修士論文

逆アセンブルを用いたWindowsデバイス
ドライバからＬｉｎｕｘデバイスドライバへと
変換する方法についての研究

氏名　平田純一
平成１２年度入学
島根大学大学院総合理工学研究科
数理・情報システム学専攻計算機科学講座
指導教官　田中章司郎
平成１４年２月２０日受理

目次

1. 序論

1.1. 本研究の目的と背景

本研究の目的は Linux のデバイスドライバを Windows のデバイスドライバから変換、生成することである。

近年、ソースコードを公開した OS である Linux が急速に普及してきている。 しかしながら、デバイスドライバが存在しないデバイスも数多く存在する。これは、Linux が GNU General public License に基づいて配布されており、誰でも自由に改変できることから、世界中の有志の手によって開発されてきたために、詳しいハードウェアの情報を知ることができないデバイス、新しいデバイスについてはデバイスドライバが開発できないからである。デバイスドライバが開発されてないデバイスを利用するには、有志の手によってデバイスドライバが開発されるのを待つか、自分でデバイスドライバを開発しなければならない。前者は確実に開発されるという保証はなく、また後者はデバイスドライバ開発の知識が必要であり一般のユーザには困難である。

一方で、PC の業界では Windows が事実上の標準の OS であるためにハードウェアベンダは自社のデバイスのデバイスドライバを開発する。

本研究では、Windows デバイスドライバが広く普及していることと、Linux、Windows の OS の違いはあるが同じ PC というハードウェア上でデバイスを操作するプログラムであるため、制御の手順が同じである点に着目し、Windows デバイスドライバのバイナリプログラムを解析し、Linux のデバイスを生成する方法を提案する。

1.2. 本論文の構成

以下に本論文の構成を述べる。

第 2 章では、Windows と Linux のデバイスドライバの構造について述べ、Windows と Linux で類似しているところ、非類似の箇所について述べ以降の章での参考とする。

第 3 章では、変換の手順の概略を述べる。

第 4 章では、第 3 章で述べた変換手順に基づいて実装した Windows－＞Linux デバイスドライバ変換プログラムについて述べる。

第 5 章では、Windows2000－＞Linux デバイスドライバ変換プログラムを用いて変換したデバイスドライバの動作検証について述べる。また、変換して得られた

Linux デバイスドライバと通常の方法で作成された同機能をもつデバイスドライバとの比較を行った。

第 6 章では、本論文全体をまとめ、今後の課題についてまとめる。

以降の章では Windows は Windows2000 のことをさし、Linux はカーネルに 2.2 系のものを採用しているものをさす。

2. デバイスドライバについて

本章では、デバイスドライバを、「プログラムの流れ」、「データ移動」、「変換に必要な各種情報」という点から説明を行う。Windows、Linux の順に述べ、そこから Windows と Linux の間で共通箇所、非共通箇所をまとめる。

図においてプロセス-カーネル間に線を引いているが、これはプログラムの実行権限のレベルの違いを表しており、メモリアドレス空間も異なる。デバイスドライバとカーネルは同じ特権レベルで動作し、ほとんどの処理を実行でき、メモリ空間はカーネルと同じである。

2.1. Windows デバイスドライバの構造

Windwos におけるデバイスドライバのバイナリフォーマットはマイクロソフト社が定義した、“ Portable Executable File Format”[1]が使用されている。

2.1.1. プロセスからデバイスドライバへのアクセス

Windows デバイスドライバは、プロセスに名前つきデバイスを提供してる。名前つきデバイスは特殊なファイルであり、デバイスファイルと呼ばれ、パスは「¥¥.¥xxx」で与えられる。（xxx はデバイスドライバが定義する）

デバイス xxx にアクセスするためには通常のファイル入出力と同様に CreateFile 関数で「¥¥.¥xxx」ファイルを開き、ReadFile,WriteFile 関数を用いてデバイスへ読み書きを行い、デバイスファイルを使い終わってファイルハンドルを閉じるときには CloseHandle 関数を使用する。これらの関数のことをシステムコールという。

Win32 関数名	デバイスへの処理
CreateFile	デバイスファイルを開く。
ReadFile	デバイスからアプリケーションへデータを移動する。
WriteFile	アプリケーションからデバイスへデータを移動する。
CloseHandle	デバイスファイルを閉じる。

表 2.1　Windows デバイスドライバへアクセスする関数

2.1.2. Windows デバイスドライバのプログラムの流れ

Windows デバイスドライバを OS にインストールする時に初期化関数 DriverEntry 関数（図 2-1①）が呼びだされる。

DriverEntry ではデバイスドライバの初期化が行われる。具体的には、前項で述べたデバイスへのシステムコール CreateFile,ReadFile,WriteFile,CloseFile を処理するための関数（図中の Create,Read,Write,Close）をカーネルへ登録するためにカーネル内のテーブルへ各関数へのポインタの値を代入することと、デバイスへのインターフェースであるデバイスファイルの作成を行う。

DriverEntry が終了するとカーネルは他のプロセスへ制御を移し、デバイスドライバはデバイスファイルへのシステムコールが呼ばれるまで何も行わない。

デバイスドライバのコードはカーネル機能を使用するためにカーネルが用意しているカーネル関数を呼び出す(図 2-1②)。デバイスドライバとカーネル関数の関係は、アプリケーションプログラムとライブラリ関数のようなものである。

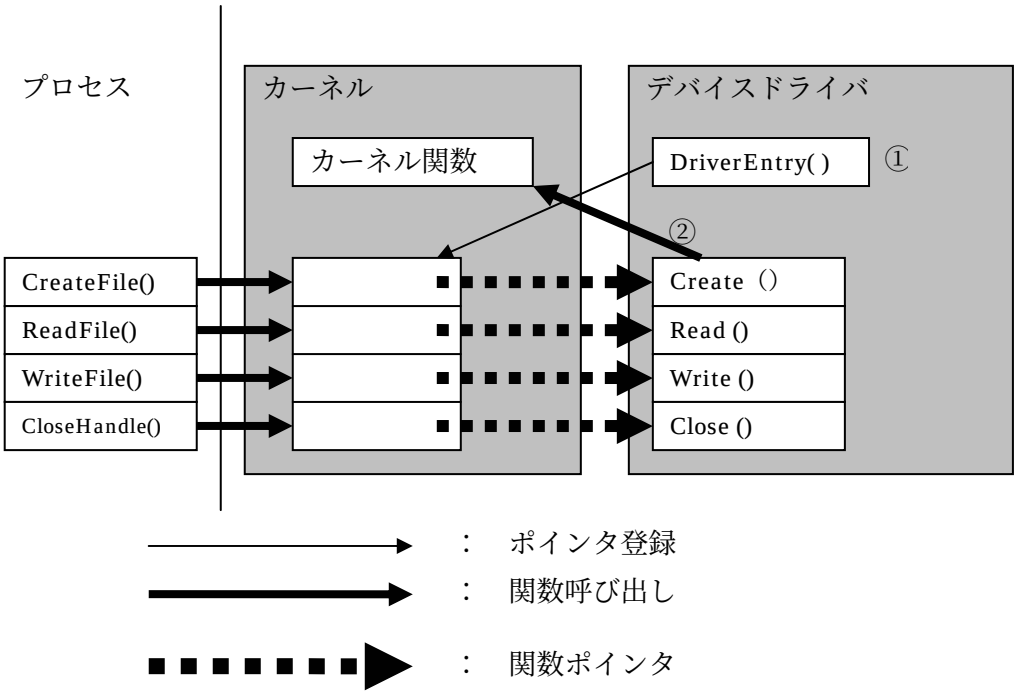


図 2-1　Windows デバイスドライバのプログラムの流れ

2.1.3. Windows デバイスドライバのデータの流れ

この項では、プロセスとデバイス間でどのような手順でデータが移動していくかを述べる。

システムコール ReadFile はデバイスからデータを読む、つまりプロセスから見ると、デバイスからプロセスへデータのコピーを行う。その内部では、先の項で述べたようにシステムコール ReadFile が呼び出されるとデバイスドライバが定義した関数 Read を呼ぶ。関数 Read はデバイスからデータを読む（図 2-2①)。そして関数 Read はデバイスから読んだデータをプロセスへ渡す（図 2-2②)。

