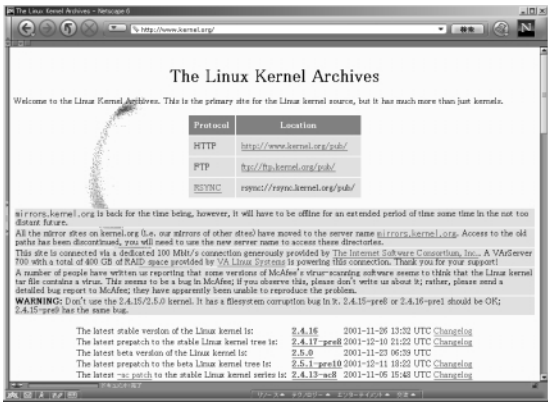
(編集部)

はじめに

組み込みシステムの開発パラダイムが大きく変わろうとしています．20年以上前に，商用の**リアルタイムOS**(以下RTOS)が登場して以来，組み込みシステムでは，さまざまなRTOSが使われてきました．そして現在，この組み込みシステム向けOSの選択肢として，**Linux**が注目されています．

Linuxは，もともとデスクトップ環境向けに開発されたため，ハードディスク上で動作し，コンソールやディスプレイなどをもったシステム構成だと思われがちですが，カーネルやシステムの構成により，ファイルシステムをROM/RAM上に構築した「ディスクレスシステム」や，loginコンソールなどを必要としない「ヘッドレスシステム」としても使用できます．つまり，組み込みシステムのOSとして使用することができるわけです．

〔図1〕Linux最新バージョンのダウンロードサイト



1 Linuxについて

1.1 概要

Linuxは，マルチタスク，仮想メモリ，共有ライブラリ，デマンドローディング，メモリ管理，TCP/IPネットワーク機能などを含んだUNIXクローンのOSです．1991年にLinus Torvalds氏によりLinuxの原型が開発されてからすでに10年以上にわたり，修正/拡張が続けられています．現在(本稿執筆時点)のLinuxの最新バージョンは安定版が2.4.20，開発版が2.5.59であり，`http://www.kernel.org/`からダウンロードができます(**図1**)．

現在，このカーネルツリーで正式にサポートされているCPUアーキテクチャには，x86，Alpha，SPARC，68K，PowerPC，PowerPC64，ARM，SH，S/390，MIPS，PA-RISC，IA-64，VAX，AMD x86-64，CRISなどがあり，他のOSとは比べものにならないほど豊富なCPUをサポートしています．また，アーキテクチャに依存するコードと共通部分が分かれているため，他のアーキテクチャへの移植も比較的容易にできます．

1.2 組み込みLinuxの動向

日本のサーバ環境としては，Linuxが現在約35.5%の普及率(日本Linux協会のデータによる)ですが，組み込みシステム分野においては，これから急速に伸び

〔表1〕ワールドワイドでのアプリケーション別の組み込みLinux出荷予測

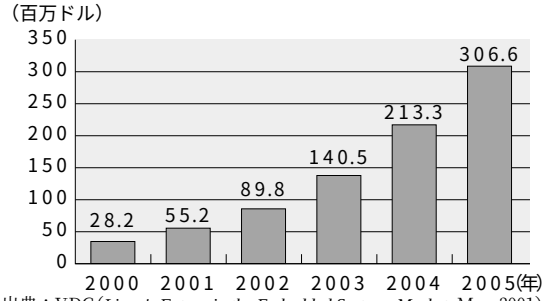
分　野	2000年	2005年
テレコム/データコム	13.2	126.7
コンシューマエレクトロニクス	8.9	121.1
FA	1.4	14.2
リテイルオートメーション	1	8.7
OA	0.9	6.6
軍事/航空	0.9	9.4
自動車（情報系）	0.7	7.4
ストレージ/サーバ	0.6	6.6
医療	0.4	4.0

※単位：百万
※ツール，サービスなどは除く

出典：VDC（*Linux's Future in the Embedded Systems Market*, May 2001）

〔図2〕ワールドワイドにおける組み込みLinux，ソフトウェア，開発ツール，サービスの市場規模

年	2000	2001	2002	2003	2004	2005
金額（百万$）	28.2	55.2	89.8	140.5	213.3	306.6



出典：VDC（*Linux's Future in the Embedded Systems Market*, May 2001）

ていくと思われます．

2000年は，「組み込みLinux元年」とでもいうべき年で，各メーカーが製品化への検討や試作を始めました．現在では，Linuxを評価する段階はある程度終わり，実際に製品搭載に向けた開発が数多く行われています．2002年の前半には，いくつかの製品が市場に出てきました．

