2004年度 卒業論文

# ARM向け組み込みOSにおける実行性能改善に関する研究

提出日　平成17年 2月 2日
指導　中島 達夫 教授

早稲田大学 理工学部 情報学科

学籍番号　1g01p105-1
氏名　村田 雄

# 目 次

**第 1 章　序論** **1**
1.1 背景 . . . . . 1
1.2 目的 . . . . . 1
1.3 論文の構成 . . . . . 2

**第 2 章　CPU アーキテクチャ** **3**
2.1 ARM . . . . . 3
2.1.1 ARM とは . . . . . 3
2.1.2 レジスタ . . . . . 3
2.1.3 Current Program Status Register . . . . . 4
2.1.4 パイプライン . . . . . 7
2.1.5 Memory Management Unit . . . . . 8
2.1.6 例外処理 . . . . . 12
2.2 MIPS . . . . . 15
2.2.1 MIPS とは . . . . . 15
2.2.2 CPU レジスタ . . . . . 15
2.2.3 System Control Coprocessor (CP0) . . . . . 16
2.2.4 Status Register . . . . . 16
2.2.5 Cause Register . . . . . 18
2.2.6 メモリ管理 . . . . . 19
2.2.7 例外処理 . . . . . 20

**第 3 章　開発環境** **25**
3.1 OS/161 . . . . . 25
3.1.1 OS/161 . . . . . 25
3.1.2 System/161 . . . . . 26
3.2 CerfCube . . . . . 26
3.2.1 CerfCube . . . . . 26
3.2.2 I-Boot . . . . . 28
3.3 クロスコンパイル環境 . . . . . 28

**第 4 章　設計および実装** **30**
4.1 概要 . . . . . 30
4.2 ブートプロセス . . . . . 30

4.3 メモリ管理 . . . . . 33
4.4 例外処理 . . . . . 34
4.5 プロセス管理 . . . . . 35
4.6 システムコール . . . . . 35
4.7 デバイスドライバ . . . . . 35
4.7.1 タイマ . . . . . 35
4.7.2 シリアル . . . . . 36
4.8 ファイルシステム . . . . . 36

**第 5 章 評価** **38**
5.1 測定対象 . . . . . 38
5.2 測定方法 . . . . . 38
5.3 測定結果 1 . . . . . 39
5.4 高速化手法 . . . . . 40
5.4.1 Copy-Only-Data . . . . . 40
5.4.2 Copy-on-Write . . . . . 41
5.5 測定結果 2 . . . . . 44

**第 6 章 考察** **47**

**第 7 章 結論** **49**

# 図 目 次


2.1 user モードで使用できるレジスタ . . . . . 3
2.2 Program status register . . . . . 4
2.3 ARM の全レジスタ . . . . . 6
2.4 3 ステージパイプライン実行例 . . . . . 7
2.5 L1 page table entries . . . . . 8
2.6 TTB . . . . . 9
2.7 L2 page table entries . . . . . 9
2.8 L1 page table walk . . . . . 10
2.9 L2 page table walk . . . . . 11
2.10 IRQ 発生時の動作 . . . . . 13
2.11 MIPS のレジスタ . . . . . 15
2.12 MIPS CP0 レジスタレイアウト . . . . . 16
2.13 MIPS Status レジスタ . . . . . 17
2.14 MIPS Cause レジスタ . . . . . 18
2.15 MIPS 仮想メモリ . . . . . 20
2.16 例外発生時における Status レジスタ . . . . . 22
2.17 例外からの復帰時における Status レジスタ . . . . . 22

4.1 ARM-OS/161 メモリマップ . . . . . 32
4.2 プロセス 1 のメモリ空間 . . . . . 34
4.3 プロセス 2 のメモリ空間 . . . . . 34
4.4 OS/161 と ARM-OS/161 のファイルシステム . . . . . 37

5.1 プログラムの通常の実行時間 . . . . . 40
5.2 通常のロード . . . . . 40
5.3 Copy-Only-Data . . . . . 41
5.4 Copy-on-Write . . . . . 42
5.5 Copy-Only-Data の実行時間 . . . . . 44
5.6 Copy-on-Write (0 page copy) の実行時間 . . . . . 45
5.7 Copy-on-Write (32 page copy) の実行時間 . . . . . 46

6.1 ページコピー発生回数と実行時間の関係 . . . . . 47

# 表 目 次


2.1 プロセッサモード . . . . . 5
2.2 コンディションフラグ . . . . . 7
2.3 MMU のサポートするページテーブル . . . . . 8
2.4 各例外の優先度，および発生時のモード，ビットの対応 . . . . . 13
2.5 ベクタテーブル . . . . . 14
2.6 MIPS CP0 のレジスタ . . . . . 17
2.7 MIPS における例外の種類 . . . . . 21
2.8 例外ベクタのベースアドレス . . . . . 23
2.9 例外ベクタのオフセットアドレス . . . . . 23
2.10 Cause レジスタと Status レジスタへの割り当て . . . . . 24

3.1 CerfCube のハードウェア仕様 . . . . . 27
3.2 CerfCube のメモリマップ . . . . . 27

5.1 実行プログラムのサイズ . . . . . 38
5.2 プログラムの通常の実行時間の詳細 . . . . . 39
5.3 Copy-Only-Data の実行時間の詳細 . . . . . 44
5.4 Copy-on-Write (0 page copy) の実行時間の詳細 . . . . . 45
5.5 Copy-on-Write (32 page copy) の実行時間の詳細 . . . . . 46

6.1 ページコピー発生回数と実行時間の関係の詳細 . . . . . 47

## 概 要

近年，OS は様々な要求を取り込むために機能が増加し，その構造が大規模化，複雑化している．この様な状況の中，膨大なソースコードをもつ Linux のような OS は理解が困難であり，OS の内部構造を学ぶ上では不適切である．そこで我々は，小規模な OS である OS/161 を ARM プロセッサ向けに移植を行った．これにより，OS を理解する上で OS の移植は非常に効率の良い方法であることが分かった．また，組み込み機器では応答性やリアルタイム性といった要素は重要であるため，その性能を向上させる手法として Copy-on-Write の実装および評価を行った．この結果，Copy-on-Write は性能改善に非常に有効であり，最悪なケースでもオーバヘッドは 1.39%と非常に小さいことが分かった．

**Abstract**

In recent years, the structure of operating systems becomes larger and more complicated because operating systems have taken various demands. In such a situation, learning operating systems which have huge souce codes like Linux is very difficult. Then, we ported OS/161 that is a small-scale operationg system for the ARM processor. As a result, I found out that the porting operating system is a very efficient method to learn about operating system. In embedded systems, response times and real time is impoertant elements. So, I implemented a Copy-on-Write mechanism for improving the performance. As a result, Copy-on-Write is very effective to improve the performance, and I found that the overhead is only 1.39% in worst case.

# 第1章　序論

## 1.1　背景

近年，携帯電話や PDA のような小型携帯端末が多機能化している．また，デジタルテレビやハードディスクレコーダなど多機能な家電機器が登場し，組み込み機器がますます複雑化している．

特に，携帯電話は急速な普及に付随して様々な機能が追加された．当初はメールや Web 閲覧機能などであったが，近年では，デジタルカメラ機能を始めとしてテレビ機能や電子マネー機能が搭載されるまでになった．搭載されているプロセッサも，少し前のパーソナルコンピュータに匹敵するものである．プロセッサの処理能力増加によって，携帯電話で動作させることのできるソフトウェアの規模も大規模化している．また，カメラなどの外部デバイスと連携したソフトウェアも開発されている．

組み込み機器の高機能化に伴い，機器を構成するハードウェアや OS も複雑化している．このような複雑な機能に対応するために，一部の組み込み機器に Linux が搭載され始めてきた．しかし，一般的に Linux は従来のリアルタイム OS と比較すると，速度面や安定性において問題がある．特に，組み込み機器では応答性や耐障害性は重要であり，Linux ではこの様な問題点を完全に解決できてはいない．

Linux のソースコードは非常に膨大な量及ぶため，このような問題点を発見するのは困難である．また，問題解決のためには OS に関して詳細な知識を持っていなければならない．ところが，Linux のように膨大なソースコードからなる OS の理解は極めて困難である．

## 1.2　目的

OS の構造を理解するためには，メモリ管理や割り込み処理など OS の基本的な仕組みを理解する必要がある．また，OS のソースコードを読むだけでなく，OS のソースコードを書き換えるなど実際に触れる方がより詳細な理解が可能である．そこで，比較的規模の小さい OS の移植を行い，OS に関する知識を取得し理解を深めることを本研究の目的とする．

また，OS の実行においてボトルネックとなる箇所を特定し性能改善案の提案及び検証を行うことにより，組み込み機器向け OS に求められる性能を実現するための手法を探ることもまた本研究の目的である．

## 1.3 論文の構成

まず第 2 章で本研究の関連技術として ARM, MIPS のアーキテクチャについて述べる．次に第 3 章で，対象とする OS とハードウェアについて述べる．第 4 章では実装の流れと実装内容を述べ，第 5 章では問題点の指摘および具体的な性能改善案の実装とその評価について述べる．第 6 章では評価結果をもとに考察を述べる．また，第 7 章で関連研究を示し，第 8 章では将来課題について述べる．

# 第2章　CPUアーキテクチャ

## 2.1　ARM

### 2.1.1　ARMとは

ARMとは，イギリスのプロセッサメーカであるARM(Advanced RISC Machine)社，および同社の設計したCPUプロセッサのことをいう．以降，本論文においてARMはプロセッサを指すものとする．ARMは，1983年から1985年にイギリスのAcorn Computers社で開発されたRISC(Reduced Instruction Set Computer)方式のプロセッサである．消費電力が少ないため，携帯電話やハンドヘルドPCなど携帯機器の組み込み用プロセッサとして広く普及している．

### 2.1.2　レジスタ

ARMのuserモードにおいて，使用できるレジスタは図2.1の通りである．userモードとは，アプリケーションが実行されるときに，通常使用される保護モードのことである．実行モードに関しては，2.1.3にて詳細に述べる．レジスタは，16個のデータレジスタと，2個のステータスレジスタで構成される．

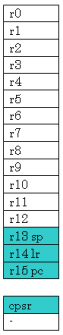


図 2.1: userモードで使用できるレジスタ

#### データレジスタ

データレジスタは r0 から r15 という名前で使用される．r0 から r12 までのレジスタは汎用レジスタであり，一方 r13, r14, r15 の 3 つのレジスタは特別な用途に用いられる．用途は以下の通り．

- r13: sp(stack pointer) とも呼ばれ，一般的にスタックの先頭を指すレジスタとして使用する
- r14: lr(link register) とも呼ばれ，関数呼び出し後の戻りアドレスを指すレジスタである
- r15: pc(program counter) とも呼ばれ，プロセッサによって読み出される次の命令のアドレスを指すレジスタである

#### ステータスレジスタ

ステータスレジスタには，cpsr(current program status register), spsr(saved program status register) の 2 種類がある．ステータスレジスタの詳細については，2.1.3 にて述べる．

### 2.1.3 Current Program Status Register

cpsr は，プロセッサの状態のモニタおよび制御を行う．cpsr の基本的なレイアウトを図 2.2 に示す．cpsr は，flags, status, extension, control の 4 つのフィールドに分けられる．

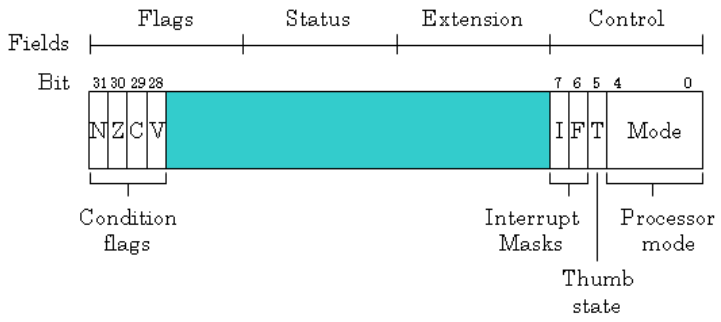


図 2.2: Program status register

### ARM のプロセッサモード

ARM のプロセッサモードには 7 種類あり，それぞれのモードは特権モードと非特権モードの 2 種類に分けられる．実行モードによって，アクティブなレジスタ群や，cpsr レジスタへのアクセス権が異なる．

