MMUなしプロセッサ用Linuxの共有ライブラリ機構

大谷 浩司、 高岡 正、 近藤 政雄、臼田 尚志
株式会社 アックス

1 はじめに

MMUがないプロセッサ用のLinuxとしてμＣｌｉｎｕｘ [1](以降は便宜上uClinuxと記す) が開発されている。

uClinuxでは、仮想記憶機構をサポートしないため、通常のLinuxに比べ幾つか制限が存在する。共有ライブラリ機構もそのひとつである。通常のLinuxの共有ライブラリ機構は、仮想記憶機構を利用して実現されているため、uClinuxでは動作しない。しかし、共有ライブラリはメモリ消費抑制、ソフトウェア保守などに有効であり、実現の要望が強い。

我々は、このたびuClinux向けの共有ライブラリ機構を開発し、実装したので報告する。

以下では、まず、uClinux上の共有ライブラリ機構の現状を述べた後、我々の開発した方式の利点を述べる。次に、方式の概要を説明し、続いて詳細を述べる。最後に他の方式を説明する。

2 uClinux上の共有ライブラリ機構の現状

現在、ARMプロセッサとColdFireプロセッサにのみ、共有ライブラリ機構が存在することが知られている。ARMプロセッサについてはRidgeRun社が2002年の3月に開発を発表したが、残念ながらRidgeRun社は活動を停止した。ARMの共有ライブラリ機構は、現在、Cadenux社 [2]が受け継いでいる。[3] ColdFireプロセッサについては、SnapGear社 [4]が、2002年の4月に開発を発表し、提供している。[5]

これらの方式については、後ほど説明する。

3 共有ライブラリ機構の利点

我々の開発した共有ライブラリ機構は、通常のLinuxのものの利点をほとんどすべて受け継いでおり、以下の利点を持つ。

1. ライブラリ共有によりメモリ消費を抑制できる。

   ひとつのプログラムだけで考えれば、共有でない場合よりもメモリ消費は多くなる。しかし、共有する部分が多ければ、システム全体としてのメモリ消費は少なくなる。これは、ライブラリの関数を多く使用する複雑なプログラムが多いほど効果が大きい。

2. プログラムのコード共有により、メモリ消費を抑制できる。

   プログラムのコード等も複数のプロセスで共有可能であるため、シェルなど複数のプロセスが動作するプログラムの場合は、大きくメモリ消費を抑制できる。

3. 実行オブジェクト・ファイルが小さくなり、ファイル容量を削減できる。

   各アプリケーションに重複して含まれていたライブラリのコードが、各ファイルに必要でなくなるため、ファイルシステム全体としてファイル容量を大きく削減できる。

4. ライブラリを修正してもプログラムのリンクをやり直す必要がない。

   プログラム起動時に動的にリンクするため、ライブラリに変更があっても実行が可能である。このため、プログラムの再リンクが必要でなくなり、プログラムの保守が容易になる。また、プログラムのオブジェクトの変更なしに、ライブラリの特別バージョンを利用する事もできる。

5. 通常の Linux のものと比較して実行オブジェクト・ファイルが小さい。

   uClinux で利用される実行ファイル形式は bFLT と呼ばれ、実行オブジェクト・ファイルが小さい。今回開発した方式は、bFLT 形式を拡張した実行ファイル形式を採用し、通常の Linux の ELF 形式に比べて、40%から 15%ほど小さい。

6. XIP(eXecute In Place) に対応している。

   ロード時にコード部分を一切変更しないため ROM 化した場合などに、RAM 上にコピーせずに実行させる事が可能である。これは、メモリ消費を大きく抑制する効果がある。

4 方式の概要

本方式の理解を容易にするために、まず、既存の uClinux でのプロセスのメモリ上のイメージを説明する。

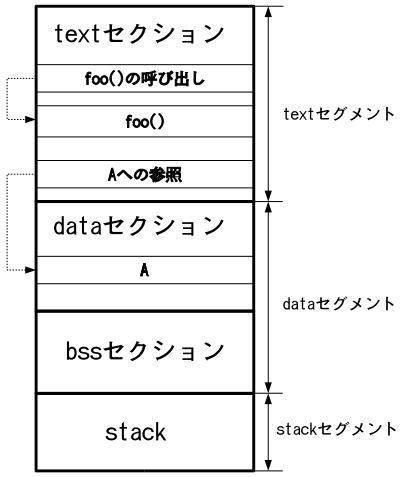

図 1: プロセスのメモリ上のイメージ

図 1 に示すように、イメージは、text セグメント、data セグメント、stack セグメントの三つのセグメントから構成される。text セグメントには、コードや書き換え不可のデータが入る text セクションが含まれる。data セグメントには、明示的に初期化されたデータの入る data セクション、明示的に初期化されていないデータが入る bss セクションが含まれる。stack セグメントには、スタックが割り当てられる。通常の Linux とは異なりスタックはプロセス毎に固定サイズが割り当てられ、拡大する事はない。