現在，組み込みLinuxを採用，あるいは採用を検討している製品として，通信機器，通信インフラといったテレコム/データコム分野と，携帯電話，PDA，ディジタルTVなどをはじめとするディジタルコンシューマ機器の二つの分野への採用が急速に伸びています（**表1**）．

米国のVDCのデータによれば，現在55百万ドルの組み込みLinuxやサービス市場が，2005年には6倍の306.6百万ドルに成長するという予測もされています（**図2**）．

では，なぜ今，組み込みLinuxが急速に普及しているのでしょうか？　組み込みLinuxを選択する理由としては，次のような点があげられます．

●ソースが入手可能

Linuxの場合，完全にオープンソースであるため，問題の切り分けが容易であり，他のRTOSに比べインターネットや書籍などから多くの情報を容易に入手できます．また，商用RTOSの場合，海外に本拠地をもっているベンダが多く，会社の合併やサポート体制の問題などが生じた場合，いざとなれば自分達でメンテナンスしていくことができるという**オープンソース**の安心感のために，Linuxを採用している企業もあるようです．

●ロイヤリティフリー

商用RTOSのほとんどは，搭載製品に対してロイヤリティが発生します．また，自社製のRTOSを使用していても，ネットワークやファイルシステムなどは自作できず，結局はロイヤリティがかかるソフトウェアコンポーネントを購入することすらあります．このロイヤリティは，直接製品の価格に影響してくるので，Linuxのようなロイヤリティフリーでファイルシステムやネットワーク機能をサポートしているOSは魅力です．

●デバイスドライバが豊富

シリアル，イーサネットをはじめUSB，IEEE1394など豊富なデバイスドライバがすでに開発されています．

●ネットワークプロトコルスタックやミドルウェアが豊富

Linuxでは，デスクトップ環境で培われた豊富なGUI，Javaやプロトコルなどが，オープンソースコミュニティから入手可能です．

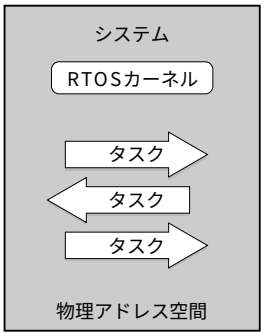
●安定している

安定という面では，すでにサーバでの実績があります．また，強力なメモリ保護機能により，一つのプロセスが問題を起こしてもシステム全体への影響を最小限にできます．

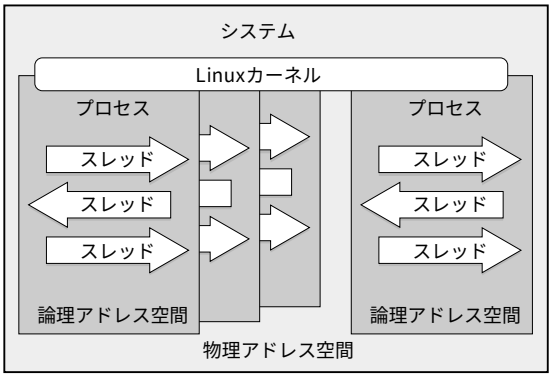
その反面，組み込みLinuxについては，それほど情報が豊富ではないため，次のような点で採用に不安感をもっている開発者も多いのではないかと思います．

- Linuxの知識をもった組み込みシステムのエンジニアの不足
- 組み込みLinuxについての情報不足

〔図3〕
RTOSにおけるタスクのイメージ



〔図4〕Linuxにおけるスレッドのイメージ



- メモリサイズ
- リアルタイム性
- 開発ツールの不足
- ライセンス問題

それぞれの点について，次項より詳しく説明していきます．

2 RTOSとの比較によるLinuxの解説

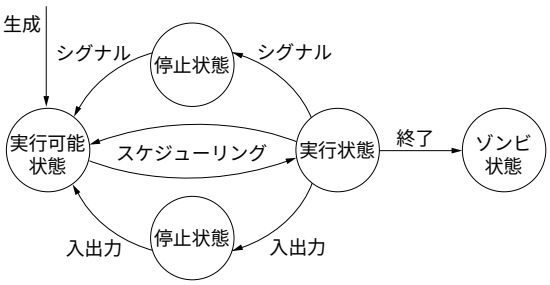
組み込みシステムの開発者で，かつLinuxに精通している技術者は多くありません．そのために，導入をためらっている開発者が非常に多いのが実情のようです．