システムコール WriteFile はデバイスへデータを書く、つまり、プロセスが所持す

るデータをデバイスへコピーする。内部では、デバイスドライバが定義した関数 Write を呼ぶ。関数 Write はプロセスのデータを渡され（図 2-2③）、デバイスへ書き込む（図 2-2④）。

データ移動の方法は

　　デバイスドライバ　⇔　デバイス　　：　（図の①④）

に対してはデバイスによって異なる。

　　プロセス　　⇔　デバイスドライバ　：　（図の②③）

に対しては mov,movs といった CPU 命令を使用する。

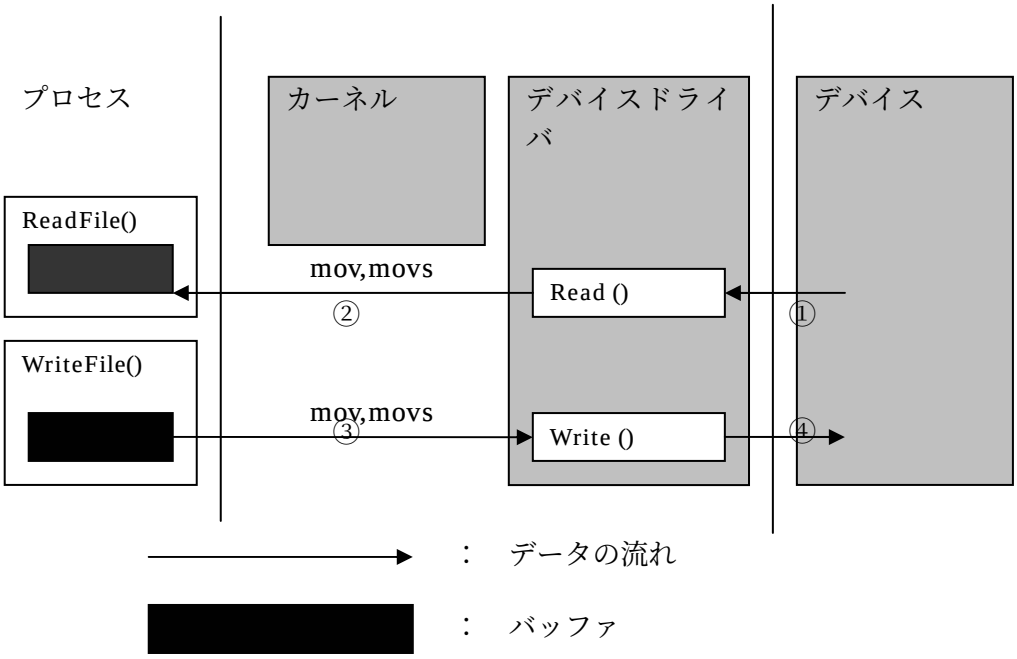


図 2-2　Windows デバイスドライバのデータの流れ

2.2. Linux デバイスドライバについて

Linux デバイスドライバのバイナリフォーマットは、UNIX System Laboratories が開発した Executable and Linking Format(ELF)が使用されている。

2.2.1. プロセスからデバイスドライバへのアクセス

Linux においてもプロセスからデバイスドライバへアクセスするには Windows と同じようにデバイスファイル“ /dev/xxx”へのファイル入出力システムコール（open,read,write,close など）を使用する。

関数名	デバイスへの処理
Open	デバイスファイルを開く。
Read	デバイスからプロセスへデータを移動する。
Write	プロセスからデバイスへデータを移動する。
Close	デバイスファイルを閉じる。

表 2.2　Linux デバイスドライバへアクセスする関数

2.2.2. Linux デバイスドライバのプログラムの流れ

Linux デバイスドライバを OS（カーネル）へインストール（組み込む）するときに初期化関数 init_module(図 2-3①)が呼ばれる。関数 init_module は open,read,write,close に対応する処理（図 2-3 の Create,Read,Write,Close）へのポインタを記述した構造体 file_operations をカーネルへ登録する。

init_module が終了するとカーネルは別のプロセスへ制御を移し、以降はデバイスドライバへのシステムコールが呼ばれるまではデバイスドライバは何もしない。プロセスがデバイスへのシステムコールを発行するとデバイスドライバは機能する。

デバイスドライバのコード(図 2-3 の Create,Read,Write,Close とそのサブルーチン)はカーネル機能を使用するためにカーネルが用意しているカーネル関数を呼び出す(図 2-3②)。

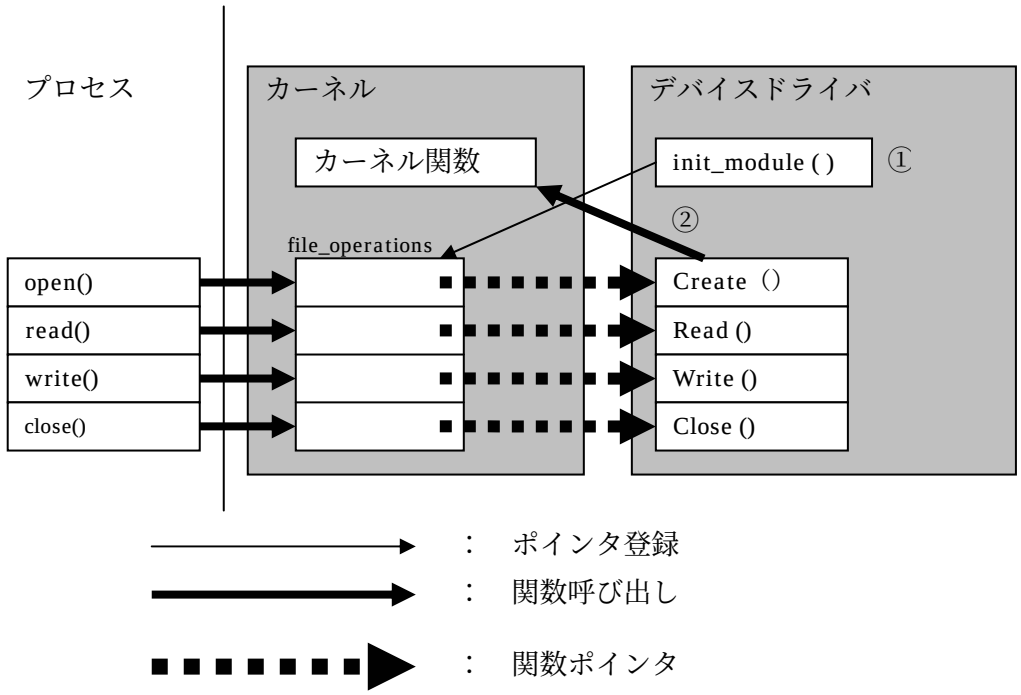


図 2-3　Linux デバイスドライバのプログラムの流れ

2.2.3. Linux デバイスドライバのデータの流れ

Linux ではプロセスからデバイスドライバへデータを移動させるにはカーネル関数 copy_from_user を、デバイスドライバからプロセスへデータを移動させるにはカーネル関数 copy_to_user を使用する。