- 特権モード
  abort, fast interrupt request, interrupt request, supervisor, system, undefined の 6 種類のモードから成る．cpsr レジスタに対する全ての読み込み・書き込みが許可されている．
- 非特権モード
  user モードのみから成る．cpsr レジスタの Control field に対して，読み込みのみ許可されている．Condition flag に対しては，読み込み・書き込み，共に許可されている．

表 2.1 に各モードに対応する，cpsr 内のプロセッサモードのビットパターンを示す．

| モード | 略称 | 特権 | モードビット |
|---|---|---|---|
| Abort | abt | yes | 10111 |
| Fast interrupt request | fiq | yes | 10001 |
| Interrupt request | irq | yes | 10010 |
| Supervisor | svc | yes | 10011 |
| System | sys | yes | 11111 |
| Undefined | und | yes | 11011 |
| User | usr | no | 10000 |

表 2.1: プロセッサモード

プロセッサモードは，cpsr レジスタの値を直接書き変えることで明示的に変更が可能である．また，例外や割り込み発生時は，ハードウェアによって自動的に切り替えられる．モードの切り替えが発生する例外や割り込みは，reset, interrupt request, fast interrupt request, software interrupt, data abort, prefetch abort, undefined instruction である．例外や割り込みによってモードが切り替わったとき，モードが切り替わる直前の cpsr が spsr(saved program status register) にコピーされる．以前のモードに復帰させる場合は，特殊な命令を使用して spsr の値を cpsr に復元する．

### バンクレジスタ

図 2.3 に ARM の全 37 のレジスタを示す．この中で影の付いた 20 個レジスタは，プロセッサが特定のモードでなければ使用することはできない．このようなレジスタは，バンクレジスタと呼ばれる．

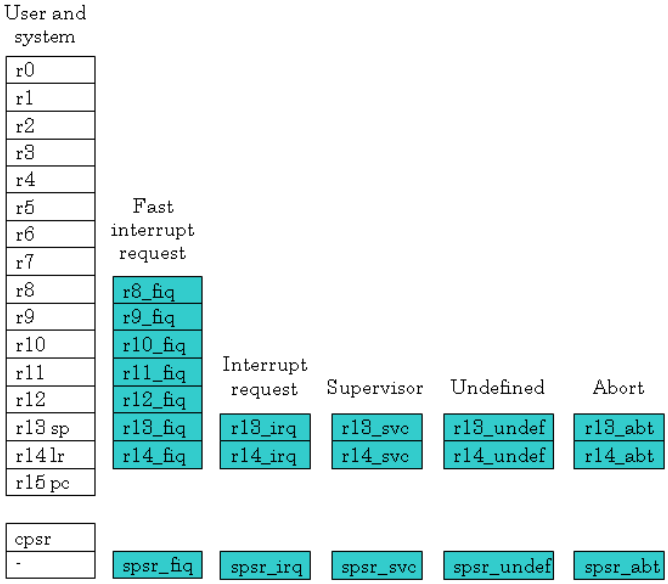


図 2.3: ARM の全レジスタ

user モードを除いた全てのプロセッサモードでは，cpsr の mode ビットを書き換えることで，プロセッサモードを変更することができる．また，system モードを除いた全てのプロセッサモードでは，user モードのレジスタに対応したバンクレジスタを持っている．プロセッサモードが変更されると，新しいプロセッサモードのバンクレジスタが，前のプロセッサモードのレジスタと切り替わる．

例えば，プロセッサが user モードから supervisor モードに切り替わったとき，r13, r14 レジスタにアクセスする命令を考える．r13, r14 はバンクレジスタであるから，r13_svc, r14_svc レジスタを指す．つまり，命令で参照されるレジスタは r13_svc, r14_svc であり，user モードのレジスタ r13_usr, r14_usr は全く影響を受けない．一方，バンクレジスタではない r0 から r12 のレジスタは，通常通りアクセスされる．

### 割り込みマスク

割り込みマスクは，プロセッサの割り込みを無効化するために使用される．cpsr には I ビットと F ビットの 2 つのマスクビットがあり (図 2.2 を参照)，I ビットに 1 をセットすると Interrupt request(IRQ) が，F ビットに 1 をセットすると Fast interrupt request(FIQ) がそれぞれマスクされる．

### コンディションフラグ

cpsr のコンディションフラグは，比較演算や ALU(Arithmetic Logic Unit) 処理の結果に応じて更新される．ほとんどの ARM の命令は実行されるか否かを，コンディ

ションフラグの値によって決めることができる．表 2.2 はコンディションフラグと，コンディションフラグがセットされる要因を示している．

| フラグ | フラグ名 | セットされる要因 |
|---|---|---|
| Q | Saturation | オーバーフロー　かつ/または　飽和状態発生時 |
| V | oVerflow | 符号付きオーバーフロー発生時 |
| C | Carry | 符号無しキャリー発生時 |
| Z | Zero | 演算結果が 0 |
| N | Negative | 演算結果のビット 31 が 1 |

表 2.2: コンディションフラグ

### 2.1.4　パイプライン

パイプラインを使用すると，他の命令がデコードおよび実行中に次の命令をフェッチできるため，スループットを向上させることができる．ここでは，次の 3 ステージから成るパイプラインを例に説明する．

- フェッチ: メモリから命令をロードする
- デコード: 実行される命令を解釈する
- 実行: 命令を実行し，結果をレジスタに書き込む

次の図 2.4 はパイプラインとプログラムカウンタ (pc) の関係を示したものである．

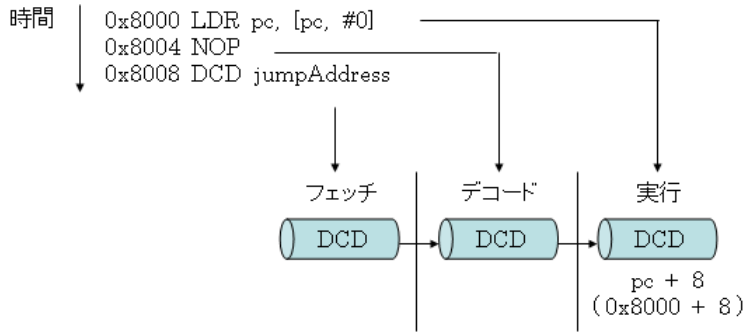


図 2.4: 3 ステージパイプライン実行例

3 ステージから成るパイプラインでは，pc が指すアドレスは実行ステージに位置する命令のアドレスの 8 バイト先を指す．つまり，pc は実行されている命令の 2 つ先の命令のアドレスを常に指している．また，分岐命令や pc を直接更新するような分岐では，パイプラインがフラッシュされる．

### 2.1.5 Memory Management Unit

#### ページテーブル

ARM の MMU は複数段のページテーブルをもった構成をしている．2 段のページテーブルがあり，1 段目を L1 ページテーブル，2 段目を L2 ページテーブルと呼ぶ．

L1 ページテーブルは master ページテーブルまたは section ページテーブルと呼ばれる．L1 ページテーブルは L2 ページテーブルの開始アドレスまたはセクションと呼ばれる 1MB のページへのアドレス変換を行う PTE(Page Table Entry) をもつ．L1 ページテーブルは 4GB のアドレス空間を 1MB のセクションに分割するため，4096 個の PTE をもつ．

| 名前 | タイプ | メモリ量 (KB) | ページサイズ (KB) | PTE の数 |
|---|---|---|---|---|
| Master/section | L1 | 16 | 1024 | 4096 |
| Fine | L2 | 4 | 1,4 または 64 | 1024 |
| Coarse | L2 | 1 | 4 または 64 | 256 |

表 2.3: MMU のサポートするページテーブル

L1 ページテーブルは以下の 4 種類のエントリを含む．

- セクションページ → 1MB のページ
- Fine L2 ページテーブル → 1024 個のエントリをもつページテーブル
- Coarse L2 ページテーブル → 256 個のエントリをもつページテーブル
- Fault → アボートを発生させる

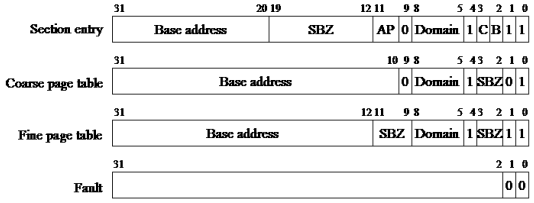


図 2.5: L1 page table entries

MMU は PTE の下位 2 ビットによってタイプを判別する (図 2.5 を参照)．

セクション PTE は 1MB のメモリのブロックのアドレスを含む．仮想アドレスの上位 12 ビットを PTE の上位 12 ビットと置き換え，物理アドレスを計算する．

Coarse PTE と Fine PTE は L2 ページテーブルへのポインタを含む．L2 ページテーブルはそれぞれ 1KB，4KB のアラインに乗っていなければならない．

Fault PTE はページフォールトを発生させる．ページフォールトは Prefetch Abort または Data Abort を発生させるが，それはどのようなメモリアクセスを試みたかに依存する．

L1 ページテーブルのメモリ内での位置は CP15 のレジスタ 2 に設定する．CP15 のレジスタ 2 は TTB(translation table base address) と呼ばれ，仮想メモリでの L1 ページテーブルのアドレスを指すレジスタをもつ．コプロセッサのレジスタ 2 のフォーマットを図 2.6 に示す.

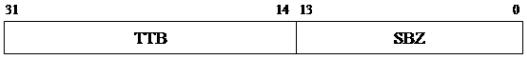


図 2.6: TTB

L2 ページテーブルは以下の 4 種類のエントリを含む.

- Large Page → 64KB のページ
- Small Page → 4KB のページ
- Tiny Page → 1KB のページ
- Fault Page → アボートを発生させる

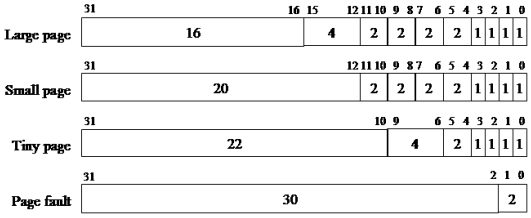


図 2.7: L2 page table entries

MMU は PTE の下位 2 ビットによってタイプを判別する．(図 2.7 を参照)

Large PTE は物理アドレスの 64KB のブロックの開始アドレスを含む．エントリにはアクセス権を設定するビットフィールドが 4 つあり，64KB を 4 つに分割した 16KB のサブページごとにアクセス権を設定することができる．

Small PTE は物理アドレスの 16KB のブロックの開始アドレスを含む．Large PTE 同様エントリにはアクセス権を設定するビットフィールドが 4 つあり，4KB を 4 つに分割した 1KB のサブページごとにアクセス権を設定することができる．

Tiny PTE は物理アドレスの 1KB のブロックの開始アドレスを含む．エントリにはアクセス権を設定するビットフィールドは 1 つしかない.

Fault PTE はページフォールトを発生させる．ページフォールトは Prefetch Abort または Data Abort を発生させるが，それはどのようなメモリアクセスを試みたかに依存する.