ここではITRONや商用RTOSと組み込みLinuxをいくつかの点で比べてみることにより，組み込みOSとしてのLinuxを説明していきます．

2.1 RTOSタスクとLinuxプロセス/スレッドの違い

多くのRTOSでは，「タスク」がカーネルによってスケジューリングされる実行単位です．CPUのMMU（Memory Management Unit）を使用せずに，物理アドレスと論理アドレスが同一のフラットなメモリ空間で動作します．RTOSにおけるタスクはメモリ空間を共有しており，タスクごとのメモリ保護などは行われていないのが一般的です（**図3**）．ただし，プロセスモデルを採用しているRTOSも存在しています．

〔図5〕Linuxのプロセスの状態遷移



一方，Linuxではプロセスという単位でスケジューリングが行われます．プロセスはそれぞれ独立した仮想メモリ空間をもっています．それぞれのプロセスのメモリ空間は独立しており，他のプロセスのメモリ空間にアクセスすることはできません．Linuxのカーネル自体もプロセスと独立したカーネル空間で動作しており，プロセスからカーネルの資源にアクセスすることはできません．

また，LinuxはPOSIXのスレッド（pthread）もサポートしています．スレッドは，プロセスの中に存在する最小の処理単位であり，一つのプロセス内でスケジューリングされます．ただし，Linuxのスレッドの実装では，メモリ空間を共有したプロセスとして実装されており，スケジューラなどはプロセスと同様に同一のスケジューラでスケジューリングされています．スレッドはメモリ空間を共有しているため，スレッド間で共有するリソースへアクセスする場合，RTOSと同様にセマフォなどによる排他制御が必要となります．アプリケーションプログラムから見ると，Linuxのスレッドは RTOSでのタスクに似ているといえるかもしれません（**図4**）．

Linuxのプロセスの状態遷移を，**図5**に示します．一般的なRTOSとそれほど違わないことがわかると思います．

簡単なプロセスとスレッドのプログラム例を示します．**リスト1**では，プロセスの生成を行っています．次に，同様の処理をスレッドで行った例を**リスト2**に示します．

〔リスト1〕プロセスの生成

```
/* プロセスの生成 */

#include <sysy/types.h>
#include <unistd.h>
#include <errno.h>

main()
{
    pid_t new_PID;
    int status;

    new_PID = fork();
    switch (new_PID) {
         case 0 :
              /* 子プロセスの処理はここから始まります */
              printf("I am the child!!¥n");
              sleep(2);
              exit(1);
              break;
         case -1 :
              /* 何かエラーが発生しました */
              exit(errno);
              break;
         default :
              /* 親プロセスの処理の続きです */
              printf("I am the parent! my child's
                              PID is %d¥n", new_PID);
              /* 子プロセスが終了するのを待ちます */
              wait(&status);
              break;
    }
    exit(0);
}
```

〔リスト2〕リスト1と同様の処理をスレッドで行う

```
/* スレッドの生成 */

#include <stdlib.h>
#include <stdio.h>
#include <sys/types.h>
#include <pthread.h>
#include <unistd.h>

void *thread_function(void *arg);

main()
{
    int res;
    pthread_t new_TID;
    void *thread_result;

    /* スレッドを生成します */
    res = pthread_create(&new_TID, NULL,
                                thread_function, NULL);
    if (res != 0)
         exit(EXIT_FAILURE);

    /* 子スレッドの終了を待ちます */
    res = pthread_join(new_TID, &thread_result);
    if (res != 0)
         exit(EXIT_FAILURE);

    exit(0);
}

/* 生成されたスレッド */
void *thread_function(void *arg)
{
    printf("I am New thread! --
                               thread_function()¥n");
    sleep(3);
    /* スレッドを終了します */
    pthread_exit("Bye!");
}
```

2.2 メモリ管理

プロセスのところでも説明しましたが，メモリモデルがフラットなRTOSと異なり，LinuxはMMUを使用してメモリ管理をしています．ユーザープロセスにはそれぞれ仮想メモリ空間を割り当て，カーネル自体はカーネル空間で動作します．Linuxのメモリ空間は，4Kバイト(サイズが異なるアーキテクチャもある)ごとのページで管理されています．

カーネル空間は，カーネルの初期設定時にマッピングされると，その後変更されることはありませんが，プロセスに割り当てたユーザー空間は，このページ単位で連続したメモリに割り当てられるともかぎらず，ディスク上にスワップされることもあります．しかし，このページング機構のおかげでシステム全体のメモリを効率よく使用することができます．