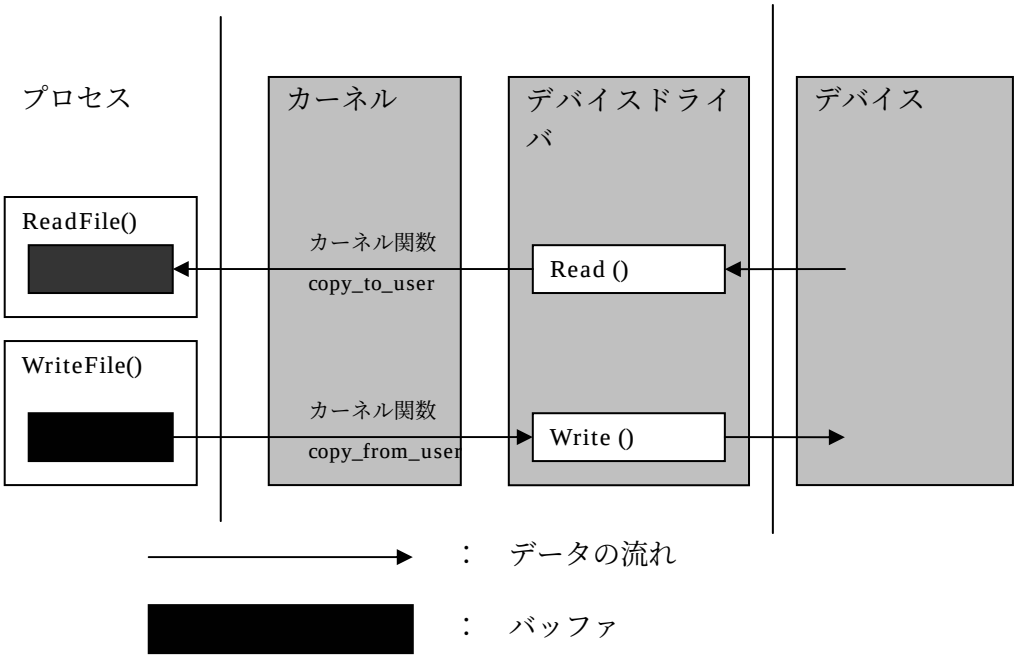


図 2-4　Linux デバイスドライバのデータの流れ

2.3. Windows と Linux デバイスドライバの共通点

プロセスはデバイスファイルを介してデバイスドライバへアクセスする。

そして、デバイスファイルへのシステムコールの機能についても共通する。

以下の表に対応関係を示す。比較のために同機能の関数を横に並べた。

C ライブラリ関数 fopen は CreateFile,open にラップされた関数である。

C ライブラリ関数	Windows	Linux
Fopen	CreateFile	open
Fread	ReadFile	read
Fwrite	WriteFile	write
Fclose	CloseHandle	close

表 2.3　Cライブライ関数、Windows、Linux システムコール対応表

初期化関数がシステムコールを処理する関数をカーネルへ登録し、デバイスへのシステムコールが発行されるまで何も行わないことも似ている。

以上で述べたように、プログラムの動作の流れという点についての違いは見あたらない。図 2-1、図 2-3 を見ると関数名以外の違いはないのがわかる。

故に、Windows デバイスドライバのコードを使用して、Linux でデバイスドラ

イバとして動かすには、

図 2-1(Windows)の	図 2-3(Linux)の
Create を解析して	Create として実行
Read を解析して	Read として実行
Write を解析して	Write として実行
Close を解析して	Close として実行

すればよい。

2.4. Windows、Linux デバイスドライバの非共通点

Windcows デバイスドライバと Linux デバイスドライバの違いは次のように挙げられる。

① バイナリファイルフォーマットの違い。
② プロセス,デバイスドライバ間のデータ移動方法
③ データ構造の違い
④ 関数への引数をスタックから廃棄するタイミング
⑤ カーネル関数

① に関しては 2.1 の冒頭、2.2 の冒頭で述べたように

Windows	Portable Executable File Format
Linux	ELF

が採用されている。

② に関しては 2.1.3 と 2.2.3 で述べたように
Windows　CPU のデータ移動命令 mov,movs
Linux　カーネル関数 copy_to_user,copy_from_user
をそれぞれ使用する。

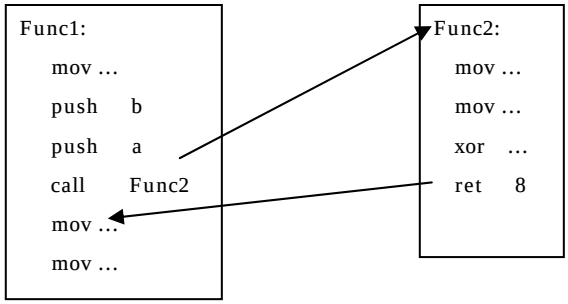
③ に関して、Windows は Windows で定義している型があり、Linux は Linux で定義した型があるということである。本研究では関数の引数の型、及び数が違うということについて扱った。

④ について例を用いて説明を行う。関数 Func2 が関数 Func１からよばれるとする。Func2 のプロトタイプは次のようにする。なお、int 型は４バイトと定義されている。

```
void Func2( int a,int b)
```

Windows では呼ばれた関数内で引数をスタックから廃棄する。
下図の ”ret　8” 命令が Func2 の引数(４ｘ２＝8 バイト)を廃棄してから Func1

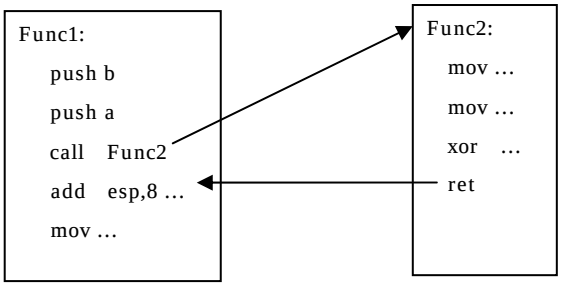
へと戻っている。



一方、Linux では呼んだ関数が呼ばれた関数の引数をスタックから廃棄する。
Func2 から Func1 へ移行する瞬間の処理は

Func2:ret → Func1:add esp,8

のようになっており、Func2 が終了してから呼び出し側の関数 Func１で esp(スタックポインタを表すレジスタ)に Func2 の引数のサイズ 8 を加算することでスタックから関数 Func2 の引数をスタックから廃棄している。



⑤ は Windows のカーネル関数は Linux では実行できない。
もし、Windows カーネル関数 ExAllocatePoolWithTag を使用するプログラムを Linux で実行しようとしてもできない。何故なら ExAllocatePoolWithTag と名のついた関数が Linux カーネルには存在しないためにデバイスドライバをインストール時にエラーメッセージ“ unresolved Symbol ExAllocatePoolWithTag”を提示し終了する。

以上で述べた Windows、Linux デバイスドライバ間の５つの非共通点をどのように解消するかが本研究での問題であり、後の章（変換手順、実装）で詳しく述べていく。

3. デバイスドライバの変換手順

本章ではWindowsデバイスドライバをLinuxデバイスドライバへ変換する手順を示す。説明のやり方として、まず変換手順の全体を示し、個々の要素について説明を行う。

以下に本研究でのデバイスドライバ変換の手順を図に示す。

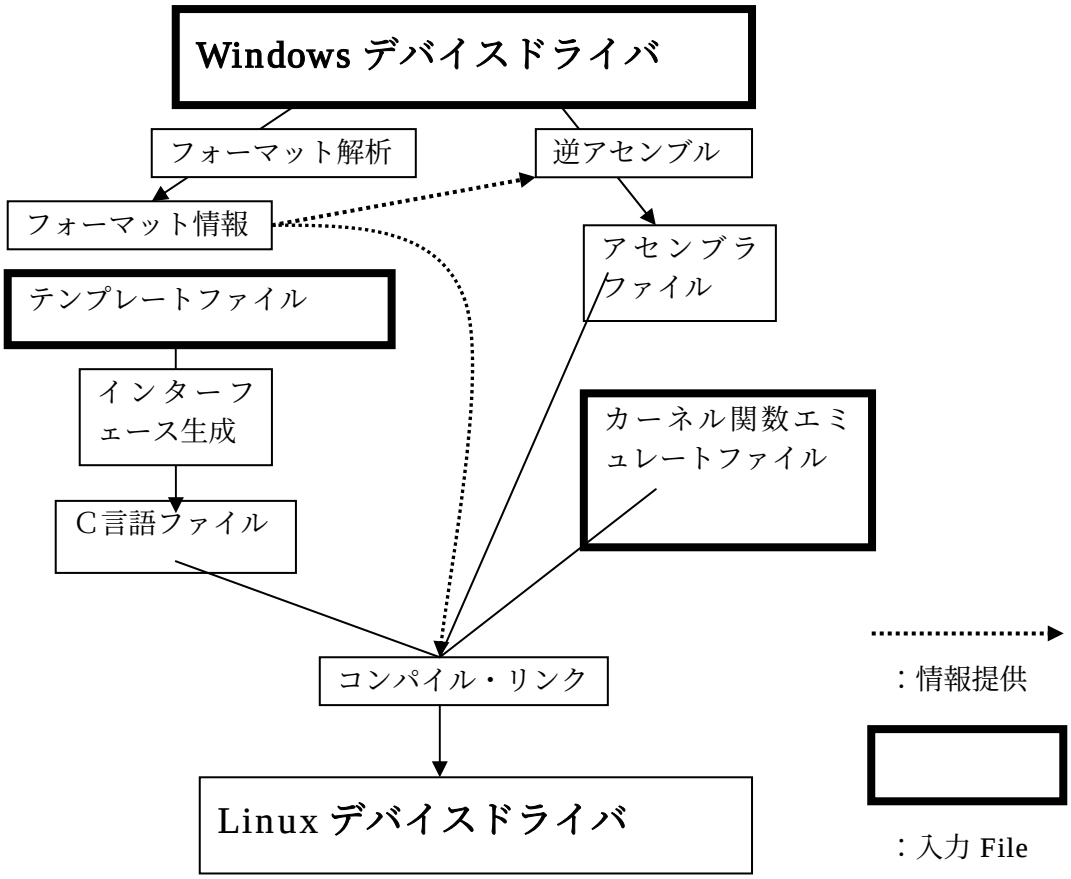
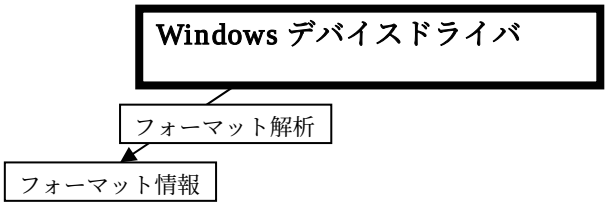