### Translation Lookaside Buffer

TLB(Translation Lookaside Buffer) はもっとも最近使われた PTE を保存するためのキャッシュである．ARM アーキテクチャには TLB を操作するコマンドは 2 種類しかない．一つは TLB のフラッシュ，もう一つは TLB によるアドレス変換のロックである．

メモリアクセスが発生すると MMU はその仮想アドレスに対応するエントリが TLB 内にキャッシュされているか調べる．もしエントリがあるなら，TLB は仮想アドレスを物理アドレスに変換する．エントリがない場合，すなわち TLB をミスヒットした場合，メインメモリを参照しページテーブルによるアドレス変換を行う．ページテーブルを走査し有効な PTE があった場合，それを TLB にキャッシュし，物理アドレスへの変換を行い，メモリアクセスを行う．

TLB ミスヒットが起こった場合のページテーブルによるアドレス変換について説明する．

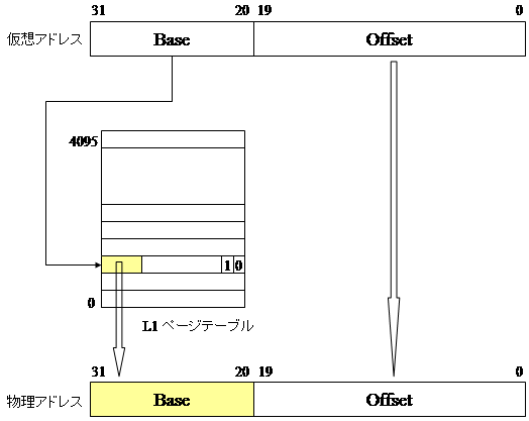


図 2.8: L1 page table walk

MMU が 1MB のセクションページを走査する場合 (図 2.8)，エントリは master L1 ページテーブルの中にあるので，1 段目の走査だけですむ．MMU は仮想アドレスの先頭の 12 ビットを用いて，L1 master ページテーブルの 4096 個のエントリの中の一つを選択する．エントリの下位 2 ビットが’10’ ならば，PTE は有効な 1MB のセクションページを指していることになる．PTE は TLB にキャッシュされ，PTE の上位 12 ビットと仮想アドレスの下位 20 ビットを合わせて物理アドレスを計算する．

1，4，16 または 64KB のページを走査する場合 (図 2.9) はアドレス変換のためにページテーブルを 2 段走査しなければならない．そのため仮想アドレスを 3 つに分割する．まず，仮想アドレスの先頭 12 ビットが L1 master ページテーブルの PTE を選択する．PTE の下位 2 ビットが’01’ ならば PTE は Coarse Page を指す L2 ページテーブルの先頭アドレスを含み，下位 2 ビットが’11’ ならば Fine Page を指す L”ページテーブルの先頭アドレスを含む．次にこのアドレスと仮想アドレスの 12-19 ビットを合わせて L2 ページテーブルの PTE を選択する．最後にこの PTE の先頭 20 ビットと

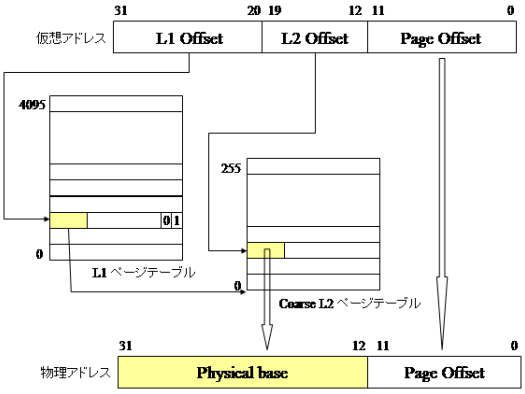


図 2.9: L2 page table walk

仮想アドレスの下位 12 ビットを合わせて物理アドレスを導き出す.

### 2.1.6 例外処理

例外が発生すると ARM プロセッサは特定のモードへ移行する．例外発生時の動作は，

1. cpsr[1]を例外モードの spsr[2]にコピー
2. pc[3]を例外モードの lr[4]にコピー
3. cpsr の I ビット[5]，F ビット[6]を設定，モードを設定
4. pc の値を特定のアドレスに設定

ARM プロセッサは Reset, Undefined Instruction, Software Interrupt(以下 SWI), Prefetch Abort, Data Abort, Interrupt Request(以下IRQ), Fast Interrupt Request(以下 FIQ) の 7 種類の例外を持つ．次にそれぞれの例外について述べる．

Reset はシステムの初期化を行う．Reset は例外の中で最も高い優先度をもつ．また Reset は各動作モードのスタックポインタの初期化も行う．発生時に cpsr の I ビットと F ビットは 1，モードは Reset モードに設定される．

Data Abort はメモリコントローラか MMU が命令が不正なメモリ領域へのアクセスを検知するか，User モードで実行中のプログラムが許可されていない領域へアクセスした際に発生する．これを利用してマッピングされていない領域へのアクセスを検知して，動的にマッピングを変更することで仮想メモリを実現することができる．発生時に I ビットは 1，モードは Abort モードに設定される．

IRQ は外部要因によって割り込みが発生した際に発生する．IRQ は 2 番目に優先度の低い割り込みである．IRQ は FIQ か Data Abort が発生していない場合に発生する．発生時にシステムは発生原因 (cause) レジスタを参照して割り込みの原因を特定し，適切な処理を行う．cpsr の I ビットが 1 のとき IRQ は発生しない．発生時に I ビットは 1，モードは IRQ モードに設定される．

FIQ は外部要因によって割り込みが発生した際に発生する．FIQ は優先度の高い割り込みである．FIQ は Data Abort の発生していない場合に発生する．IRQ と同様にシステムは発生原因を特定し，適切な処理を行う．cpsr の F フラグが 1 のとき FIQ は発生しない．発生時に I ビットと F ビットは 1，モードは FIQ モードに設定される．

Prefetch Abort は命令のフェッチの際に不正な領域にアクセスすることによって発生する．この例外は命令がパイプラインの実行ステージに到達し，なおかつ他の優先度の高い例外が発生していない場合に発生する．発生時に I ビットは 1，モードは Abort モードに設定される．

Sowtware Interrupt は SWI 命令が実行され，なおかつ他の優先度の高い例外が発生していない場合に発生する．発生時に I ビットは 1，動作モードは Supervisor モードに設定される．

[1] current program status register
[2] saved program status register
[3] program counter
[4] link register
[5] cpsr の第 7 ビット
[6] cpsr の第 6 ビット

| 例外 | 優先度 | モード | I ビット | F ビット |
|---|---|---|---|---|
| Reset | 1 | SVC | 1 | 1 |
| Data Abort | 2 | ABT | 1 | 0 |
| FIQ | 3 | FIQ | 1 | 1 |
| IRQ | 4 | IRQ | 1 | 0 |
| Prefetch Abort | 5 | ABT | 1 | 0 |
| SWI | 6 | SVC | 1 | 0 |
| Undefined Instruction | 6 | UND | 1 | 0 |

表 2.4: 各例外の優先度，および発生時のモード，ビットの対応

Undefined Instruction は ARM 命令セット，Thumb 命令セット[7]に存在しない命令がパイプラインの実行ステージに到達し，なおかつ他の優先度の高い例外が発生していない場合に発生する．ARM は命令がコプロセッサで処理可能かどうかを問い合わせ，どのコプロセッサでも処理できない場合は Undefined Instruction が発生する．発生時に I ビットは 1，モードは Undefined モードに設定される．

例として User モードで動作中に IRQ 例外が発生した場合の動作を図 2.10 に示す．

図 2.10: IRQ 発生時の動作

各例外の優先度，発生時に移行するモード，フラグの関係を表 2.4 に示す．

### ベクタテーブル

例外が発生するとその発生原因によって ARM プロセッサは特定のアドレスへブランチする．このアドレスの範囲をベクタテーブルという．通常，ベクタテーブルは 0x00000000 - 0x0000001c の範囲を指すが，プロセッサによっては MMU の設定をすることでベクタテーブルをより高位のアドレス 0xffff0000 - 0xffff001c に移すことができる．各例外とアドレスの対応を表 2.5 に示す．

通常，ベクタテーブルのエントリには次に示すようなブランチ命令が格納されている．

**b <address>** b 命令はアドレス <address> にブランチする．ただしブランチ先は b 命令の前後 32MB に制限される．

**ldr pc, [pc, #offset ]** ldr 命令はメモリから pc へアドレスをロードする．ただし offset の値は ±0 - 4KB に制限される．つまりロードするアドレスは ldr 命令の前後 4KB の範囲に配置されていなければならない．

[7] 16bit に圧縮された命令セット

| 例外 | アドレス | アドレス (高位) |
|---|---|---|
| Reset | 0x00000000 | 0xffff0000 |
| Undefined Instruction | 0x00000004 | 0xffff0004 |
| SWI | 0x00000008 | 0xffff0008 |
| Prefetch Abort | 0x0000000c | 0xffff000c |
| Data Abort | 0x00000010 | 0xffff0010 |
| 未使用 | 0x00000014 | 0xffff0014 |
| IRQ | 0x00000018 | 0xffff0018 |
| FIQ | 0x0000001c | 0xffff001c |

表 2.5: ベクタテーブル

**mov pc, #immediate** mov 命令は即値 immediate を pc へコピーする．immediate の値は 8bit の値を偶数回右ローテートした値に制限される．

また，ベクタテーブルにはブランチ命令以外の命令を格納することもできる．例えば次に示すコードでは FIQ のベクタテーブルエントリに FIQ 例外ハンドラを直接設置している．

```
0x00000000: ldr     pc, [pc, #reset]
0x00000004: ldr     pc, [pc, #undef]
0x00000008: ldr     pc, [pc, #swi]
0x0000000c: ldr     pc, [pc, #pabt]
0x00000010: ldr     pc, [pc, #dabt]
0x00000014: ldr     pc, [pc, #none]
0x00000018: ldr     pc, [pc, #irq]
0x0000001c: sub     lr, lr, #4
            stmdb   sp!, {r0-r3}
            bl      fiq_isr
            ldmdb   sp!, {r0-r3}
            movs    pc, lr
```

## 2.2 MIPS

### 2.2.1 MIPS とは

MIPS は，Microprocessor without Interlocked Pipline Stages の略称で，1981 年から 1983 年の間にスタンフォード大学で開発された RISC(Reduced Instruction Set Computer) 方式のプロセッサである．また，MIPS Computer Systems 社は，スタンフォード大学の MIPS を拡張したプロセッサ R2000/R3000 を開発した．R2000/R3000 は，RISC の特徴であるシンプル性を目指して設計されている．以降，本論文では特に明記しない限り，プロセッサは MIPS Computer Systems 社の R2000/R3000 を指すものとする．

### 2.2.2 CPU レジスタ

CPU レジスタは，32 個の汎用レジスタ，プログラムカウンタ (PC)，整数乗算と除算の結果を保存する 2 個のレジスタで構成される．これらのレジスタは全て 32 ビットである．CPU レジスタを図 2.11 に示す．

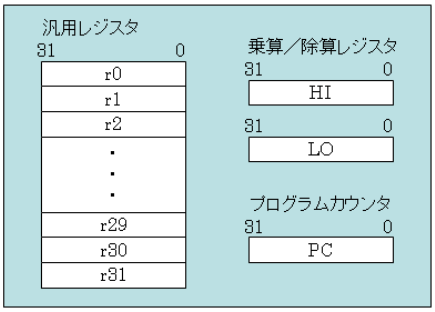


図 2.11: MIPS のレジスタ

#### 汎用レジスタ

2 個の汎用レジスタには，次のような役割がある．

- r0: 定数 0 を保持している．結果を保存する必要がない命令のターゲットレジスタとして使用されるほか，0 の値を必要とする場合のソースレジスタとしても使用される．
- r31: JAL, BLTZAL, BLTZALL, BGEZAL, BEGZALL といった Jump and Link 命令を使用したとき，戻りアドレスを保持するリンクレジスタとして暗黙的に使用される．

その他のレジスタは，通常の汎用レジスタとして利用できる．

#### 特殊な用途に使用されるレジスタ

CPU レジスタには，特定の命令によって暗黙的に使用もしくは更新される 3 個の特殊なレジスタがある．3 個のレジスタは，次の通り．

- PC: プログラムカウンタ
- HI: 乗算／除算結果の上位ワードを記録
- LO: 乗算／除算結果の下位ワードを記録

整数乗算の場合，64 ビットの乗算結果がレジスタ HI, LO に保存される．整数除算の場合，商がレジスタ LO に，余りがレジスタ HI にそれぞれ保存される．