text, data, stack の各セグメントは、この順にメモリ上に連続して割り当てられる。

コードは PIC(Position Independent Code) になっている。関数呼び出し、データ参照はともに、セグメントがメモリ上で連続している事を前提に PC(Program Counter) 相対で行なわれる。

次に、共有ライブラリ機構でのプロセスのメモリ・イメージを図 2 に示す。

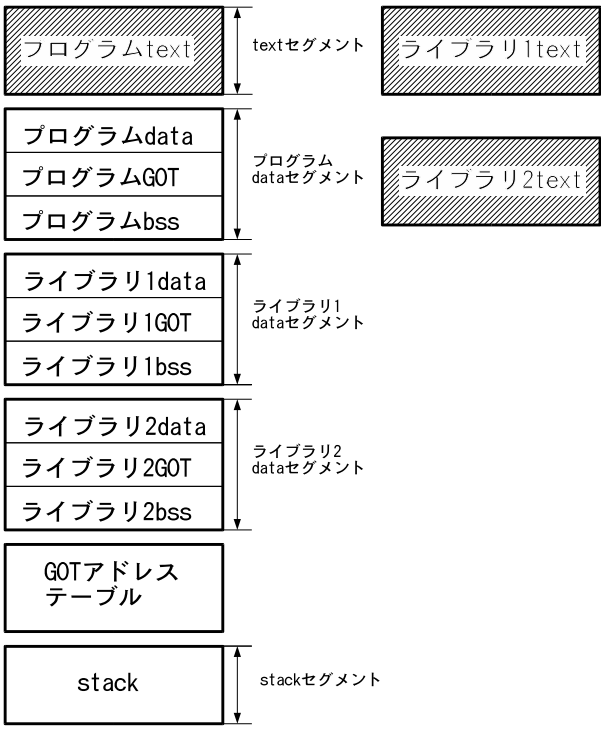

図 2: 共有ライブラリ機構のメモリ上のイメージ

図 2 は、プログラムが共有ライブラリ 1 と共有ライブラリ 2 を利用する場合を示している。

共有ライブラリ機構では、text セグメントを各プロセスで共有し、data セグメントを各プロセス固有のものを割り当てる。プログラムと各ライブラリの text セグメントは、メモリ上で独立に存在する。プロセスを

起動した時に、そのプロセス用の data セグメントと stack セグメントが確保される。data セグメントは、さらにプログラムと各ライブラリ用の data セグメントに分かれている。

プログラムまたはライブラリ (以降は、単にモジュールと記す) 用の各 data セグメントには、GOT(Global Offset Table) セクションが追加されている。他のモジュールへの参照は、GOT のエントリを介した間接参照になっている。

また、各モジュールの data セグメントの次には、GOT アドレス・テーブルが存在する。GOT アドレス・テーブルは、各モジュールが自身の data セグメントのアドレスを知るために使用される。

各モジュールでは、自身の data セグメントをアクセスするのに、ベースレジスタ相対で行なっている。各ベース・レジスタは、呼び出された関数の入口で GOT アドレス・テーブルを利用して GOT の先頭を指すように設定される。