2.3 タスク，プロセス間の同期/通信手段

RTOSを使用したマルチタスクシステムの場合，タスク間の同期を行う際，カーネルにより同期，通信などのシステムコールが用意されています．Linuxにおいても，同様のプロセス/スレッド間での同期/通信手段がシステムコールによってサポートされています．

RTOSでは，排他制御や同期手段としてセマフォが使用されますが，Linuxも当然のことながらセマフォをサポートしています．mutexや状態変数についても，もともとUNIXやPOSIXでもっているものなので，Linuxは当然サポートしています．タスク間の通信手段としてはメッセージキューやメールボックスを使用しますが，Linuxではパイプといった処理で実現可能です．また，イベントフラグにしても，Linuxのシグナル処理で同等の機能をもたせることが可能です．まとめると**表2**のようになります．

2.4 デバイスドライバ

ITRONなどの場合，デバイスドライバという標準的なAPI(アプリケーションインターフェース)が現在のところないため，明確にデバイスドライバとタスクという分け方をしていない実装も大半を占めているようです．VxWorksやpSOS+といった商用RTOSでは，それぞれデバイスドライバのAPIをもっており，元

〔表2〕
RTOS，Linuxの同期/通信手段の比較

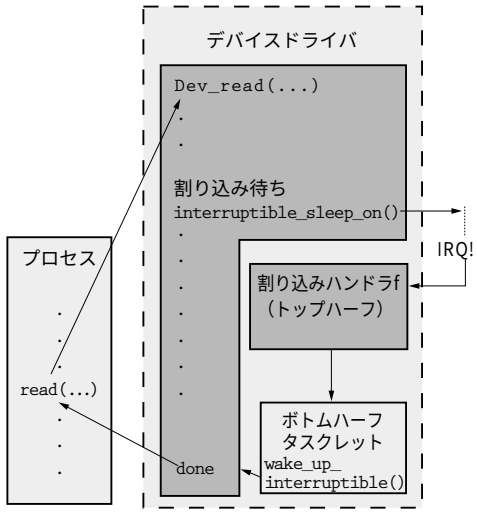
RTOS 同期/通信	Linux 同期/通信
セマフォ（バイナリ/カウンティング）	SVR4セマフォ
mutex	POSIX mutex，状態変数
共有メモリ	共有メモリ
メッセージキュー/メールボックス	パイプ/FIFO，SVR4キュー
イベントフラグ	シグナル
タイマ	POSIXタイマ/アラーム，sleep()/usleep()

となる考え方がUNIX系に近いものなので，比較的Linuxのデバイスドライバについてもイメージしやすいと思います．しかし，実際にはITRONや他の商用RTOSいずれの場合もタスク内の処理で直接I/Oへのアクセスを行っている例が多いようです．周期的にI/Oポートを監視する，あるいはデータ入出力を行うといった処理をすべてタスクが直接行っています．

一方，Linuxの場合，基本的にプロセスから直接ハードウェアへのアクセスはできません．Linuxの場合，I/Oなどのデバイスをアクセスする際には，デバイスドライバで行う必要があります．Linuxのデバイスドライバの構造は，**図6**のようになります．プロセスが`write()`システムコールを発行すると，カーネル空間に移り，登録されているデバイスドライバの読み込み処理〔`Dev_Read()`〕を実行します．

`Dev_Read()`では，デバイスからのデータを読み込むのですが，この図では，割り込みハンドラがデータを取り込むまで，`interruptible_sleep_on()`システムコールによりスリープします．ハードウェアからの割り込みが発生し，割り込みハンドラが実行されます．

〔図6〕Linuxのデバイスドライバの構造



Linuxの割り込み処理は，トップハーフとボトムハーフ（Linux 2.4では，タスクレットやソフトIRQ）の二つの構造に分けることができます．たとえばトップハーフでは，デバイスからデータを取り組むだけといったリアルタイム性が厳しい処理を行い，その後，ボトムハーフ処理でそのデータを処理するなどといった実装がされます．データの処理が終わるとボトムハーフ処理は，`wake_up_interruptible()`システムコールなどにより，スリープしているプロセス〔具体的には`Dev_Read()`〕を起床させます．

ソースで見ると，**リスト3**のようになります．特定のデバイス用ではなく，何もしない架空のキャラクタデバイスドライバです．コメントを読んでいただければ，デバイスドライバの構造がイメージできると思います．デバイスドライバといっても，それほど複雑なものではないことがわかります．