### 2.2.3 System Control Coprocessor (CP0)

CP0 は CPU チップに組み込まれたコプロセッサで，System Control Coprocessor とも呼ばる．CP0 はオペレーティングシステムをサポートするのに必要な機能を提供している．例えば，メモリ管理や例外処理，スケジューリング，クリティカルリソースのコントロールなどである．また，kernel モードや user モードなどの実行モードの切り替えも管理する．メモリ管理については 2.2.6，例外処理については 2.2.7 にて説明する．

利用可能なレジスタを図 2.12 に示す．図に示したように，仮想アドレスを物理アドレスに変換する仮想メモリシステムは on-chip TLB に実装されている (MIPS プロセッサ R2000/R3000/R4000)．また，各レジスタの機能を表 2.6 に示す．Status レジスタは 2.2.4 で，Cause レジスタは 2.2.5 でそれぞれ述べる．

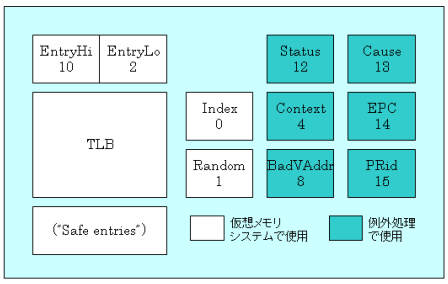


図 2.12: MIPS CP0 レジスタレイアウト

### 2.2.4 Status Register

Status レジスタは，実行モードや割り込み許可に関する情報を含んでいる．Status レジスタは，図 2.13 に示すように，各フィールドに分割される．

各フィールドの詳細は次の通りである．

| レジスタ番号 | レジスタ名 | 役割 |
|---|---|---|
| 0 | Index | TLB エントリのインデックス |
| 1 | Random | ランダムに生成される TLB インデックス |
| 2 | EntryLo | TLB エントリの下位半分 |
| 4 | Context | ページテーブルエントリへのポインタ |
| 8 | BadVAddr | アドレスに関する例外が発生したアドレス |
| 10 | EntryHi | TLB エントリの上位半分 |
| 12 | SR | Status レジスタ |
| 13 | Cause | 例外が発生したことを示す |
| 14 | EPC | 例外発生時のプログラムカウンタ |
| 15 | PRId | プロセッサのメーカや改訂 |

表 2.6: MIPS CP0 のレジスタ

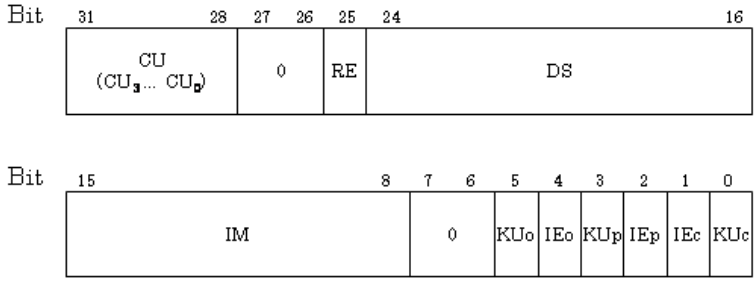


図 2.13: MIPS Status レジスタ

- IE: IEo, IEp, IEc は，Old/Previous/Current の割り込み許可状態を示す，3 段のスタックを構成している．
- KU: KUo, KUp, KUc は，Old/Previous/Current の実行モードを示す，3 段のスタックを構成している．スタックに値が push されるのは例外が発生したとき，スタックの値が pop されるのは Restore From Exception(RFE) 命令が発行されたときである．
- IM: Interrupt Mask フィールドは，8 つ割り込みをコントロールする 8 ビットのフィールドである．割り込みが許可され，対応する Status レジスタの Interrupt Mask ビットと Cause レジスタの Interrupt Pending フィールドが，共にセットされているとき，割り込みが発生する．
- DS: Diagnostic フィールドは，自己診断，キャッシュや仮想メモリのチェックに使用される．
- RE: Reverse Endian ビットは，ユーザモードで使用されるエンディアンを反転

させるために使用される．

- CU: Coprocessor Usability フィールドは，4つのコプロセッサ (CP0, CP1, CP2, CP3) のアクセスをコントロールする．ただし，CP0 は kernel モードで実行中の場合は，$CU_0$ の状態に関わらず常に利用可能である．

#### 実行モード

MIPS には，kernel モードと user モードの２つの実行モードがある．ユーザモードで実行されている場合，プログラムは CPU と FPU，仮想アドレス空間にアクセスすることができる．一方，プログラムがカーネルモードで実行されている場合，仮想メモリのマッピングの変更やシステム環境のコントロール，プロセス間のコンテキストスイッチなど，プロセッサの全ての機能を使用することができる．カーネルモードに切り替わるのは，電源投入時，割り込みや例外やエラーが発生したときである．プロセッサによっては，上記の 2 つの実行モードに加えて，Supervisor モードや debug モードもサポートされている．

### 2.2.5 Cause Register

Cause レジスタは，最後に発生した例外を記録する．5 ビットの例外コード (ExcCode) は，発生した例外の種類を示し，残りのフィールドは例外に関する情報を含んでいる．Cause レジスタは，図 2.14 に示すように，各フィールドに分割される．

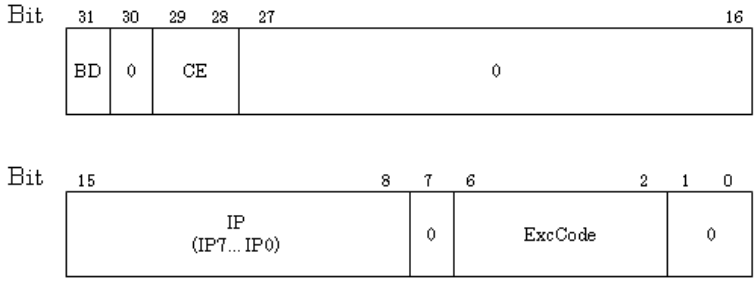


図 2.14: MIPS Cause レジスタ

各フィールドの詳細は次の通りである．

- ExcCode :Exception Code フィールドは，発生した例外の例外コードを示す．
- IP :Interrupt Pending フィールドは，external，internal，コプロセッサ，ソフトウェア割り込みが発生したことを示す．IP フィールドのうち，IP8 ビットおよび IP9 ビットは，ソフトウェア割り込みをコントロールするビットで，ソフトウェア割り込みのセットやリセットを行う．IP10 ビットから IP15 ビットまではハードウェア割り込みの発生を示す

- CE :Coprocessor Error フィールドは，Coprocessor Unusable exception が発生したとき，参照されているコプロセッサの番号を示している．
- BD :Branch Delay ビットは，最後に発生した例外が分岐の delay slot で発生したのかどうかを示している．

### 2.2.6 メモリ管理

#### User モードの仮想アドレス

user モードでは，図 2.15 の左図に示すように，アドレス 0x00000000 から始まる 2G バイトのアドレス空間 (kuseg) が利用可能である．それぞれの仮想アドレスには，ASID(Address Space Identifier) と呼ばれる 6 ビットの値が付加され，各ユーザプロセスに対して固有のアドレス空間を形成している．各プロセスに対して ASID を割り当てることで，コンテキストスイッチの際も TLB の状態を維持することができる．kuseg に対する全ての参照は TLB を通して行われ，キャッシュを使用するかどうかは TLB エントリの設定によって決定される．有効な仮想アドレスの最上位ビットは全て 0 でクリアされる．ユーザモードでは，最上位ビットがセットされたアドレスへの参照を行うと，Address Error 例外が発生する．

#### Kernel モードの仮想アドレス

kernel モードでは，図 2.15 の右図に示すように，4 つの異なるアドレス空間が利用可能である．3 つ目のアドレス空間はカーネルのために割り当てられていて，それぞれ仮想アドレスの上位ビットで識別される．4 つ目は kuseg で，user モード用のアドレス空間である．

- kseg0: 仮想アドレスの上位 3 ビットが 100 の場合，選択されるアドレス空間は kseg0 という名前の 512MB のカーネル空間である．kseg0 への参照は TLB を経由してはマップされるのではない．つまり，選択される物理アドレスは，仮想アドレスから 0x80000000 を引いた値となる．このアドレスへの参照は，常にキャッシュが利用可能である．
- kseg1: 仮想アドレスの上位 3 ビットが 101 の場合，選択されるアドレス空間は kseg1 という名前の 512MB のカーネル空間である．kseg1 への参照は TLB を経由してマップされるのではない．つまり，選択される物理アドレスは，仮想アドレスから 0xa0000000 を引いた値となる．このアドレスへの参照は，常にキャッシュを利用することできず，物理メモリが直接参照される．
- kseg2: 仮想アドレスの上位 2 ビットが 11 の場合，選択されるアドレス空間は kseg2 という名前の 1GB のカーネル空間である．仮想アドレスには ASID が付加され，固有のアドレス空間を形成している．

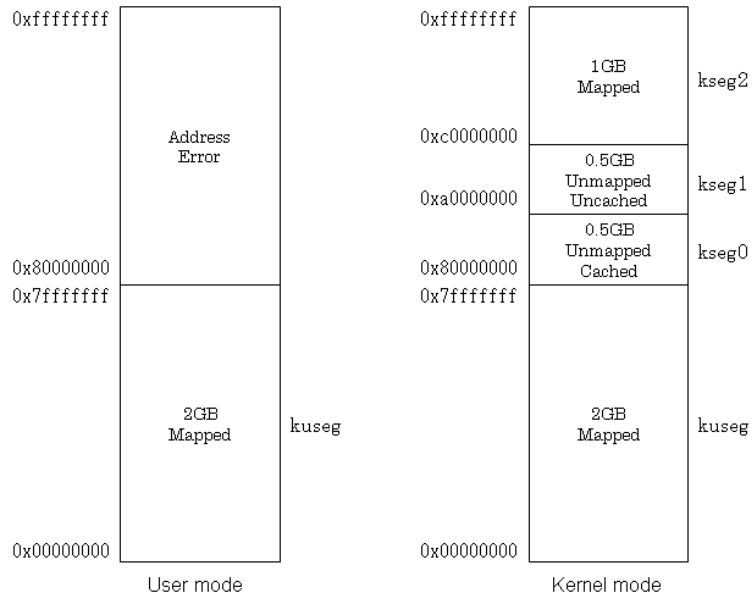


図 2.15: MIPS 仮想メモリ

### 2.2.7 例外処理

次に MIPS における例外処理について説明する．

CPU が例外を検出したとき，それまで実行していた命令のシーケンスは延期され，プロセッサは User モードから抜けて強制的にカーネルモードに移行する．カーネルモードでは例外的なイベントや非同期なイベントを処理することができる．

CP0 レジスタには例外処理に関連した情報が含まれており，例外が発生したときにはオペレーティングシステムは，例外の原因や CPU の状態を CP0 レジスタを調査することで取得する．

MIPS における例外の種類を表 2.7 に示す．MIPS ではハードウェア割り込みは表 2.7 における Interrupt として，システムコールは System Call として処理される．

例外を処理するために，プロセッサは割り込みを禁止して特定アドレスにあるハンドラを強制的に実行する．中断していた処理を再開させるために，オペレーティングモードの PC(Program Counter) や割り込みの許可は例外処理の終了後に復元される．

CPU は例外が発生した直後に，プログラムの実行再開アドレスを EPC(Exception Program Counter) にロードする．EPC 内の実行再開アドレスは例外を引き起こした命令のアドレスか，もしくは，もし命令が branch ディレイスロットで実行されていた場合は，ディレイスロット直前の branch 命令のアドレスである．例外処理の終了後は，EPC にロードされているアドレスから処理を再開し，例外発生前の実行を引き続き行う．