次章では、このイメージを利用して如何に共有ライブラリが動作するかを詳細に述べる。

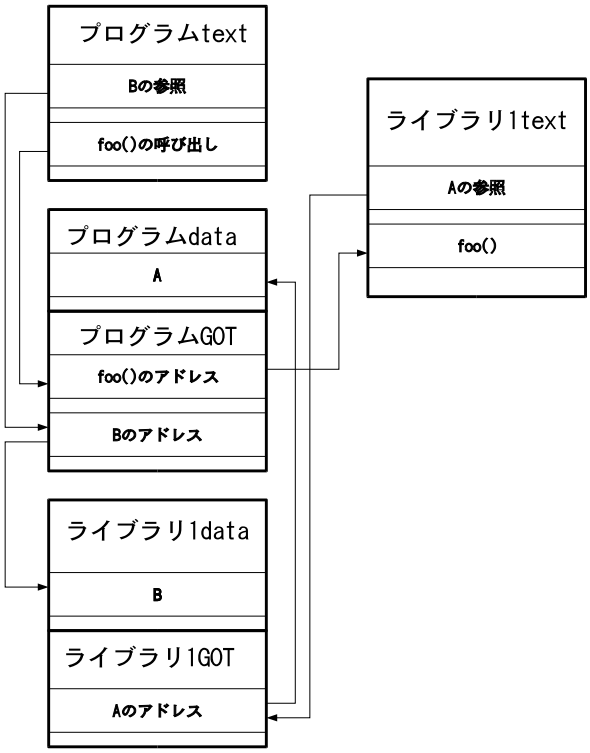

図 3: GOT を使った参照

5 方式の詳細

5.1 GOT(Global Offset Table)

共有ライブラリなどで、text セグメントを書き換えないで動的にリンクする場合には、GOT という手法が知られている [6]。本方式でも GOT を採用している。

共有ライブラリにおいて、他のモジュールの変数や関数などのオブジェクトを参照する場合、そのアドレスは、モジュールを作成した時には未定である。そのため、プログラムのコードから直接参照すると、ロード時にコードを変更する必要が出てくる。しかし、本方式のように ROM 化を許したり、共有したりする場合には text セグメントを変更することはできない。そのため、他のモジュールのオブジェクトを参照する場合は、自身の data セグメントを介して間接的に参照する。この間接参照を行なうためのテーブルを GOT(Global Offset Table) と呼ぶ。GOT の各エントリには、他のモジュールのオブジェクトのアドレスが入る。GOT のエントリに正しいアドレスを設定するのは、動的リンカ&ローダの責任である。

図 3 に GOT を利用した参照の例を示す。図 3 では、プログラムは共有ライブラリ 1 を利用している。プログラム側に変数 A、ライブラリ 1 側に変数 B、関数 foo() が存在している。各 data セグメントに、GOT を設ける。プログラムから、ライブラリ 1 の変数 B を参照する場合には、プログラムの GOT のエントリを介して間接参照する。また、プログラムからライブラリの関数を呼ぶ出す場合にも、GOT からそのアドレスを取得後呼び出す。ライブラリ 1 からプログラムの変数 A を参照する場合には、ライブラリ 1 の GOT のエントリを介して間接参照する。

5.2 data セグメントのアクセス方式

text セグメントと data セグメントは、メモリ上で分離され、その配置はロード時になるまで決定されない。また、text セグメントを書き換えてはいけない。そのため、コードからは絶対アドレスを使って data セグメントをアクセスする事はできない。さらに、text

セグメントと data セグメントの相対的な位置もロード時になるまで決定されないため、PC 相対でアクセスする事もできない。そこで、data セグメントを指すベースレジスタを設けて、このベースレジスタ相対で data セグメントをアクセスする事にした。このベースレジスタは、data セグメントのどこを指していても良いのだが、中心の辺りを指す方が正負両方のオフセットが利用でき、より多くのデータのアクセスが容易になる。本方式では GOT セクションを data セグメントの中心部に配置し、GOT セクションの先頭を指す事とした。このベースレジスタを GOT レジスタと呼ぶことにする。data セクションは負のオフセット、bss セクションは正のオフセットを使ってアクセスする。

data セグメントをアクセスする場合のオフセットは、モジュール作成時に決定されていなければいけない。共有ライブラリでは、実行時にどのようなモジュールとの組合せで、利用されるかは不明である。そのため、ただひとつの data セグメントを割り当てる方法は、オフセットが決定できないために、採用できない。そこで、各モジュール毎に data セグメントを用意し、各モジュールを実行する場合には、図 4 に示すように、そのモジュールの data セグメントを指すように GOT レジスタを設定する。