図 3-1　デバイスドライバ変換の手順（全体）

Windows デバイスドライバファイル　->　Linux デバイスドライバ　への過程で太枠で囲ったテンプレートファイル、カーネル関数エミュレートファイルの２つを変換のサポートに使う。

3.1. フォーマット解析



Windows デバイスドライバのバイナリファイルフォーマットを仕様から解析し、必要な情報をメモリへ格納する。

必要な情報を挙げておく。

A) AddressOfEntryPoint
　初期化関数 DriverEntry へのファイルポインタ

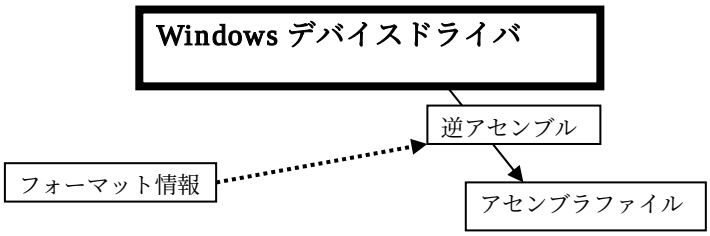
B) ImageBase
　デバイスドライバがロードされるアドレス

C) データセクションの位置とサイズ
　グローバルデータが配置されている位置及びサイズ

D) ImportAddrss とシンボル名
　デバイスドライバからカーネル関数を呼ぶのに使用するシンボル名とアドレス領域の情報。

ここで得られた情報は次節で説明する逆アセンブルで使用される。

3.2. 逆アセンブル



Windows デバイスドライバのコード部分を逆アセンブルにかけ、結果をアセンブラファイルで出力する。出力するアセンブラコードの形式には Netwide Assmbler（NASM）の形式を採用した。Linux 上でのカーネルプログラムの開発は GNU Assembler を使用することが、GNU Assembler は記法に細かい規定が多く逆アセンブラを実装するのが困難であるために採用を見送った。

フォーマット情報から初期化関数 DriverEntry のファイルポインタを得る。得られたファイルポインタから逆アセンブラを開始する。次にシステムコールを処理する関数 Open,Read,Write,Close の逆アセンブルを行う。これらの関数は DriverEntry から登録されるため、DriverEntry を逆アセンブル中にこれらの関数のアドレスを得ることができる。ここで得られたアドレスとは、デバイスドライバをメモリへロードしたときのアドレスの値であり、この値から Windows デバイスドライバファイル内の位置を求めるためにファイルポインタを下のような式で変換を行い、そのファイルポインタの位置からデータをデバイスドライバファイルから読み込み、逆アセンブルを行う。

fp = address – ImageBase

1 つの関数を逆アセンブルするときの流れは以下の図のようになっている。

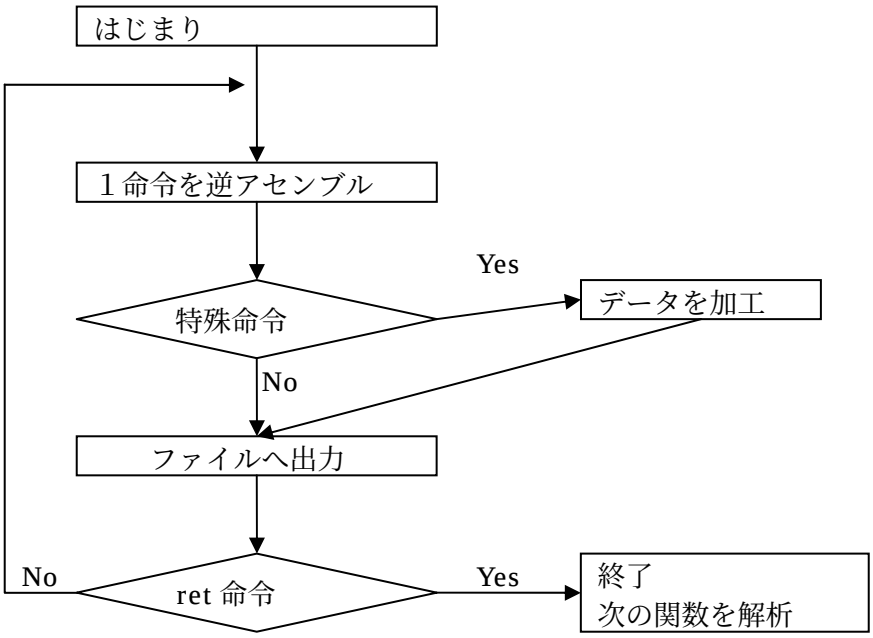


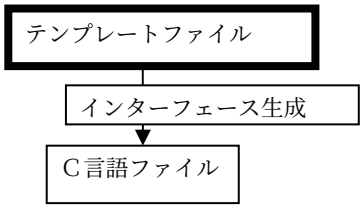
図 3-2　１関数を逆アセンブルするときの流れ

ここでいう特殊命令とは、オペランドにバイナリファイルのフォーマットが絡んでいる命令であり、今回はデータ移動命令 mov と関数呼び出し命令 call、関数から復帰命令 ret とした。

しかし、すべての mov,call,ret 命令に対してデータ加工を施すわけではなくオペラ

ンドにより、mov 命令 はグローバルデータ（範囲はデバイスドライバ内）を指していると思われる値、call 命令はオペランドが Windows カーネル関数をさす場合、ret 命令は Linux デバイスドライバのコードから呼ばれる関数については､引数の廃棄のタイミング（2.4 節）の都合からオペランドを削除して出力する。

3.3. インターフェース生成



2.4 節で述べた②③を解決する仕組みを提供するためにプロセス、Linux デバイスドライバ、Windows デバイスドライバ間のインターフェースを定義する。

解決すべき問題の③型の違いを read システムコールを例に説明を行う。
Linux デバイスドライバで read システムコールを処理する関数は

```
ssize_t    readLinux( struct file *file,char *buf,size_t count,loff_t ppos ) ;
```

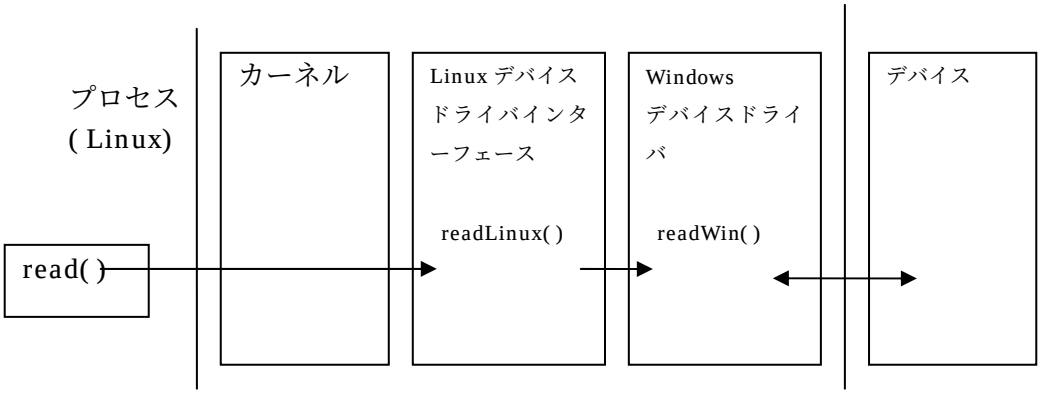
で定義されている。一方 Windows では ReadFile システムコールを処理する関数は

```
int        readWin( IN PDEVICE fdo,IN PIRP irp ) ;
```