また，実行モードを保存し，割り込み許可状態に復帰させる際には，KU(Kernel/User mode) ビットおよび IE (Interrupt Enable) ビットを保存しておくための，Current/Previous/Old

| 例外名 | 例外発生原因コード | 説明 |
|---|---|---|
| Reset | - | 現在の処理を中断して reset ベクタから処理を再開させる |
| TLB Refill | TLBL/TLBS | 参照アドレスが TLB エントリのいづれにも一致しない |
| TLB Invalid | TLBL/TLBS | 参照する仮想アドレスが無効な TLB エントリと一致しない |
| TLB Modified | Mod | D ビットがセットされていない仮想アドレスへのアクセス |
| Bus Error | IBE/DBE | バスインタフェースから外部割り込みシグナルが発生 |
| Address Error | AdEL/AdES | word アラインされていない word の load，fetch もしくは store 命令の実行 |
| Integer Overflow | Ov | オーバフローが発生する add もしくは sub 命令の実行 |
| Trap | Tr | トラップ処理 |
| System Call | Sys | SYSCALL 命令の実行 |
| Breakpoint | Bp | BREAK 命令の実行 |
| Coprocessor Unusable | CpU | coprocessor-unable ビットがセットされていない状態でのコプロセッサ命令の実行 |
| Interrupt | Int | 各種割り込み処理 |

表 2.7: MIPS における例外の種類

の 3 段階のスタックが使用される．KUc ビットおよび IEc ビットは実行モードが Kernel モードであるか User モードであるかを，および割り込みが許可状態かどうかを示している．

例外が発生したとき，KUp，IEp，KUc および IEc の値は，それぞれ KUo，IEo，KUp，IEp にそれぞれ保存される．KUc および IEc はクリアされ，プロセッサは割り込み禁止状態で，カーネルモードで実行を開始する．(図 2.16 を参照)

例外処理から戻ってくると，KUc，IEc，KUp および IEp ビットは KUp，IEp，KUo および IEo からそれぞれ復元される．(図 2.17 を参照)

このようにして割り込みが発生した際に実行コンテキストは保存される．

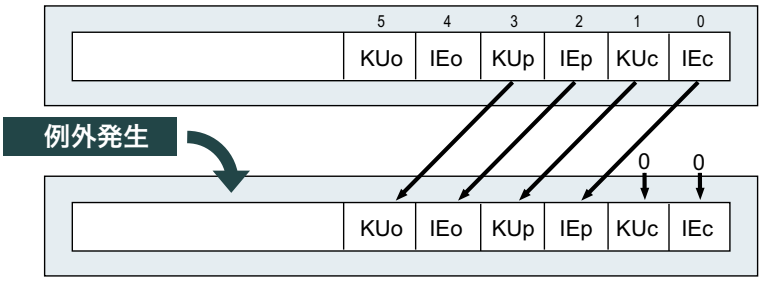


図 2.16: 例外発生時における Status レジスタ

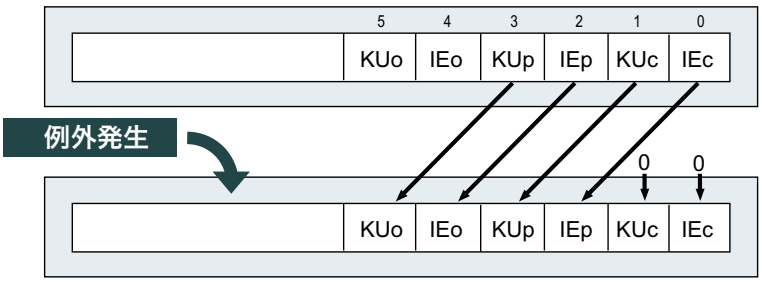


図 2.17: 例外からの復帰時における Status レジスタ

### ベクタテーブル

Reset は常に 0xbfc00000 にあるベクタを参照する．その他の例外のためのアドレスは，Status レジスタの BEV ビットによって決定される例外ベクタのベースアドレスとオフセットアドレスの組み合わせによって決まる．表 2.8 および表 2.9 に例外ベクタのベースアドレスとオフセットアドレスを示す．

| BEV ビット | R2000，R3000 および R6000 | R4000 |
|---|---|---|
| 0 | 0x80000000 | 0x80000000 |
| 1 | 0xbfc00100 | 0xbfc00200 |

表 2.8: 例外ベクタのベースアドレス

| 例外の種類 | R2000，R3000 および R6000 | R4000 |
|---|---|---|
| TLB refill，EXL = 0 | 0x000 | 0x000 |
| Cache Error | - | 0x100 |
| Others | 0x080 | 0x180 |

表 2.9: 例外ベクタのオフセットアドレス

#### 割り込み処理

MIPS は 8 つの割り込み要求をサポートしている．これらは次の 4 つの項目に分類することができる．

- ソフトウェア割り込み： 2 種類のソフトウェア割り込み要求があり，それぞれ Cause レジスタの IP0 と IP1 のビットに書き込むことにより発行される
- ハードウェア割り込み： 割り込み番号 0 から 5 まで割り当てられた最大 6 つまでのハードウェア割り込み要求があり，実装に依存したプロセッサへの外部要求により発行される
- タイマ割り込み： タイマ割り込みは Count レジスタと Compare レジスタが同じ値になったときに発生する
- Performance counter 割り込み： performance counter 割り込みは，counter の最上位ビットが 1 で，performance counter control レジスタの IE ビットによって割り込みが許可状態になったときに発生する

現在の割り込み要求の種類は，Cause レジスタの IP フィールドを通して参照することができる．表 2.10 に割り込み要求における Cause レジスタのビットのマッピングを示す．

Cause レジスタの IP フィールドのそれぞれのビットは Status レジスタにおける IM フィールドのビットに対応している．割り込みが許可され，対応する Status レジスタの InterruptMask ビットと Cause レジスタの Interrupt Pending ビットが共にセットされているとき，割り込みが発生する．

- interrupt request ビットが Cause レジスタの IP フィールドのうちのいずれかであること

| | | Cause レジスタのビット | | Status レジスタのビット | |
|---|---|---|---|---|---|
| 割り込みの種類 | 割り込み番号 | 番号 | 名前 | 番号 | 名前 |
| ソフトウェア割り込み | 0 | 8 | IP0 | 8 | IM0 |
| | 1 | 9 | IP1 | 9 | IM1 |
| ハードウェア割り込み | 0 | 10 | IP2 | 10 | IM2 |
| | 1 | 11 | IP3 | 11 | IM3 |
| | 2 | 12 | IP4 | 12 | IM4 |
| | 3 | 13 | IP5 | 13 | IM5 |
| | 4 | 14 | IP6 | 14 | IM6 |
| ハードウェア割り込み, タイマ割り込み, performance counter 割り込み | 5 | 15 | IP7<br>0 | 15 | IM7 |

表 2.10: Cause レジスタと Status レジスタへの割り当て

- corresponding mask ビットが Status レジスタの IM フィールドのいずれかであること
- Status レジスタ内の IE ビットが 1 であること

# 第3章　開発環境

## 3.1　OS/161

### 3.1.1　OS/161

OS/161 は 1996 年にハーバード大学の System Research at Harvard グループによって開発された教育用オペレーティングシステムである．オペレーティングシステムは年々大規模化してきており，現在の Linux 2.4 を例に挙げるとソースコードは数百万ラインにも及ぶ．それに比べ，OS/161 はソースコードが 2 万ライン程度で，比較的小規模といえるオペレーティングシステムである．

OS/161 にはスタンドアロンカーネルとシンプルなユーザアプリケーション実行環境が含まれ，C 言語およびアセンブリ言語で記述されている．

OS/161 のアーキテクチャの特徴は，次のようなものがあげられる．

1. BSD のようなソースツリー構造とビルド環境
2. カーネルのマシン依存部と非依存部への分割構造
3. NetBSD に見られるようなデバイス拡張フレームワーク
4. VFS レイヤによる複数ファイルシステムの適用

また，OS/161 のコードは以下のようなものを提供している．

1. MIPS へのポーティング
2. System/161 のためのドライバセット
3. ホストシステムファイルへのアクセスのためのファイルシステムデバイス
4. 非常に小規模な仮想メモリシステム
5. シンプルなスケルトンファイルシステム
6. シンプルなラウンドロビンスケジューラ
7. カーネル内でのスレッドパッケージ
8. セマフォの実装

一方，OS/161 には実装されていないコードも多く，例えばシステムコールや，完全なファイルシステム，ユーザ空間のプロセスなども実装されていないので，その部分は独自に実装する必要がある．

この OS/161 は MIPS のハードウェアシミュレータである System/161 上で動作する．System/161 については次節で説明する．

### 3.1.2 System/161

System/161 は OS/161 と同様にハーバード大学の System Research at Harvard グループによって開発された MIPS アーキテクチャのハードウェアシミュレータである．対応している MIPS のバージョンは MIPS R2000 および MIPS R3000 であり，x86 アーキテクチャ上で動作させることができる．

System/161 は非常にシンプルに設計されている．System/161 を設計する上で最も重要視された目標は，その上で動作するオペレーティングシステムを複雑化させることなく必要な機能の提供を可能にすることである．そのため，System/161 は，ディスク，シリアルポート，タイマやランダムジェネレータなどをサポートするためのシミュレートされたデバイスが組み込まれた，シンプルなバスアーキテクチャを持っている．これらのデバイスは実際のデバイスと同様に，レジスタを通してアクセスされる．

また，System/161 はビッグエンディアンモードで動作する MIPS R2000 プロセッサシミュレーションも含んでいる．

コードは全て C 言語で記述されており，コード量は 9000 ライン程度である．複数のプロセッサタイプに対応できるように構成されている．

System/161 は OS/161 と共に動作させることを想定して設計されている．

## 3.2 CerfCube

### 3.2.1 CerfCube

CerfCube は Intrinsyc Software 社製の小型組み込みデバイスであり，同社製の小型に最適化されたインターネット端末用のヘッドレス組込み型デバイスである CerfBoard という基盤をベースとして設計されている．そのため，CerfCube も同様に小型に最適化されているのが特徴であり，その筐体は約 7cm の角立方体サイズである．CPU には，Intel 社製の Strong ARM SA-1110 を搭載しており，そのクロック周波数は 192MHz である．また，16MB のフラッシュメモリと 32MB の SDRAM が利用可能である．外部接続は Ethernet，シリアルポートおよび USB などを使用して行うことができる (CerfCube の詳細なハードウェア仕様については表 3.1 を，メモリマップについては表 3.2 を参照)

CerfCube には i-Linux 2.4 もしくは Windows CE OS 3.0 がプレインストールされている．本研究では i-Linux 2.4 がプレインストールされている機種を用いた．以下では，i-Linux がプレインストールされているバージョンの CerfCube について述べる．

| | |
|---|---|
| CPU | Intel StrongARM SA1110 192MHz |
| メインメモリ | 32MB SDRAM (32-bit データバス) |
| フラッシュメモリ | Intel Strata フラッシュメモリ 16MB(16-bit データバス) |
| シリアル | 3-RS232C シリアルポート (2 ライン) |
| 表示装置 | 1 LED |
| Ethernet | 10 Base-T |
| コンパクトフラッシュ | Type I，Type II CF カードインターフェース |
| USB | Type B ポート |
| 消費電力 | 5.0 VDC 400mA (コンパクトフラッシュデバイスなし) |
| 大きさ | 57mm × 69mm |

表 3.1: CerfCube のハードウェア仕様

| | 使用ブロック<br>(1 ブロック 128KB) | データ項目 | アドレス範囲 | 大きさ |
|---|---|---|---|---|
| Flash | 0 | Bootloader (I-Boot) | 0x00000000 | 128KB |
| | | | 0x0001FFFF | |
| | 1-2 | Bootloader Reservec | 0x00020000 | 256KB |
| | | | 0x0005FFFF | |
| | 3-10 | Linux Kernel | 0x00060000 | 1MB |
| | | | 0x0015FFFF | |
| | 11-128 | JFFS2 FileSystem | 0x00160000 | 14.6MB |
| | | | 0x00FFFFFF | |
| Unused | | | 0x01000000 | |
| | | | 0xBFFFFFFF | |
| RAM | | | 0xC0000000 | 32MB |
| | | | 0xC1FFFFFF | |