2.5 スケジューリング

ほとんどのRTOSでは，タスクごとの優先順位ベースのスケジューリングが行われています．つまり，各タスクがそれぞれ優先順位をもっており，優先順位が高いタスクが常に実行されます．

これに対して，Linuxのスケジューリングはタイムシェアリングをベースにしており，各プロセスに対して平等にCPU時間を割り当てるようにスケジューリングされています．

Linuxの基準時間（Tick）はデフォルトで10msなのですが，このTickの倍数でしかスケジューリングされないと勘違いしている人も多いようです．Linuxのスケジューリングされるタイミングは，割り込みハンドラ終了時，システムコール終了時とアイドル時です．

したがって，Tick刻みでスケジューリングされるプロセスもあれば，非同期に発生する割り込み終了後や，システムコール終了後にスケジューリングされるプロセスもあります．

ただし，Linuxカーネルはプリエンプティブなカーネルではないので，割り込みハンドラ終了時には必ず

〔リスト3〕キャラクタデバイスドライバのコード例

```
/* サンプル キャラクタ デバイスドライバ （Linux Kernel Version 2.4 以降）*/

#if defined(CONFIG_MODVERSIONS)
#define    MODVERSIONS
#include <linux/modversions.h>
#endif

#include <linux/module.h>
#include <linux/init.h>
#include <linux/kernel.h>
#include <linux/tqueue.h>
#include <linux/sched.h>
#include <linux/timer.h>
#include <linux/interrupt.h>
#include <asm/uaccess.h>

#define MYDRIVER_MAJOR 240                /* デバイスメジャー番号 */
#define MYDRIVER_NAME  "Test Driver"      /* デバイス名 */

DECLARE_WAIT_QUEUE_HEAD(mydriver_zz);     /* interruptible_sleep_on() 用のキュー */


/************************************************************************
 * <割り込みハンドラ（トップハーフ）>
 */
void mydriver_interrupt(int irq, void *dev_id, struct pt_regs *fp)
{
    /* デバイスからデータを取り込みます */

    /* 割り込みの後処理としてタスクレットの実行フラグ
       をセットします．*/
    tasklet_init(&mydriver_tasklet, mydriver_task, (unsigned long)dev_id);
    tasklet_schedule(&mydriver_tasklet);
    return;
}


/************************************************************************
 * <割り込み後処理（タスクレット）>
 */
void mydriver_task(unsigned long data)
{
    /* 割り込みハンドラが取り込んだデータを加工したり
       します．*/

    /* スリープしている プロセスを起床させます．*/
    wake_up_interruptible(&mydriver_zz);
    return;
}


/************************************************************************
 * <オープン処理>
 * この関数は，アプリケーションからopen()が呼ばれたときにコールされ
 * ます．
 */
static int mydriver_open(struct inode *inode, struct file *file)
{
    int rc;
    /* デバイスの初期化やワークエリアの確保、初期化
       タイマなどの設定などを書きます．*/

    /* デバイスの割込みハンドラを登録します．*/
    rc = request_irq(IRQXX, mydriver_interrupt, SA_SHIRQ, MYDRIVE_NAME, NULL);
    if (rc) {
        return -EBUSY;
    }

    MOD_INC_USE_COUNT;
    return 0;
}


/************************************************************************
 * <リリース処理>
 * この関数は，アプリケーションからclose()が呼ばれたときにコールされ
 * ます．
```

〔リスト3〕キャラクタデバイスドライバのコード例（つづき）

```
 */
static int mydriver_release(struct inode *inode, struct file *file)
{
    /* ワークエリアの開放，タイマの削除などの処理を
       書きます．*/
    MOD_DEC_USE_COUNT;

    return 0;
}


/************************************************************************
 * <読み込み処理>
 * この関数は，アプリケーションからread()が呼ばれたときにコールされ
 * ます．
 */
static ssize_t mydriver_read(struct file *file, char *buf, size_t count, loff_t *offset)
{
    unsigned char data[10];

    /* デバイスからデータを読み込みます．ここでは，
       実際にデータを読み込むのは，デバイスドライ
       バと仮定して，スリープさせます．  */
    interruptible_sleep_on(&mydriver_zz);

    /* ペンディングされているシグナルを確認します．  */
    if (singnal_pending(current))
         return -EINTR;

    /* ボトムハーフ処理で発行された wake_up_interruptible()
       によって，起床し以下のコードが実行されます．  */

    /* dataはカーネル空間ですので，カーネル空間からユー
       ザー空間のbufにデータをコピーします．  */
    if (copy_to_user(buf, &data, sizeof(data)))
              return -EFAULT;
         return mydriver_final_count;
    } else {
         return -1;
    }
}


/************************************************************************
 * <書き込み処理>
 * この関数は，アプリケーションからwrite()が呼ばれたときにコールされ
 * ます．
 */
static ssize_t mydriver_write(struct file *file, const char *buf, size_t count, loff_t *offset)
{
    char data[10];

    /* デバイスに書き込むデータ buf は，ユーザー空間のデー
       タですので，カーネル空間の data にコピーします */
    if (copy_from_user(&data, buf, sizeof(data)))
         return -EFAULT;

    /* ここで，data の値をデバイスに書き込みます */

    return sizeof(data);
}


/************************************************************************
 * <ファイル構造体>
 * この構造体で，このデバイスドライバへの各操作に対応する関数ポインタ
 * を指定しています．
 */
static struct file_operations my_fops = {
    read:     mydriver_read,
    write:    mydriver_write,
    open:     mydriver_open,
    release:  mydriver_release,
};
```