で定義されている。IN PDEVICE,IN PIRP 型は Windows が定義する型である。

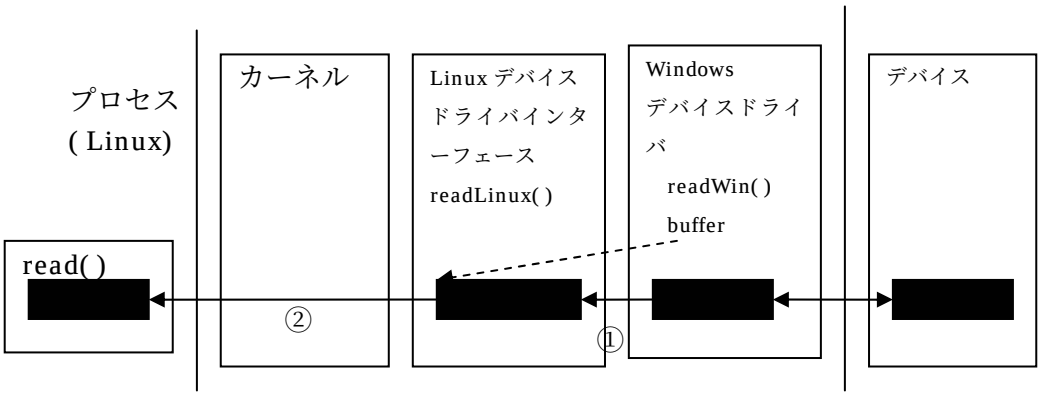
readWin を直接 Linux デバイスドライバとして使うために初期化関数 init_module で readWin を read システムコールを処理する関数として登録するようにプログラミングするとする。しかし、read システムコールが呼ばれ readWin を実行しようとすると引数の型が異なるため、引数 fdo,irp を参照する命令があった場合、えたいの知れないアドレスを参照することになり、プログラムが暴走する危険性がある。

そこで、readLinux で定義した関数を、read システムコールを処理する関数として登録することにし､readLinux から readWin を呼び出す｡readWin は図 2-1 の Read を逆アセンブルした関数であり、実際のほとんどの処理(デバイスへのアクセス等)は readWin が行う。システムコールと readWin との境界に readLinux があるということでインターフェースと呼ぶことにした。

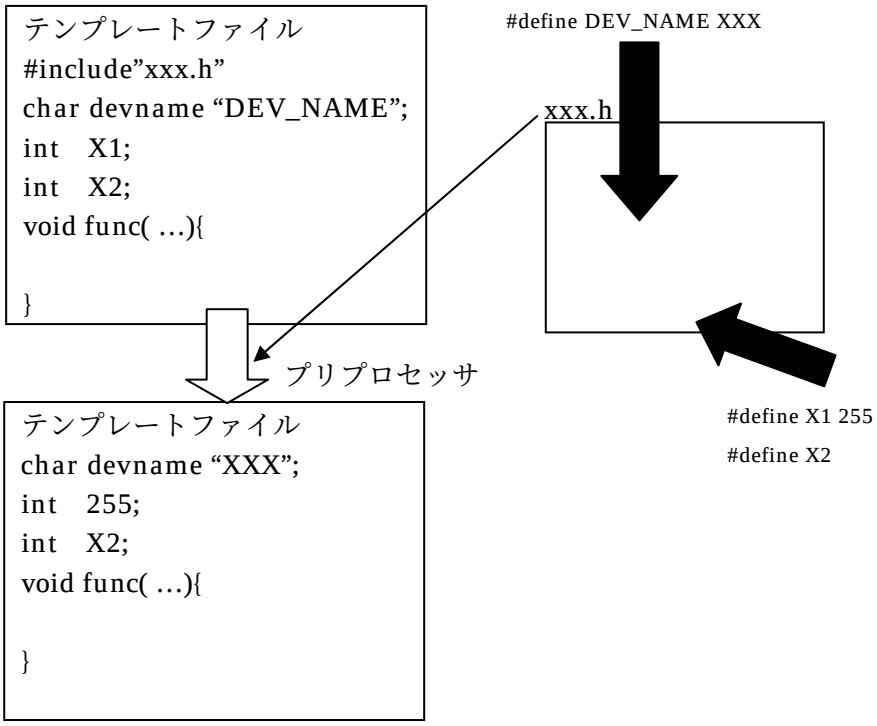


同様に②プロセス⇔デバイスドライバ間のデータ移動方法の問題を解消しているか説明を行う。

readWin からみるとプロセスのバッファの領域は引数の構造体フィールド irp->AssociatedIrp.SytemBuffer が指しているアドレスからである。ゆえにこのフィールドをプロセスのバッファの値を代入すれば直接プロセスへデータを送ることができるとは思うがそのアドレスはわからない（readLinux の引数 buf はプロセスのデータ領域のアドレスをさしているがアドレス空間が異なる）。readLinux の buf から求めることもできそうではあるが、Linux デバイスドライバではカーネル関数をわざわざ使っているところを推察するとやってはいけないと思われる。そこでドライバインターフェース readLinux 上にバッファを設け、irp->AssociatedIrp.SystemBuffer（名が長いため図中では buffer と省略）をここに設定する。そうすることで、readWin 内でのプロセスへのデータ移動のあて先は readLinux のバッファになる（図中①）。全ての処理が終了してから readLinux のバッファをカーネル関数 copy_to_user を用いてプロセスへデータを移動させる（図中②）。



テンプレートファイルは C 言語で書く、デバイス名のようなデバイス依存する情報、変換時に定義したい情報は、前もって定義できないためプログラム中のそういったデータについてはマクロにしておく。そして、”xxx.h”をインクルードしておく。変換時に xxx.h ファイルを作成し、マクロで定義されている情報を xxx.h ファイルに書き込む。C 言語のプリプロセッサでマクロが展開され、本当の情報がプログラムにでてくる。



3.4. カーネル関数エミュレート

逆アセンブル時にいくつか Windows カーネル関数呼び出しが出現する。これらの関数は Linux デバイスドライバからは実行以前にインストール時に弾かれる。

そこで、Windows カーネル関数をエミュレートする関数を定義する。この関数をの名前はエミュレート対象となる Windows カーネル関数と同名にする。そうすることでリンク時にエミュレート関数が呼ばれるようにアドレスを設定してくれる。Windows カーネル関数が呼ばれたときにエミュレート関数が呼ばれる。つまり、デバイスドライバ内に参照するアドレスが存在するために、最終的に生成される Linux デバイスドライバで外部を参照するためのシンボル名として Windows カーネル関数名が登録されることもなく Linux デバイスドライバのインストール時にエラーがでることもない。

例として Windows カーネル関数 ExAllocatePoolwithTag を用いて Windows カー

ネル関数のエミュレートがどのように行われているか説明する。

① Windows デバイスドライバがメモリ確保のエミュレート関数 ExAllocatePoolwithTag を呼ぶ。

②エミュレート関数 ExAllocatePoolwithTag はメモリ確保の Linux カーネル関数 vmalloc を呼び、vmalloc がメモリを確保する。