表 3.2: CerfCube のメモリマップ

CerfCube は開発者向けの製品ではあるが，WWW サーバ機能を備えているので Web サーバとして利用することも可能である．その他，ファイルサーバとしての機能も備えている．筐体にはディスプレイに接続するための端子がないため，操作や設定は通常 Ethernet 経由でパーソナルコンピュータなどのブラウザからアクセスすることによって行う．本研究では，minicom というプログラムを用いてシリアル経由で，CerfCube の操作や設定を行った．

電源プラグを差すと数秒後に，Intrinsync Software 社製のブートローダである I-Boot が自動的に i-Linux を起動させる．この I-Boot は Linux と Windows CE の起動に対応している．i-Linux が起動する前に ENTER キーを押すことで，I-Boot のコンソールモードを移行することもできる．今回，実装したオペレーティングシステムを起動するにあたり，この I-Boot を用いてカーネルのメモリ上へのロードなどを行っている．次に I-Boot でオペレーティングシステムを実行する手順を述べる．

### 3.2.2 I-Boot

I-Boot がオペレーティングシステムを起動する際には，まず Flash メモリ上のカーネルイメージのマジックナンバを検査して，正しい値であった場合にはカーネルを RAM 上に展開するアドレスを取得する．次に Flash メモリから RAM 上にカーネルイメージのコピーを行う．カーネルイメージのコピーが終了すると，ロードされたカーネルの先頭番地に制御を移し，カーネルを実行させることができる．

Flash メモリにカーネルイメージを書き込まなくても，TFTP を用いてホストマシンからカーネルイメージをダウンロードして起動することも可能である．この場合は，まずホストマシンから RAM 上にカーネルイメージのコピーを行う．RAM 上のカーネルイメージのマジックナンバを検査して，カーネルを RAM に展開するアドレスを取得する．ここで，取得したアドレスと実際のアドレスが異なった場合はカーネルの再配置を行う．

マジックナンバはカーネルイメージの先頭から 0x24 バイト目の位置に置かれ，その値は 0x016f2818 である．アドレスはカーネルイメージの先頭から 0x28 バイト目の位置に置かれ，その値は i-Linux 4.1 の bzImage の場合では 0xc0008000 となっている．本研究で実装したオペレーティングシステムにおいても，同様に 0xc0008000 を指定し，カーネルの実行開始アドレスとしている．

## 3.3 クロスコンパイル環境

クロスコンパイルとはターゲットマシンで実行可能なバイナリをホストマシンで作成することである．本研究では，開発マシンとして x86 アーキテクチャを使用した．そして，ターゲットマシンである CerfCube は ARM を搭載しているので，ARM 用のバイナリを生成するための環境構築を行った．クロスコンパイル環境を構築するには，コンパイラ，ライブラリ，ヘッダファイル，ユーティリティを用意する必要がある．CerfCube に付属している CD の中には cross-arm-toolchain というクロスコンパ

イル環境を構築するための rpm パッケージが入っているので，cross-arm-toolchain をインストールしてクロスコンパイル環境を構築した．

# 第4章　設計および実装

## 4.1　概要

本研究では，MIPS のシミュレータ上で動作する OS/161 を，ARM 向けに移植を行った．以降，本論文では ARM 上で動作する OS/161 を ARM-OS/161 と呼び，MIPS のシミュレータ上で動作する OS/161 は単に OS/161 と呼ぶものとする．

実装は，主にコードの書き換えが必要なマシン依存の部分を中心に行った．また，実装されていない部分に関しては，今回新たにコードを追加した．

実際に実装を行ったのは主に次の箇所である．

- ブートプロセス
- メモリ管理
- 例外処理
- プロセス管理
- システムコール
- デバイスドライバ
- ファイルシステム

本章では，各実装の具体的な内容について詳細に説明する．

## 4.2　ブートプロセス

CerfCube にカーネルのイメージを転送し，下記のブートプロセスを実行すると ARM-OS/161 が起動する．I-Boot に転送先のメモリアドレス 0xc0008000 を指定することで，カーネルイメージを定位置に配置することができる．同様に，I-Boot にカーネルのエントリポイント 0xc0008000 を指定することで，OS のブートプロセスが開始する．

ブートプロセスには，主に 7 つのステージからなり，次の順番で実行される．

1. シリアルの初期化
2. MMU の初期化
3. 割り込みの初期化

4. スケジューラの初期化

5. プロセスの初期化

6. ファイルシステムの初期化

7. 割り込み許可

各ステージの実装は次の通りである．

### シリアルの初期化

シリアルの初期化では，転送されるフレームのデータサイズの設定や，転送速度の設定を行う．

### MMU の初期化

MMU の初期化では，ページテーブルの初期化，メモリマッピングの設定およびアクセス権の設定を行う．

ページテーブルの初期化では，ページテーブル構造体を作成し，マッピングするメモリの仮想アドレスおよび対応する物理アドレス，マスタページテーブルのアドレス，ページテーブルの種類，ドメインをそれぞれ設定する．本実装では，Coarse ページテーブルを採用した．次に，ページテーブルの全てエントリを FAULT エントリで初期化する．初期化するページテーブルは，1 つの L1 ページテーブルおよび 6 つの L2 ページテーブルである．

メモリマッピングの設定では，実際にマップされるメモリのアドレスや範囲を決定する．マップされるのは，カーネル領域，ページテーブル領域，割り込みに関する領域，ユーザ領域，RAM ディスク領域，シリアル領域，タイマに関する領域などである．

ここでは，上記の各領域を Region 構造体に設定してメモリマッピングを行う．まず，Region 構造体に，マッピングする仮想アドレスおよび物理アドレス，1 ページのサイズ，ページ数，アクセス許可，キャッシュおよびバッファ，マスターページテーブルのアドレスを設定する．アクセス許可では，カーネル領域，割り込みに関する領域，シリアルやタイマ等に関するレジスタは，全てカーネルからのみアクセス可能とした．また，RAM ディスク (4.8 参照) の領域は，特権モードでは読み書き可能，ユーザモードでは読み込み専用とした．ユーザ領域は全モードからアクセス可能とした．これらのアクセス権の設定を有効にするために，ドメインのアクセス許可ビットを，ページテーブルエントリの設定が有効なる用に設定を行った．キャッシュとバッファは使用していない．最後に，Region 構造体の情報をページテーブルエントリに設定し，L2 ページテーブルのアドレスを L1 ページテーブルのエントリにセットして，メモリマッピングの設定は完了である．本実装におけるメモリマッピングは図に 4.1 に示した通りである．

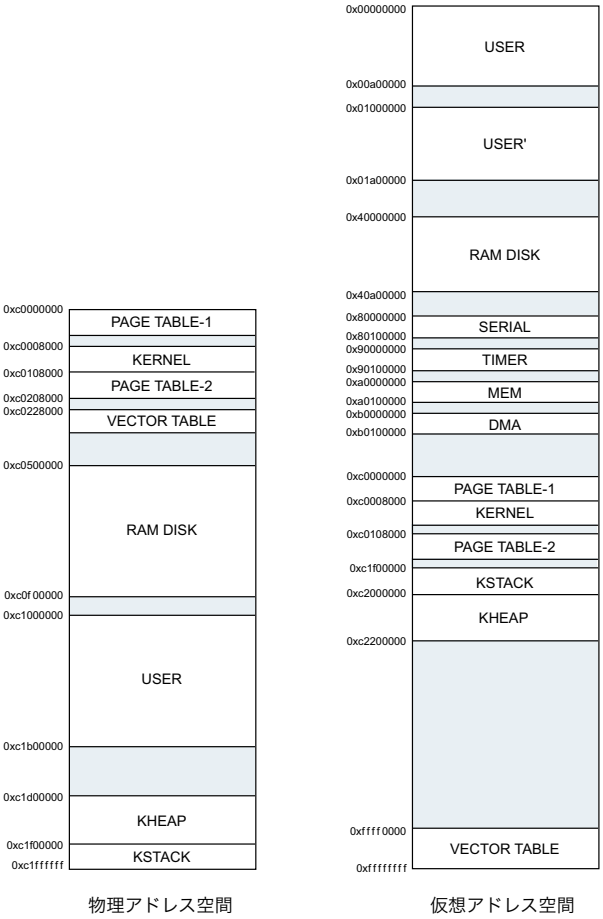


図 4.1: ARM-OS/161 メモリマップ

### 割り込みの初期化

割り込みの初期化では，まず，割り込みで使用するスタックを設定する．次に，ベクターテーブルの配置を行う．ベクターテーブルはカーネルイメージ内に既に作成されているが，ベクターテーブルを置くアドレスは 0xffff0000 に指定されている．そこで，カーネルイメージにあるベクターテーブルを，ブート時にアドレス 0xffff0000 へコピーを行う．最後にタイマの初期化を行い割り込みの設定が完了する．タイマの初期化は 4.7.1 で説明する．

### スケジューラの初期化

スケジューラの初期化では，プロセスの情報を保存するためにスケジューラが使用するキューを作成している．

#### プロセスの初期化

プロセスの構造体を生成し，全情報を初期化する．ブート時に生成されるプロセスは，起動時に各初期化を実行しているプロセスのことで，最初生成されるプロセスである．

#### ファイルシステムの初期化

本実装では，ファイルシステムに，構造が単純で仕様も明確である点を考慮して FAT12 を採用した．FAT12 は RAM ディスク (4.8 参照) 上に実現されている．

ブートプロセスでは，まず RAM ディスクのデバイス名を設定し，FAT12 を VFS に追加する．以後，FAT12 ファイルシステムへのアクセスは，VFS を経由して行われるようになる．FAT12 ファイルシステムへのアクセスに VFS を経由することで，高い抽象度を実現している．

次に FAT12 ファイルシステムの初期化を行う．FAT12 の先頭にある BPB(BIOS Parameter Block) 領域には，FAT12 に関する様々な情報が格納されている．この情報を読み取ることで，1 セクタ当たりのバイト数や 1 クラスタ当たりのセクタ数などを取得できる．次に，読み取った情報を元に，ルートディレクトリの位置や番号，データの開始位置を計算する．最後に，現在のディレクトリをルートディレクトリにセットして，ファイルシステムの初期化が完了する．

FAT12 ファイルシステムについては，4.8 にて説明する．

#### 割り込み許可

最後に，cpsr レジスタの I ビット及び F ビットをクリアすることによって，割り込み許可状態にする．

## 4.3 メモリ管理

プロセスのメモリ保護は各プロセスに固有のメモリ空間を与えることで，他のユーザプロセスからメモリを保護している．各プロセスはそれぞれ，仮想メモリ 0x00000000 から開始するメモリ空間をそれぞれ割り当てられる．コンテキストスイッチが発生すると，メモリのマッピングを変更し，次に実行されるプロセスのメモリ空間を仮想メモリ 0x00000000 から開始するようにする．1 つのユーザプロセス実行中は，他のプロセスのメモリ空間はユーザモードでアクセス可能な領域にはマッピングされていないので，誤って他のプロセスのメモリにアクセスすることはない．

例えば，プロセス 1 が現在実行中であり，図 4.2 のようにプロセス 1 のメモリがマッピングされているとする．コンテキストスイッチが発生し，実行するプロセスがプロセス 2 に切り替わる場合，図 4.3 のようにプロセス 1 のマッピングは解除され，代わりにプロセス 2 のメモリ空間がマッピングされる．このとき，プロセス 1 のメモリ空