ここで、問題となるのが、GOT レジスタにいつどのようにして正しい値を設定するかである。次項では、この問題と本方式で採用した解決法について説明する。

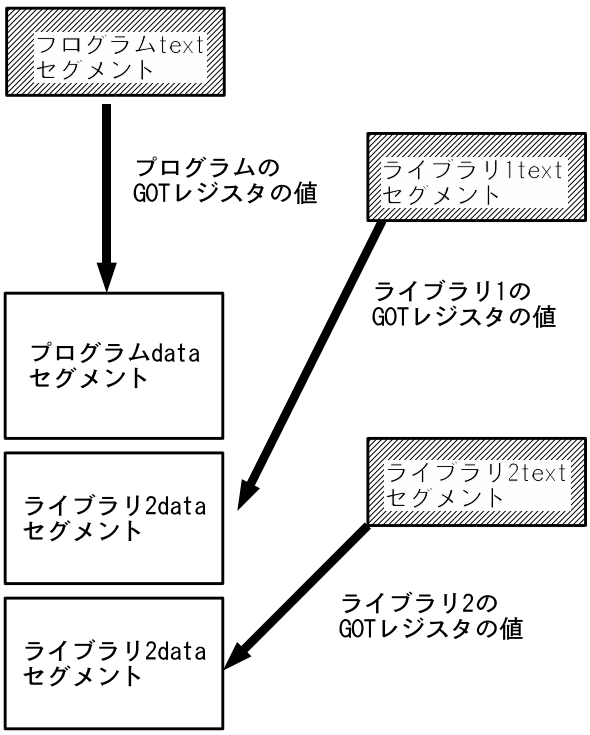

図 4: GOT レジスタ

5.3 GOT レジスタの設定方法

まず、プログラムの GOT レジスタは、メモリ配置を知る事が可能な動的リンカ&ローダが設定する。GOT レジスタの変更が必要になるのは、他のモジュールの関数を呼び出す時である。設定方法は、大きく分けて、呼び出すモジュールで行なう方法と、呼び出されたモジュールで行なう方法の二つある。

呼ぶ出すモジュールで行なう方法には、次の欠点がある。

関数のポインタを他のモジュールに渡した場合を考える。他のモジュールでは、そのポインタを利用して、関数を呼ぶ出す事になる。しかし、呼ぶ出す場所では、そのポインタがどのモジュールの関数を指すポインタかを区別する手段がない。従って、GOT レジスタを正しく設定する事はできない。そのため、呼ぶ出すモジュールで設定する方法では、関数へのポインタを他のモジュールに渡す事ができないのである。

我々は、呼び出されたモジュールで設定する方法を採用している。実際には、関数の入口で、GOT レジスタを設定する。これは、他のモジュールから呼ばれる関数だけで良い。しかし、関数のポインタの事を考えると、その関数が他のモジュールから呼ばれるかどうかは、簡単にはわからない。そこで、すべての data セグメントをアクセスする関数で GOT レジスタの設定を行なっている。

次に、自身の data セグメントのアドレスを知る方法が問題となる。以下に、本方式で採用している GOT レジスタの設定方法を述べる。

まず、共有ライブラリに 1 から始まる番号を付ける。この番号をライブラリ番号と呼ぶ。プログラムには、番号 0 を割り当てる。ライブラリ番号順にモジュールの GOT の先頭アドレスを並べたテーブルを data セグメントに用意する。これを GOT アドレス・テーブルと呼ぶ。各モジュールの GOT の先頭のエントリに

は、GOT アドレス・テーブルの先頭のアドレスを入れておく。図 5 に、例を示す。この例では、共有ライブラリ 1 と共有ライブラリ 2 を利用しており、各々ライブラリ番号 1 と 2 が割り当てられている。

このようにすると、自身のライブラリ番号を知っていれば、自身の GOT の先頭アドレスは、次の手順で簡単に得られる。

1. 現在の GOT レジスタの指すエントリから、GOT アドレス・テーブルの先頭アドレスを取得する。

   すべての GOT の先頭エントリには、GOT アドレス・テーブルの先頭アドレスが入っているので、どのモジュールから呼ばれたとしても正しい値が得られる。

2. 自身のライブラリ番号を n とすると、GOT アドレス・テーブルの n 番目のエントリには、自身の GOT の先頭のアドレスが入っている。これを GOT レジスタに設定すれば良い。

GOT アドレス・テーブルのエントリが衝突しないためには、共有ライブラリのライブラリ番号は、システムでユニークでなければならない。本方式では、コードの簡単さ、高速性を考えて、共有ライブラリ生成時にユニークな番号をコード中に埋め込むことにした。

そうすると、プログラムが利用していない共有ライブラリの GOT アドレス・テーブルのエントリは、空欄となり無駄が生じる。しかし、この無駄は仮に 100 エントリあっても 400 バイトであり、許容できるものであると我々は考える。