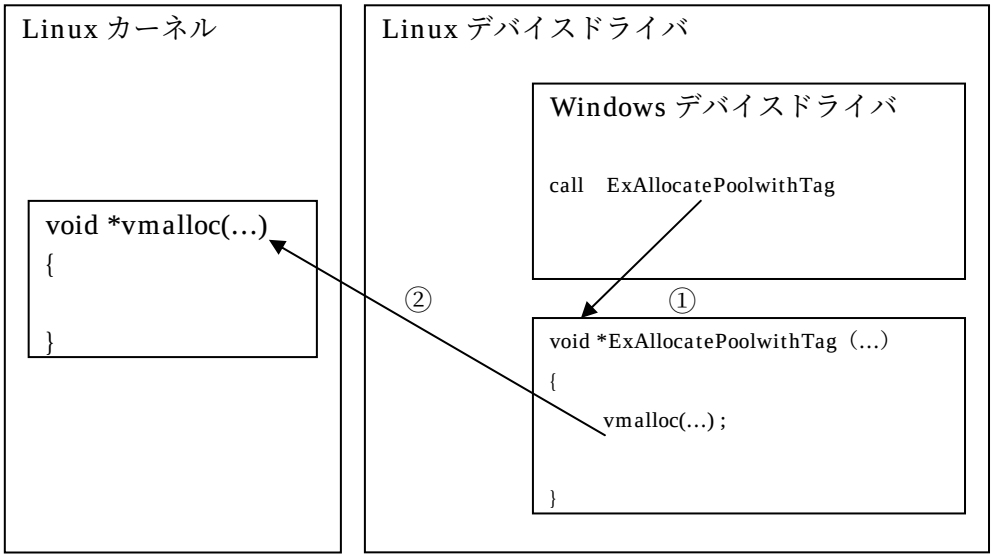
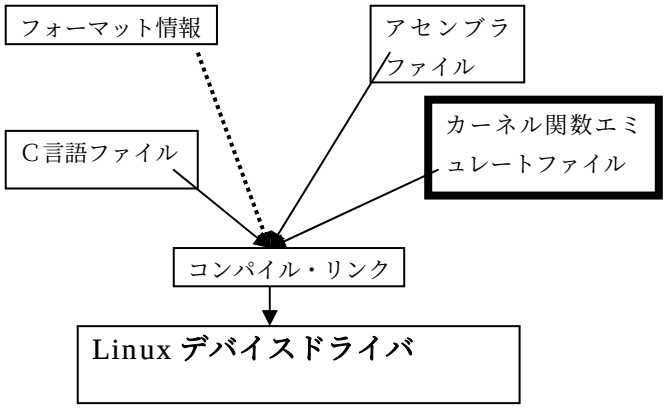


図 3-3 Windows カーネル関数をエミュレート

vmalloc と ExAllocatePoolwithTag のように Linux カーネル関数と Windows カーネル関数でよく似た機能の関数が存在するのでエミュレートもプログラムし易い。

カーネル関数エミュレートファイルはドライバによって動作及びデータを変えるようなことはないため、オブジェクトファイルで提供する。エミュレートファイル関数はＣ言語で記述、コンパイルにアセンブラファイルを出力させるようにする。そしてアセンブラファイルを編集し、アセンブルを行い、オブジェクトファイルつくる。一度アセンブラファイルを出力、編集する理由は、関数への引数の廃棄のタイミング（2.4 節）に関係がある。エミュレート関数は Windows デバイスドライバから呼ばれる関数であり引数の廃棄はエミュレート関数内で行わなければならないが、gcc を用いてエミュレート関数をコンパイルすると引数の廃棄を関数内では行わないコードを出力する。Ｃ言語レベルで引数の操作を行うことはできないためにアセンブリ言語レベルでの編集を行った。編集箇所は関数の最後の命令 “ret” を “ret XX” とした。XX は引数に使用されるバイト数である。

3.5. コンパイル・リンク



今までの過程で生成したファイルをコンパイル・リンクする。カーネル関数エミュレートファイルはカーネル関数毎に用意してあるため、使用しない関数までコンパイル・リンクするのはドライバがインストールされたときのメモリを余分に消費するので無駄である。そこで、使用する関数のみをコンパイル・リンクするための情報をフォーマット情報（3.1 節のＤ）から得る。

Linux デバイスドライバを作成するためにコンパイルを行うので、生成される Linux デバイスドライバのバイナリフォーマットはＥＬＦである。故に、図３-1 の・順で Linux デバイスドライバを生成すると、Windows デバイスドライバ、Linux デバイスドライバのバイナリフォーマットの違いは解消される。

4. 実装

4.1. フォーマット解析,逆アセンブル

図３．２を実現するには disasm で buf のデータを基にどの命令（名前、オペランド数、オペランドの種類）かという情報を変数 ins に出力し、outasm は ins を基に文字列 asmbler に NASM 形式のアセンブラ文字列を出力し、ファイルへ文字列を出力する。

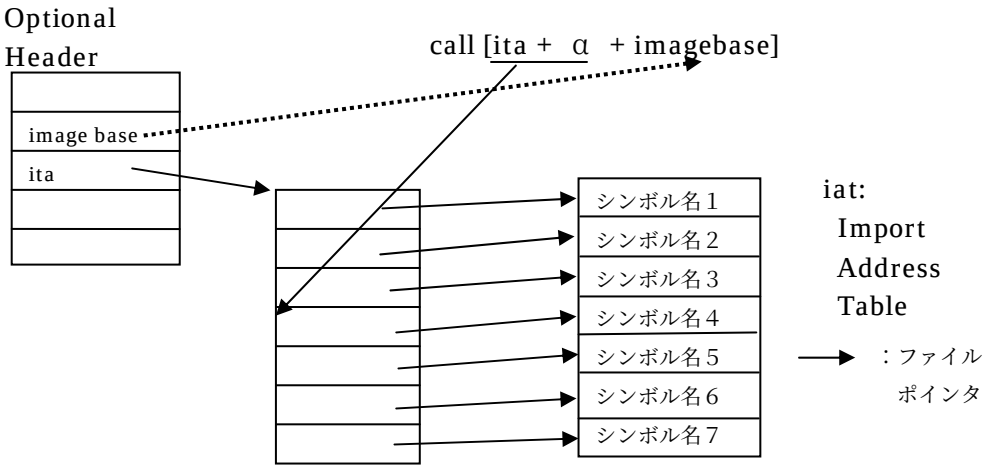
```
ｆread(buf,データ)
do {
    disasm(buf,ins);

    outasm(ins,asmbler);

    fprintf(File,asmbler);

}while(ins.name!=”ret”)
```

その際に外部のシンボル（カーネル関数名）を参照するためにシンボル情報が必要であるのでフォーマットについて説明を行う。外部シンボルを扱うためフォーマット情報は以下のようになっている



フォーマット解析の段階で、ita+α+imagebase **の値（アドレス）とシンボル名を**

持つテーブルを作成しておき、call [xxx] のような間接アドレッシングのオペランドをもつコール命令であった場合はテーブルを参照して、xxx と同じアドレスを持つものがあれば、カーネル関数であるためオペランドを対応するシンボル名にする。上の例では call シンボル名４のようになる。

4.2. インターフェース生成

インターフェースを記したテンプレートファイルはWindows デバイスドライバを逆アセンブルした関数のインターフェースと Linux デバイスドライバを動作させる上で最低限必要な処理関数とデータを用意する。

- 初期化関数 init_module
- ドライバ削除関数 cleanup_module
- システムコールを処理する関数
  createLinux,readLinux,writeLinux,closeLinux
- 上の処理で使用されるグローバルデータ
- 上の処理をコンパイルするのに必要なヘッダ一式の include

4.2.1. 初期化関数 init_module,削除関数 cleanup_module

デバイスドライバは init_module,cleanup_module を必ず用意しなければならない。今回は最低限必要な処理を記述したものを用意した。
init_module ではシステムコールを処理する関数をカーネルへ登録するために register_chrdev 関数を呼ぶ。

```
int Major ;
char *Devname = “DEV_NAME” ;
struct file_operations fops = OPERATIONS ;
int init_module(void)
{
   Major = register_chrdev(0,Devname,&fops) ;
}
```

データ

Major	デバイスのメジャー番号
Devname	デバイス名
fops	システムコールを処理する関数へのポインタを集めた構造体

Devname はデバイスによって変わるためマクロ『DEV_NAME』を代入しておく。

同様に fops もデバイスによっては一部のシステムコールは使用しない可能性が考えられるためマクロ『OPERATIONS』を代入しておく。

cleanup_module ではデバイスをカーネルから削除する関数 unregister_chrdev を呼ぶ。

4.2.2. createLinux,closeLinux

createLinux ではデバイスの利用度数（カーネルが使用する）を増やすマクロ MOD_INC_USE_COUNT を書いておく。その後に Windows デバイスドライバを逆アセンブルした関数 createWin を呼ぶ。

closeLinux では MOD_DEC_USE_COUNT マクロで利用度数を減らし、closeWin を呼ぶ。

4.2.3. readLinux,writeLinux

readLinux、はデータ移動に関する処置を施し（①）、逆アセンブラした関数 readWin を呼び（②）、最後にプロセスへのデータを移動する③。

```
ssize_t readLinux( struct file *filp,char *buf,size_t count,loff_t **ppos )
{
    IRP irp;
    DEVICE_OBJECT fdo;
    int copyed_count;

    char *bufTmp = vmalloc( count ) ;                    ①
    irp.Parameters.Read.Length = count ;
    irp.AssociatedIrp.SystemBuffer = bufTmp ;

    copyed_count = winRead(&fdo,&irp) ;      ②

    copy_to_user(buf,bufTmp,count);          ③
    return copyed_count ;
}
```