間はマッピングされていないので，プロセス 2 はプロセス 1 のメモリ空間にアクセスすることはできない．

この仕組みを利用してメモリ保護を実現している．

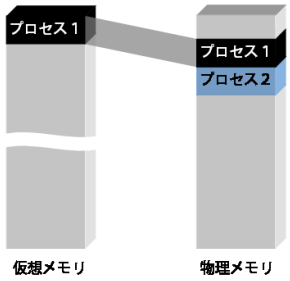


図 4.2: プロセス 1 のメモリ空間

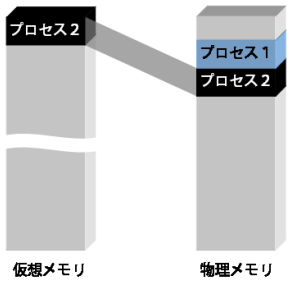


図 4.3: プロセス 2 のメモリ空間

## 4.4 例外処理

割り込みが発生すると現在実行されているプロセスは強制的に中断される．中断されたプロセスのレジスタはスタックに待避され，割り込み処理を実行した後，レジスタを復元しプロセスを再開する．割り込みには，ソフトウェア割り込み，iqr, data abort など様々あるが，レジスタの待避・復元方法は基本的に同じである．しかし，割り込み発生時のプロセッサモードによって，レジスタの待避・復元方法が異なる．

割り込みが発生すると，中断された時点での PC および cpsr の値が自動的に保存される．その後 svc モードに移行し，保存された cpsr の値を調べ割り込み発生時のプロセッサモードを取得する．

まず，割り込み発生時のプロセッサモードが User モードで合った場合を考える．r0 から r12 までは通常のロードストア命令で保存することができる．しかし，r13, r14 レジスタはバンクレジスタであるので，SVC モードから User モードの r13, r14 レジスタを直接参照することができない．そこで，特権モードから User モードのレジス

タにアクセスする特殊な命令を使用して，User モードのレジスタを保存する．

一方，割り込み発生時のプロセッサモードが SVC モードであった場合，現在 SVC モードで実行しているので，そのまま可視状態にあるレジスタを保存する．

## 4.5 プロセス管理

プロセスの切り替えはスケジューラのによって行われる．スケジューラは，タイマ割り込みが 5 回発生すると呼び出される．つまり，50 ミリ秒に 1 度の頻度で呼び出される．スケジューラは実行待ち状態にあるプロセスのキューを調べ，キューの先頭にあるプロセスを取りだし，次に実行するプロセスを選択する．中断されたプロセスと次に実行されるプロセスを比較し，同じであった場合はスケジューラを終了し中断されたプロセスを再開する．2 つのプロセスが異なる場合，まず，L1 ページテーブルエントリを書き換え，次に実行するプロセスが使用するメモリ領域を再マッピングする．次に，実行を中断されたプロセスのレジスタ情報等をプロセスの構造体に保存し，次に実行するプロセスのレジスタを復元する．レジスタを復元した時点でプログラムカウンタの値も復元されるので，即座に次のプロセスに実行が遷移する．

## 4.6 システムコール

システムコールの実装について説明する．システムコールは，swi 命令によりソフトウェア割り込みを発行してカーネルモードに移行した後，実際の処理が実行される．swi 命令には 24 ビットの値を含めることができるので，ここにシステムコールの番号を入れることにより，システムコールを識別できる．ソフトウェア割り込みによってカーネルモードに移行すると，システムコールの番号に応じて，対応する関数を実行する．関数の実行を終えると，ソフトウェア割り込みを発行した命令の次のアドレスから通常の実行を再開する．

## 4.7 デバイスドライバ

### 4.7.1 タイマ

StrongARM SA-1110 には，OS Timer Count Register(以下 OSCR) があり，OSCR の値は 1 秒間に 3686400 回インクリメントされる．OSCR が OS Timer Match Register(OSMR) の値と一致した場合は，OS Timer Status Register(以下 OSSR) の対応したビットがセットされ，タイマ割り込みが発生する．最初にタイマの初期化について説明する．まず，ブートプロセスで OSCR を 0 にリセットし，OSMR の値を 36864 にセットする．36864 クロックは 10 ミリ秒に相当するので，タイマ割り込みは 10 ミリ秒ごとに発生する．次に，OSSR をクリアして割り込み状態を初期化し，最後に OS Timer Interrupt Enable Register(OIER) の対応するビットをセットしてタイマ割り込みを許可する．以上でタイマの初期化が完了する．

次に，タイマ割り込みの処理について説明する．タイマ割り込みが発生すると，まず，OSSR をクリアして割り込み状態を解除する．次に，OSMR の値に 36864 を加算して次のタイマ割り込みの発生時刻を 10 ミリ秒先にセットする．最後に，OSMR の値が OSCR よりも小さいか調べ，OSMR の値の方が小さい場合は，OSMR の値の方が大きくなるまで OSMR の値に 36864 を加算し続ける．

### 4.7.2 シリアル

シリアルを経由したコンソールの入出力は，次のように行われる．

シリアル出力では，UART Status Register(UTSR) を調べ送信用 FIFO キューに空きがあれば，データを UART Data Register(UTDR) に書き込む．FIFO キューに空きがない場合は，空くまでポーリングし続ける．

シリアル入力では，UTSR を調べ受信用 FIFO キューが空でなければ，データを UTDR から取得する．FIFO キューが空の場合は，FIFO キューにデータが入るまでポーリングし続ける．

## 4.8 ファイルシステム

OS/161 では，図 4.4 左側のようにシミュレータを通してホストマシンのファイルにアクセスする Simulator Through ファイルシステムが使用されている．CerfCube 上で動作する ARM-OS/161 では，同じ様な環境を再現することはできないので，図 4.4 右側のように，FAT12 ファイルシステムを利用して RAM ディスク上のファイルにアクセスするように実装した．

FAT ファイルシステムは，MS-DOS の初期から用いられ，その後 Windows にも採用されたファイルシステムである．FAT ファイルシステムは，単純な構造をしているため，ファイルアクセスの仕組みは比較的シンプルである．本実装では，東京工業大学の権藤克彦 助教授が開発した教育用 OS である udos のファイルシステム部分を，ARM-OS/161 に移植を行った．

RAM ディスクとは，メモリを仮想的にディスクのように利用する仕組みのことである．RAM ディスクは，ファイルシステムのイメージを作成し，そのイメージをメモリに展開じて利用する．本実装では，FAT12 でフォーマットされたフロッピィディスクからイメージを作成した．

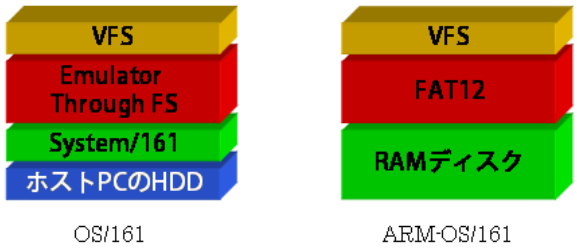


図 4.4: OS/161 と ARM-OS/161 のファイルシステム

# 第5章　評価

## 5.1　測定対象

OS/161 を CerfCube に移植するに当たり，RAM ディスクや FAT12 ファイルシステムを OS/161 に実装した．そこで，RAM ディスクやファイルシステムに関する箇所の評価として，プログラムの実行時間の計測を行った．

ARM-OS/161 におけるプログラムの実行は，一般的な fork, exec システムコールを組み合わせた，fork_exec システムコールによって行われる．fork_exec システムコールが実行されると，親プロセスは子プロセスの構造体を生成し，実行するファイルをオープンし，ファイルのヘッダを読み取る．ヘッダの情報を元にプログラムの Text 領域，Data 領域をメモリに確保した後，Text 領域，Data 領域をメモリにコピーする．次に，ページテーブルのエントリを設定して，子プロセスのメモリのマッピングを行う．最後に，プログラムのエントリーポイント，スタックポインタ，cpsr をそれぞれ設定する．子プロセスが生成されると親プロセスはスリープする．再スケジューリングによって，生成された子プロセスがディスパッチされると，子プロセスの実行が開始する．

本研究の評価では，プログラムの実行を処理ごとに区切り，各実行時間を計測した．実行したプログラムは，Data 領域に値を書き込むだけのプログラムで，プログラムサイズは表 5.1 に示した通りである．ただし，本実装ではページのサイズ 4k バイトであり，実行するプログラムはシェル上で実行させるものとする．

| 領域 | 大きさ | ページ数 |
|---|---|---|
| Data | 128k バイト | 32 ページ |
| Text | 128k バイト | 32 ページ |
| 計 | 256k バイト | 64 ページ |

表 5.1: 実行プログラムのサイズ

## 5.2　測定方法

実行時間の計測には，OSCR の値を直接参照する手法を用いた．計測したい箇所に次のようなインラインアセンブラのコードを埋め込むと，このコードが実行された時点で，コプロセッ0 のレジスタにアクセスを試みようとし，Undefined Instruction 例

外が発生する．

```
asm("mcr p0, 0, r0, c0, c0":::"r0");
```

Undefined Instruction 例外ハンドラでは，OSCR の値を取得し，スタックにその値を保存する．例外を発生させるため，User モードと Kernel モードをまたぐ計測も可能である．

最後に，全計測を終えた時点で，スタックに保存された値を表示する．

また，1 箇所の測定のオーバーヘッドは 5clock である．この値は測定時間と比較すると無視できるほど小さい．よって，計測に要する時間が結果にほとんど影響を与えていないといえる．

## 5.3 測定結果 1

測定区間は次の 7 カ所である．

- プロセスの生成 ：親プロセスが子プロセスを生成
- ディレクトリの検索 ：RAM ディスクから目的のファイルを検索
- ヘッダ読み取り ：ELF ヘッダを読み取る
- ファイルのロード ：プログラムヘッダを読み，ファイルをメモリにロード
- その他設定 ：ページテーブルエントリやレジスタの設定
- 実行開始 ：スケジューリングされるまでの時間
- データ領域への書き込み ：データ領域へ書き込む時間の総計

測定結果を表 5.2 および図 5.1 に示す．

| 測定区間 | 所要時間 [clock] | 所要時間 [msec] |
|---|---|---|
| プロセス生成 | 3254.2 | 0.882758247 |
| ディレクトリ検索 | 6786.2 | 1.840874566 |
| ヘッダ読み取り | 47074.2 | 12.76969401 |
| ファイルのロード | 963394.6 | 261.3375109 |
| その他設定 | 1710 | 0.463867188 |
| 実行開始 | 428.6 | 0.116265191 |
| Data 領域への書き込み | 5.6 | 0.001519097 |
| 合計 | 1022653.4 | 277.4124891 |

表 5.2: プログラムの通常の実行時間の詳細

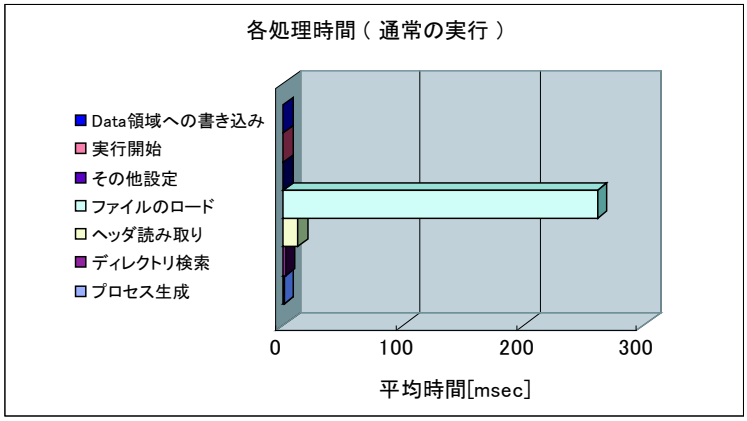


図 5.1: プログラムの通常の実行時間

以上，表 5.2 および図 5.1 から，ファイルのロードに最も時間を要していることが分かる．ファイルのロードは，図 5.2 に示したように，RAM ディスク上に配置されているファイルの Text 領域および Data 領域を，共にユーザ空間のメモリにコピーしている．次節では，このコピーのオーバーヘッドを削減する手法を検証する．