GOT アドレス・テーブルを作成し、エントリを設定するのは動的リンカ&ローダの責任である。

5.4 動的リンカ&ローダの動作

動的リンカ&ローダは、以下の手順でプログラムをロードする。

1. ファイルからプログラムの text セグメントをメモリ上にロードする。既にメモリ上に存在すれば、それをそのまま利用する。

2. プログラムの data セグメントをメモリ上に確保し、data セクション、bss セクションを初期化する。

3. プログラムが参照しているライブラリの text セグメントをメモリ上にロードする。既にメモリ上に存在すれば、それをそのまま利用する。

4. プログラムが参照しているライブラリの data セグメントをメモリ上に確保し、data セクション、bss セクションを初期化する。ただし、このプロセス用に既に、このライブラリの data セグメントが作成されていれば、作成しない。

5. ライブラリが参照するライブラリについても同様に、text セグメントのロード、data セグメントの作成を行なう。これは、連鎖的に参照しているライブラリがすべてメモリ上にロードされるまで繰り返す。

6. GOT テーブルをメモリ上に確保する。

7. 以下を各モジュールについて行なう。

   (a) GOT の先頭アドレスを GOT アドレス・テーブルの対応エントリに設定

   (b) GOT の初期化

   (c) リロケーション

   (d) DATA セクションおよび GOT セクションのインポート・シンボルを解決

8. スタックをメモリ上に確保し、初期化

9. プログラムの GOT レジスタを設定して、プログラムを実行開始

この手順のなかで、text セグメントをロードする部分では、既にロードしている場合には、それを共有して利用する必要がある。通常の Linux においては、この処理は、共有属性を指定してカーネルの mmap を呼び出し、ファイルをメモリ上にマップする (以降は、便宜上、共有 mmap と記す) ことで実現できる。しかし、uClinux では、ファイルの共有 mmap は一般には、実装されていない。そのため、本方式では、IPC(Inter Process Comunication) の共有メモリを改造して実現している。

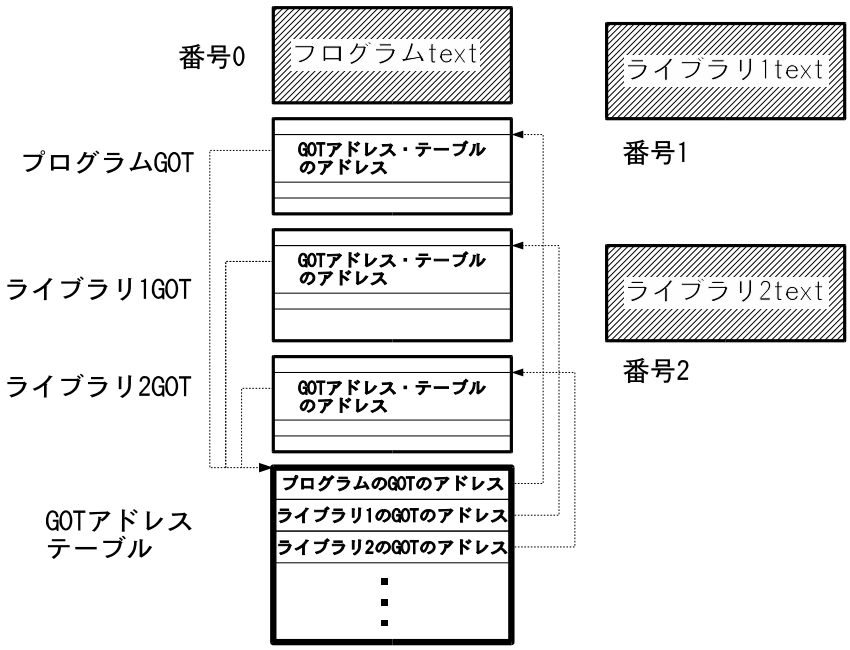

図 5: GOT アドレス・テーブル

5.5 XIP(eXecute In Place)

ROM や RAM などメモリ上のファイル・システムでの mmap が直接そのメモリ上のポインタを返すように実装されている場合は、XIP が実現できる。実験的に romfs に実装してみたところ、正常に動作する事が確認できた。

5.6 初期化ルーチンの問題

C++などの言語では、main ルーチンを実行する前に、初期化ルーチンを実行する必要がある。これは、各ライブラリでも必要である。

共有ライブラリでない場合は、すべての初期化ルーチンは、CTOR セクションにエントリアドレスが集められリストになっている。このリストに基づいて、プログラムの__main から呼び出される。