変数型 IRP,DEVICE_OBJECT 型は Windows が定義する型である。今回は全てのフィールドと機能を解析することができなかったため、read するバイト数を表す Parameter.Read.Length, ユーザバッファのアドレスを示す AssociatedIrp.SystemBuffer のフィールドのみを定義した。

writeLinux についてはバッファを確保し、引数 irp を設定、ユーザプロセスからデバイスへ書くためのデータをバッファへ copy_from_user を用いて読み込み、Windows デバイスドライバを逆アセンブルした関数 writeWin を呼び出す。

4.3. Windows カーネル関数のエミュレート

今回は 4 種類の Windows カーネル関数をエミュレートした。エミュレートした際に使った処理と記述しておく。

- ExAllocatePoolwithTag
  メモリ確保関数、内部で Linux カーネル関数 vmalloc を呼ぶ。
- ExFreePool
  メモリ解放関数、内部で Linux カーネル関数 vfree を呼ぶ。
- READ_PORT_UCHAR
  IO ポートを読む関数、ＩＮ命令を実行する。
- WRITE_PORT_UCHAR
  IO ポートへデータを書く関数、ＷＲＩＴＥ命令を実行する。

5. 検証

本章では Windows デバイスドライバから変換して生成された Linux デバイスドライバの動作の確認実験とパフォーマンスについて述べる。

確認実験としてメモリデバイスのデバイスドライバ、IO ポートアクセスのデバイスドライバの Windows デバイスドライバを作成、変換を行った。

実検に使用した環境は次のようなものである。

CPU	Celeron	
	動作速度	487.504MHz
	キャッシュ	128KB
OS	Linux ver2.2.18	
その他	普通のＰＣ	

以下、実験１、実験 2 の説明においてまず、デバイスドライバの仕様を述べ、使用される Windows カーネル関数（エミュレートしたもの）と、実験方法を述べ、その補足を行う。最後に結果について述べる。

5.1. 実験１ メモリデバイス

【 デバイスドライバの仕様 】

カーネル空間のメモリ空間の１部(8k バイト)をデバイスドライバに割り当て、この 8k バイトのメモリ領域をデバイスとする。

デバイスドライバはメモリ（デバイス）を読み書きする。

【 使用した Windows カーネル関数 】

- ExAllocatePoolwithTag
- ExFreePool

【 実験方法 】

１．8k バイトのデータを write する。

２．8k バイトおデータを read する。

３．write したデータと read したデータを比較する。

【 結果 】

write したデータ（基のデータ）と read してきたデータが一致していた。

この結果からプロセス⇔デバイスドライバ間のデータの移動に関しては正常に動作しているようである。また ExAllocatePoolwithTag、ExFreePool のエミュレートもうまく動作している。

5.2. 実験 2　IO ポートアクセスデバイスドライバ

【 デバイスドライバの仕様 】

プロセスが指定した IO ポートを読み書きする。

システムコール側の呼び出しは

read(fd,buf,port)

機能：port で指定された IO ポートから buf を読む。

CPU 命令の in に相当する。

write(fd,buf,port)

機能：port で指定された IO ポートへ buf のデータを書く。

CPU 命令の out に相当する。

【 使用した Windows カーネル関数 】

- WRITE_PORT_UCHAR
- READ_PORT_UCHAR

【 実験方法 】

RTC (Real Time Clock)から現在の時刻を得る。

RTC から得た時刻と現在の時間を比較する。

Windows2000、Linux では安全性の問題から通常のユーザプロセスから IO ポートを気軽にアクセスすることはできない。今回作成した IO ポートアクセスデバイスドライバはユーザプロセスから IO ポート番号とデータを受け取り、ユーザプロセスに代わりにデバイスドライバが指定された IO ポートへデータを読み書きする。

RTC は時計機能をもったデバイスである。時刻情報は RTC の複数のレジスタに格納されており、0x70 番ポートで RTC のレジスタを選択（番号の値を書く）、0x71 番ポートから情報を読み出す。下にレジスタ情報を示す。

番号	機能
00	秒カウンタレジスタ
02	分カウンタレジスタ
04	時カウンタレジスタ
06	曜日レジスタ
07	日　レジスタ
08	月　レジスタ
09	年　レジスタ

全情報を読むのに 7 回の write,read を繰り返す。

【 結果 】

正しい時刻を得ることができた。

結果から WRITE_PORT_UCHAR、READ_PORT_UCHAR のエミュレート関

数は正しく動作するようである。

5.3. 性能の比較

メモリデバイスドライバを通常の方法で作成する。作成したデバイスドライバと変換したデバイスドライバの実行時間を測り、比較する。

5.3.1. 時間の測り方

CPU 命令 rdtsc を使用して時間を計測する。rdtsc 命令は CPU が動作を始めてからのクロック数を 64 ビットの値で読むことができる。CPU の動作速度が 500MHZ であれば、１秒後に rdtsc 命令を実行すると 500Mをプラスした値が読み出される。カーネル空間での時間を測るのに使用する変数 jiffies 値は時間間隔が 100HZ と長いため、またカーネル関数 do_gettimeofday より実行時間が短いため時間計測に消費される時間が短いためより正確に時間が測れる。

時間計測のマクロ get_time 定義する。これはインラインアセンブラに展開される。なお、__asm__は gcc に依存の予約語であり、アセンブラプログラムを C 言語中で組めるようなっている。

```
#define   get_time(h,l)     ＼
          __asm__(“rdtsc” : “=a”(l),”=d”(h) )
```

rdtsc で読み込んだデータ（eax,edx レジスタへ格納）を上位 32 ビット値を変数 h に、下位 32 ビット値を変数 l に格納する。

以下のように get_time をデバイスドライバに組み込んで時間を計測する。

```
unsigned long long time1,time2 ;
unsigned long h1,l1,h2,l2 ;
get_time(h1,l1) ;
処理
get_time(h2,l2);
time1 = (unsigned long long)h1<<32 | l1 ;
time2 = (unsigned long long)h2<<32 | l2 ;
return   time1 – time2 ;
```

本来 return 値は読み込みバイト数であるが、今回は時間を測るという実験ということで多少変則的ではあるが、return 値を処理にかかる時間とした。

get_time をプログラム中に埋め込むタイミングは 3.3 節で述べた、インターフェース作成時であり、生成されるＣ言語ファイルを編集する。

5.3.2. 実験３ 比較

【 デバイスドライバ 】

１．4.1 節のメモリデバイスドライバ(Windows デバイスドライバを変換)

２．通常の方法で作成したメモリデバイスドライバ(機能は１と変わらず)