〔リスト3〕キャラクタデバイスドライバのコード例（つづき）

```
/*************************************************************************
 *  <ドライバ初期設定>
 *  この関数は，モジュール登録時（insmod）に呼ばれます．ドライバを
 *  スタティックリンクしてある場合は、カーネルの初期化時に呼ばれます．
 *  処理としては，カーネルにデバイスドライバを登録し，必要な変数や
 *  構造体などを初期化します．
 */
static int __init my_init_module(void)
{
    /* カーネルにデバイスドライバを登録します．その際にドライバ名と
       デバイスのメジャー番号を指定します．
    */
    res = register_chrdev(MYDRIVER_MAJOR, MYDRIVER_NAME, &my_fops);
    if (res) {
         printk("init_mydriver: register_chrdev() failed, rc==%d¥n", res);
         return -EIO;
    }
    return 0;
}


/*************************************************************************
 *  <ドライバ終了>
 *  この関数は，モジュール終了時に呼ばれます．  通常，登録されたドライバ
 *  を削除します．
*/
static void __exit my_cleanup_module(void)
{
    /* カーネルからデバイスを削除します */
    unregister_chrdev(MYDRIVER_MAJOR, MYDRIVER_NAME);
    return;
}

module_init(my_init_module);     /* モジュール/ドライバ登録 */
module_exit(my_cleanup_module);  /* モジュール/ドライバ終了 */

/* サンプルドライバ END */
```

スケジューリングされるわけではありません．これについての説明は後で行います．

2.6　リアルタイム性

組み込みシステムでは，一定時間ごとに行う処理や，割り込みが発生してから処理を実行するまでの時間などという，何らかの時間的な制限をもっています．

組み込みシステム＝リアルタイムシステム

ということではありませんが，大半の組み込みシステムにはリアルタイム性が要求されます．

実際に，一般のRTOSでは，コンテキスト時間や割り込みマスク時間というのは，数μsから何十μs程度だと思います．各RTOSベンダが提示しているコンテキストスイッチや割り込みマスク時間などのデータは，測定時間やどこからどこまでをコンテキストスイッチ時間といっているかという点などが各ベンダにより異なるため，データシートの情報だけで純粋なRTOSの性能を比較することはできません．一方，Linuxの場合，一般のRTOSよりはるかに時間がかかります．

RTOSのコンテキストには，レジスタ，スタックなどが含まれます．Linuxの場合，これ以外にメモリ空間やファイル，ユーザーなど多くの情報をもっています．また，MMUを切り替える処理も含まれるため，どうしてもRTOSに比べると重くなってしまいます．

また，Linuxはプリエンプティブなカーネルではありません．カーネルのコードは，割り込まれたり他のプロセスにデータを変更されないことを前提として処理を続けているところもあります．たとえば，システムコール中に割り込みが発生し，割り込み処理からスレッドやプロセスの起動をかけた場合でも，割り込み終了後ただちにスケジューリングされるのではなく，システムコールが終了するまで行われません（**図7**）．したがって，タイミングによっては割り込みからの応答性が非常に悪くなります．

また，割り込みからの応答性を考えた場合，割り込みマスクしている時間についても考慮する必要があります．Linuxでは，もともとx86ベースで開発が行われてきたため，割り込みレベルごとにマスクするのではなく，`cli()`，`sti()`といった割り込みすべてをマスクするように実装されている箇所が多くあります．もともとリアルタイム性を重視して作られていないの