しかし、このままでは、共有ライブラリの場合、ライブラリ側の初期化ルーチンはプログラムの初期化ルーチン・リストに入らないため、初期化が行なわれない。

そこで、本方式では以下のように、この問題を解決している。

1. ライブラリ・モジュールにも__main に相当するルーチンを設ける。これを initializer と呼ぶ事にする。このルーチンでは、そのライブラリの初期化ルーチンをリストに基づいて呼び出す。

2. 初期化ルーチン・リストに、そのモジュールが参照しているライブラリの initializer のアドレスを加える。これにより、プログラムの__main から連鎖的にすべての共有ライブラリの初期化ルーチンが実行される事になる。

3. 複数のモジュールから参照されているライブラリの initializer は、複数回呼ばれるので、initializer 内部では、実際の処理は 1 度のみ実行するようにする。

5.7 関数の置換えの問題

既にライブラリ中で定義されている関数をプログラムの独自のもので、置き換えたい場合がある。例えば、標準 C ライブラリで定義されている malloc を独自のアルゴリズムのものに置き換えたい場合などである。

しかし、本方式では、モジュール内の関数呼び出し

は、GOT を介さずに直接行なわれるため、置き換える事ができない。例えば、標準 C ライブラリとライブラリ X を利用したプログラムがあり、プログラム中に独自の malloc を定義したとすると、図 6 のようになる。

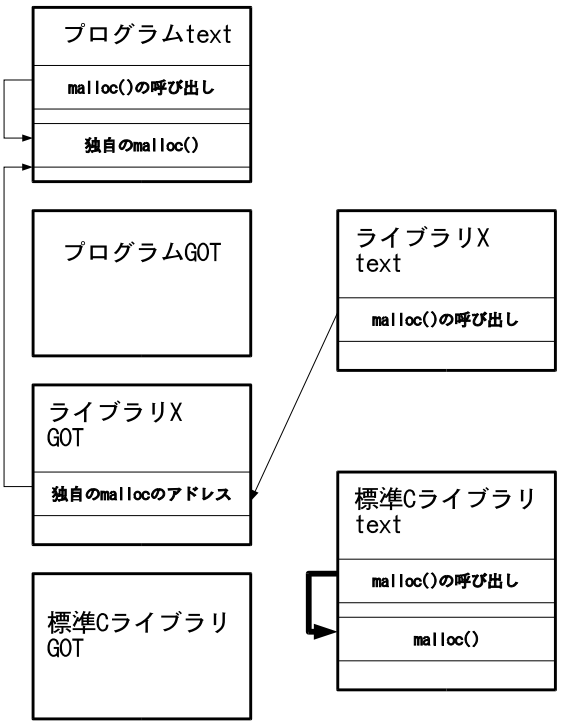

図 6: 関数の置き換え

図 6 に示すように、プログラムとライブラリ X の malloc の呼び出しは、プログラム中の独自の malloc を呼ぶ出すが、標準 C ライブラリ中の呼び出しは、標準の malloc のままである。

この問題を解決するために、本方式では、ライブラリ作成時のオプションによってすべての外部リンクの関数の呼び出しを GOT 経由にすることができる。このオプションを指定すると、同一モジュール内の関数であっても外部リンクの関数であれば、GOT 経由の呼び出しとなる。この結果、例のプログラムでは、標準 C ライブラリ内の malloc の呼び出しも GOT 経由でプログラム・モジュールの独自の malloc を呼び出す。ただし、このオプションを指定すると、実行オブジェクト・ファイル中のシンボルの増加、モジュール内の呼び出しのオーバヘッドの増加となる。

5.8 既存ライブラリの利用

本方式では、プログラム、ライブラリともに、本方式用にコンパイルするのが原則である。しかし、これでは、既に手元にライブラリがあり、利用したいがソースは持っていない場合に問題となる。本方式では、制限を設けてこのようなライブラリをプログラムに静的にリンクする事を許す。

このようなプログラムの data セグメントを text セグメントに連続してメモリ上に配置する事により実現している。このように配置すれば、プログラムにリンクされた既存のコードから見ると、data セグメントが既存のメモリ・イメージと同様に見える事を利用している。