【 実験方法 】

マクロ get_time をデバイスドライバのプログラムに埋め込む。

１と２のメモリデバイスに対して read を実行し、その時間を計測する。

転送バイト数は１、16、256、1024、2k、4k、8k を試行する。

各 20 回実行する。

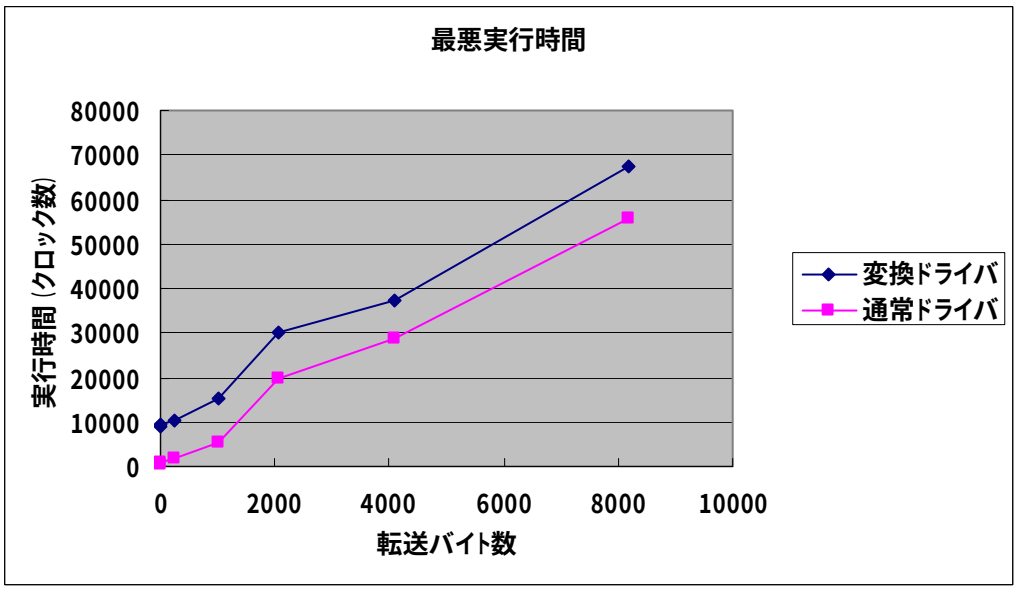
【 結果 】

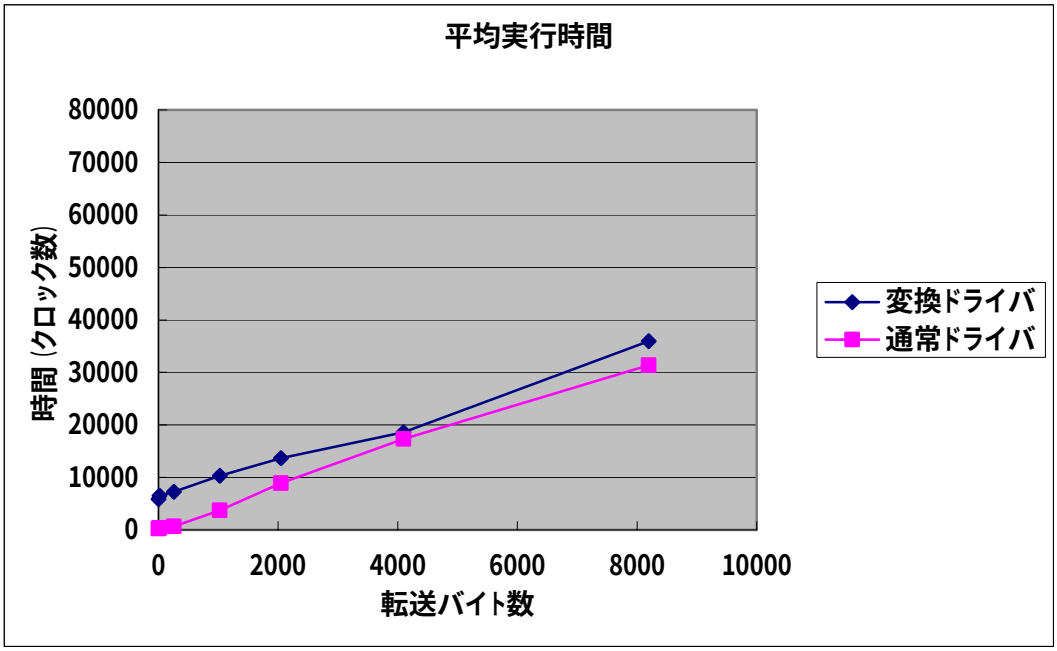
最悪実行時間（クロック数）

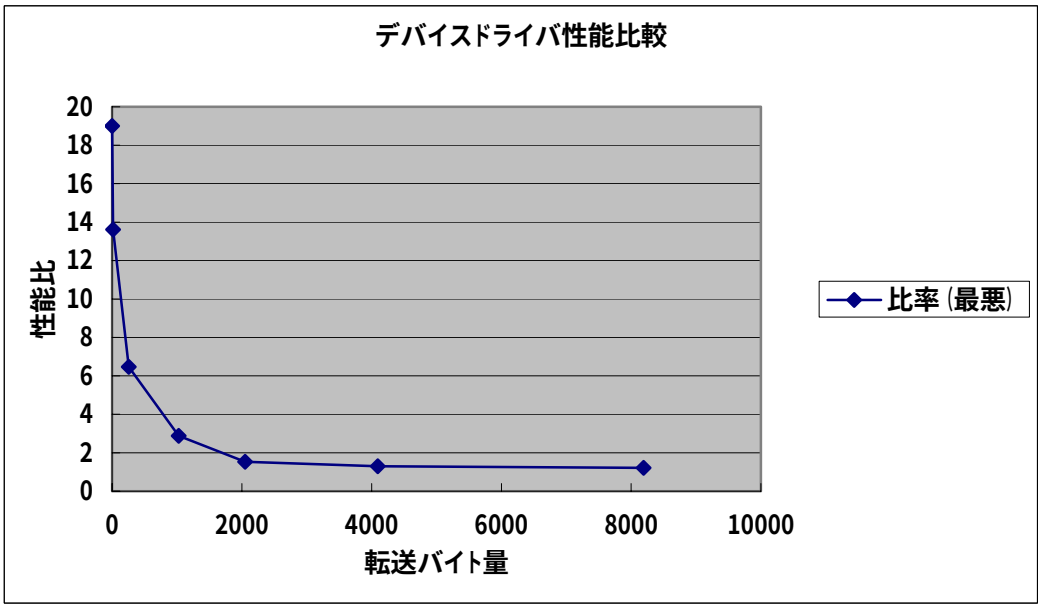
	１バイト	1024 バイト	８ｋバイト
変換ドライバ	8867	15234	67620
通常ドライバ	457	5294	55860

平均実行時間（クロック数）

	１バイト	1024 バイト	８ｋバイト
変換ドライバ	5872	10292	35956
通常ドライバ	226	3713	31363







最悪実行時間は最初の１回目の測定に現れるようである。キャッシュが働いていると思われる。

変換したデバイスドライバと作成したデバイスドライバの間には最悪で 8000～12000 クロック、平均で 5000～6000 ぐらいのある一定の開きがあるようである。こ

の時間が変換の仕組みの処理に費やされる時間であると考えらる。
　変換したデバイスドライバと作成したデバイスドライバの時間を比率で示したグラフをみると少数バイト量での転送の方が、オーバーヘッドが目立つ。最低必要な処理にかかる時間が累積するため、少バイト転送を複数繰り返すようなプログラムでは変換に用いた仕組みのオーバーヘッドが大きくなると思われる。

6. まとめ

Windows デバイスドライバのバイナリファイルを逆アセンブルを用いて解析し、インターフェースを整え、Windows カーネル関数をエミュレートすることで Linux デバイスドライバへの変換に成功した。

また実験結果から、変換したデバイスドライバは多バイト量転送の実行時間においては通常の方法で作成したデバイスと遜色ない性能を示していることを確認した。

今後の課題としては今回扱わなかった処理を変換できるようにすることである。特に時間に関する処理、割り込みの処理の変換を行わなければならない。またそれにともない、カーネル関数エミュレートする数を増やさなければならない。Microsoft Windows2000 Driver Development Kit Documentation[2]載されている Windows カーネル関数のみを数えても 650 種類が存在している。このうち半分以上はマクロで定義[1]されていると推定しても、現時点で実現した４種類では少ない。

問題点として、ＯＳのヴァージョンアップが行われると変換プログラムが使えなくなるということである。Linux に関しては、違うヴァージョン間のデバイスドライバの互換性はなくデータ構造が変化はしている。ヴァージョン間の互換性がないという傾向は今後も続いていくだろうといわれている。Linux、Windows のヴァージョンアップが行われたさいにどのように対応するかも考えなければならない。

[1] ExAllocatePool は ExAllocatePoolWithTag のマクロである。

参考文献

[1] Microsoft Portable Executable and Common Object File Format Specification Revision6.0 , 1999

[2] Microsoft Windows2000 Driver Development Kit Documentation,2000

謝辞

修士論文を書くにあたって御指導してくださった田中章司郎教授に深く御礼申し上げます。
共に学んだ同じ研究室の方々にも深く御礼申し上げます。
また、研究に使わせていただいた有益なツールの開発者、配布者に深く御礼申し上げます。

ありがとうございました。