既存のライブラリを静的にリンクしたプログラムは、共有ライブラリを利用する事はできるが、プログラムの text セグメントを共有する事はできない。また、既存のライブラリは、共有ライブラリのデータをアクセスしてはいけない。なお、共有ライブラリに既存のライブラリをリンクする事はできない。

6 他の方式

6.1 Cadenux 社の方式

Cadenux 社の方式の情報は、彼らのホームページ [3] から得られる。

Cadenux 社の方式は、最初は実行ファイル形式として ELF を採用していたが、後に bFLT を拡張した形式に変更した。この形式を xFLT[1] と呼び、方式を XFLAT と呼んでいる。

我々の方式と大きく異なるのは、GOT レジスタに相当する SB(Static Base) レジスタの設定を呼び出し側のモジュールで行なっている事である。そのため、ライブラリ番号は必要ないが、すでに指摘したように関数ポインタを他のモジュールに渡す場合に問題が生じる。この問題の解決法として、彼らはそのためのマクロと関数を提供している。これらを使って、ソースの関数ポインタを扱う部分の変更を行なう必要がある。また、Linux のシグナルの処理についても、関数ポインタを扱うために、カーネルの変更が必要である。

[1] 我々の方式でも実行ファイル形式を xFLT と呼んでいる。偶然名前が同じになってしまったが、全く別物である。

また、text セグメントの共有では、共有 mmap に頼っており、これを実装していないファイル・システムでは、共有することはできない。

6.2 SnapGear 社の方式

SnapGear 社の方式の情報は、彼らのホームページに存在する Technical Bulletin[5] から得られる。また、SnapGear 社は、共有ライブラリ機構を uClinux のコミュニティに提供しており、uClinux のホームページ [1] からダウンロードできる。

SnapGear 社の方式の特徴は、リンクを静的に行なう事である。実行時には、ロードとリロケーションのみ行なう。実行ファイル形式として、bFLT を採用しておりシンボルを一切含まない。

共有ライブラリには、番号が与えられる。静的リンク時に、参照部分には、仮の参照先アドレスが設定される。どのライブラリへの参照かを示すために、仮のアドレスの上位 8 ビットにライブラリ番号が入る。下位 24 ビットは、参照先の先頭からのオフセットである。ローダは、このライブラリ番号を見て、実際にライブラリがロードされたアドレスをオフセットに加えれば、実際のアドレスが得られる。ライブラリの名前は、ライブラリ番号から作成されており、ライブラリのロードもライブラリ番号がわかれば、可能である。

この方式では、実行オブジェクト・ファイルが小さくなり、ロード時の負荷も小さいという利点があるが、以下の欠点を持つ。

1. ライブラリを変更すると、それを利用しているプログラムすべての再リンクが必要となる。
2. 一旦リンクするとライブラリ関数の置き換えは、一切できない。
3. モジュールのアドレス空間が 16M しかない。
4. ライブラリをシステム内で最大 255 個しか利用できない。

GOT レジスタに相当するレジスタへの値の設定方式は、我々のものと非常に似ている。しかし、GOT アドレス・テーブルは、プロセスにひとつではなく、各モジュール毎にひとつ存在する。そのため、GOT アドレス・テーブルのアドレスを求めるのに、メモリをアクセスする必要がない。従って、メモリ・アクセスが 1 回減るが、消費メモリが増加するという欠点がある。

text セグメントの共有では、やはり、共有 mmap に頼っており、これを実装していないファイル・システムでは、共有することはできない。

7 おわりに

本論文では、我々が開発した uClinux 向けの共有ライブラリ機構について、その利点、方式について説明し、他の方式との比較を行なった。我々は、既に実際のプロセッサに実装し、有効性を確認した。XIP については、本論文執筆時には、実験的に実装し動作を確認したが、今後、本格的に実装していきたい。

最後になったが、情報提供や、テストなど開発に協力して頂いた方々に感謝したい。

参考文献

[1] “μＣｌｉｎｕｘホームページ”, http://www.uclinux.org/

[2] “Cadenux 社ホームページ”, http://www.cadenaux.com/

[3] “Cadenux XFLAT Shared Libraries”, http://www.cadenaux.com/xflat/

[4] “SnapGear 社ホームページ”, http://www.snapgear.com/

[5] SnapGear Technical Bulletin #9, “uClinux - Shared Libraries”, http://www.snapgear.com/tb20020409.html

[6] John R. Levine 著, 榊原一矢 監訳, ポジティブエッジ訳, “Linkers & Loaders”, オーム社開発局