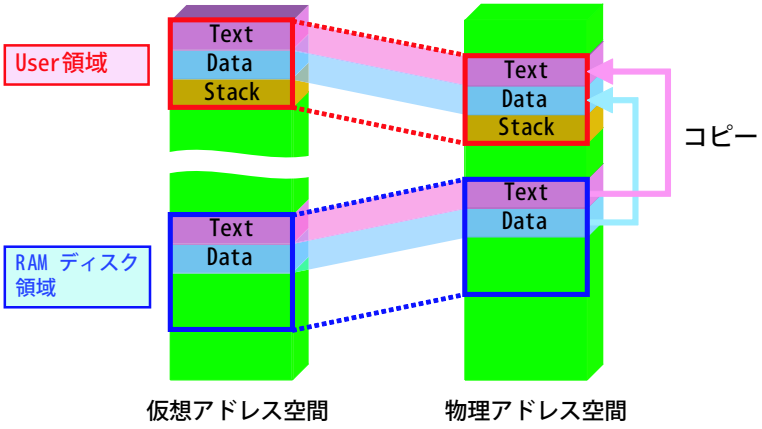


図 5.2: 通常のロード

## 5.4 高速化手法

### 5.4.1 Copy-Only-Data

プログラムの Text 領域には，実行されるコードが含まれている．プログラムを実行中に自らこのコード書き換えることはほとんどはないので，通常は Text 領域に対する書き込みは発生しない．よって，プログラムロード時のコピーを削減する方法として，

プログラムの Text 領域のコピーを行わないという手法が考えられる．本論文では，このように Text 領域のコピーを行わず Data 領域のみコピーする手法を Copy-Only-Data と呼ぶ．

Copy-Only-Data を用いた場合，図 5.3 に示したように，プログラムの実行時は RAM ディスク上の実行ファイルの Text 領域が参照される．つまり，RAM ディスク上の Text 領域を，本来 Text 領域がコピーされるであろう仮想アドレスに対してマッピングを行えばよい．この場合，Text 領域は読み取り専用でマッピングを行う．また，Data 領域は通常通りコピーを行う．

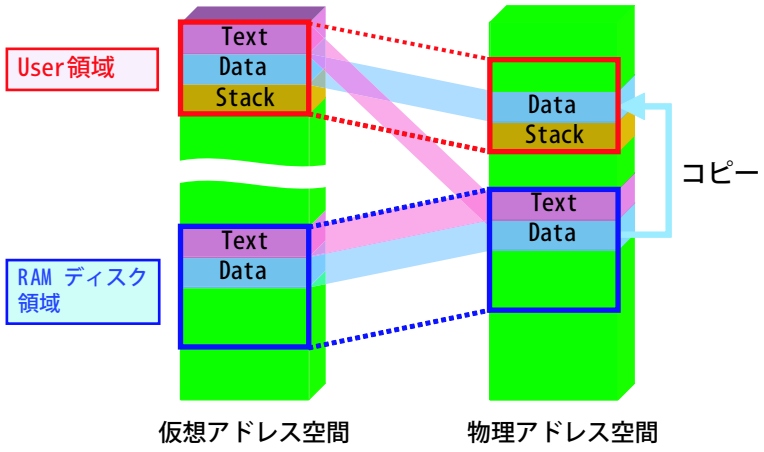


図 5.3: Copy-Only-Data

### 5.4.2 Copy-on-Write

プログラムの Data 領域に対して，必ずしも書き込みが行われるとは限らない．Data 領域に書き込みが発生しない場合は，Data 領域のコピーも行わない方がオーバーヘッドが少なくなる．よって，プログラムロード時のコピーのオーバーヘッドをさらに削減する方法として，プログラムの Text 領域および Data 領域のコピーを行わないという手法が考えられる．

本実装では，Copy-on-Write という手法を用いた．Copy-on-Write は，データに対して書き込みが発生した場合に，そのデータのコピーを作成して書き込みを行なうという手法である．

Copy-Only-Data の場合と同様に，プログラムの実行時は RAM ディスク上の実行ファイルの Text 領域および Data 領域が参照され，共に読み取り専用でマッピングされる．この概略を，図 5.4 に示す．

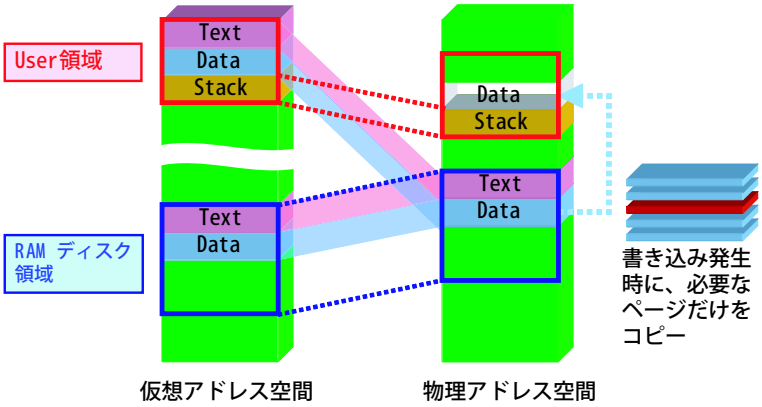


図 5.4: Copy-on-Write

ここで，問題となるのが Data 領域に対して書き込みが発生した場合である．本実装では，Data 領域に対する書き込みが発生した場合は次のような流れで処理が行われる．

1. Data Abort が発生
2. Data Abort が発生したアドレスを取得
3. 命令の種類を判別
4. 書き込み先メモリアドレスを特定
5. 書き込み先メモリアドレスを調べる
6. メモリマッピングを変更
7. ページをコピー
8. 書き込み命令を再実行

各処理について説明していく．

### Data Abort が発生

Data 領域は読み取り専用でマッピングされているため，書き込みを行おうとすると Data Abort が発生する．

### Data Abort が発生したアドレスを取得

Data Abort が発生すると，自動的に Abort モードへ移行する．その際，r14_abt レジスタに自動的に発生時のプログラムカウンタが保存される．このプログラムカウン

タの値は，Data Abort が発生した命令の 2 命令先を指しているので，プログラムカウンタから 8 バイト分差し引いたアドレスが，Data Abort が発生したアドレスである．

#### 命令の種類を判別

Data Abort が発生したアドレスから命令を取得し，その命令がメモリに対して書き込みを行う命令かどうかを調べる．メモリに書き込みを行う命令は，STRB, STR, STRH, STRD, STMDA, STMIA, STMDB, STMIB である．その他の命令の場合は，通常の Data Abort の処理を実行する．

#### 書き込み先メモリアドレスを特定

取得した命令から書き込み先アドレスを算出する．書き込み先アドレスは，ベースアドレスにオフセットを加算した値で求められる．ベースアドレスはレジスタを指定するが，オフセットはレジスタで指定されている場合，即値で指定されている場合，レジスタの値をシフトした値で指定されている場合が考えられる．また，それぞれの場合に対して，メモリに書き込みを行う前にオフセットを加算する場合，書き込みを行った後にオフセットを加算する場合がある．

#### 書き込み先メモリアドレスを調べる

書き込み先メモリアドレスが RAM ディスク領域かどうかを調べる．RAM ディスクの領域外の場合は，通常の Data Abort の処理を実行する．

#### メモリマッピングを変更

ページテーブルエントリを書き換え，メモリマッピングを変更する．Data 領域がコピーされるメモリ領域を，メモリのユーザ空間に読み書き可能状態でマッピングする．

#### ページをコピー

書き込み先メモリアドレスを含む 1 ページを，メモリのユーザ空間にコピーする．

#### 書き込み命令を再実行

Data Abort が発生したメモリへの書き込みを行う命令を再び実行する．書き込み許可状態でマッピングされているので，先ほど中断された書き込みが行われる．

## 5.5 測定結果 2

表 5.3 および図 5.5 に Copy-Only-Data の実行時間を示す.

| 測定区間 | 所要時間 [clock] | 所要時間 [msec] |
|---|---|---|
| プロセス生成 | 3168.4 | 0.859483507 |
| ディレクトリ検索 | 6786 | 1.840820313 |
| ヘッダ読み取り | 47080.4 | 12.77137587 |
| ファイルのロード | 497978.2 | 135.0852322 |
| その他設定 | 1716 | 0.465494792 |
| 実行開始 | 428.8 | 0.116319444 |
| Data 領域への書き込み | 5.2 | 0.00141059 |
| 合計 | 557163 | 151.1401367 |

表 5.3: Copy-Only-Data の実行時間の詳細

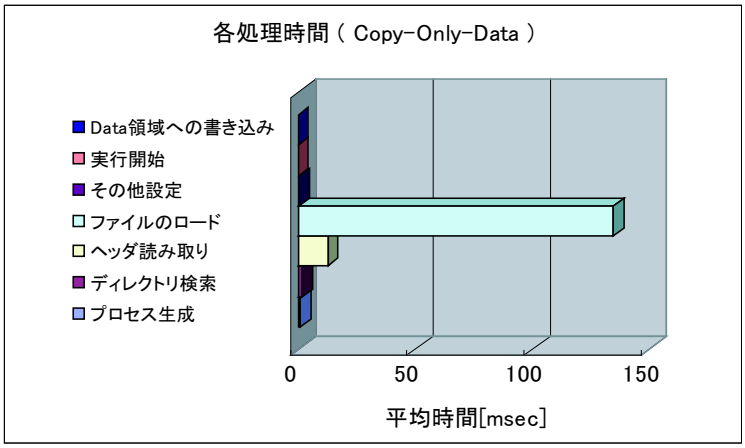


図 5.5: Copy-Only-Data の実行時間

Copy-on-Write を用いた手法では，コピーが発生するページ数に応じて実行時間が変化することが考えられる．そこで，ページのコピーが全く発生しない場合，5, 10, 15, 20, 25, 30, 31 ページのコピーが発生する場合，全ページ (32 ページ) のコピーが発生する場合の実行時間をそれぞれ計測した．ここでは，ページのコピーが全く発生しない場合および全ページのコピーが発生する場合の計測結果を示す．

表 5.4 および図 5.6 は，Copy-on-Write を用いてページのコピーが 1 度も発生しなかった場合の実行時間である．Copy-on-Write を用いた場合，ページのコピーが 1 度も発生しないのは最も高速なケースである．

| 測定区間 | 所要時間 [clock] | 所要時間 [msec] |
|---|---|---|
| プロセス生成 | 3333 | 0.904134115 |
| ディレクトリ検索 | 6785.2 | 1.840603299 |
| ヘッダ読み取り | 47082.4 | 12.7719184 |
| ファイルのロード | 32560 | 8.832465278 |
| その他設定 | 1721.6 | 0.467013889 |
| 実行開始 | 429 | 0.116373698 |
| Data 領域への書き込み | 5.2 | 0.00141059 |
| 合計 | 91916.4 | 24.93391927 |

表 5.4: Copy-on-Write (0 page copy) の実行時間の詳細

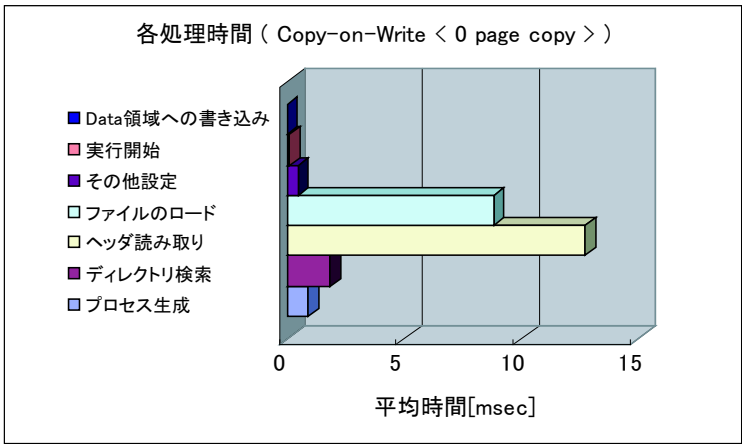


図 5.6: Copy-on-Write (0 page copy) の実行時間

表 5.5 および図 5.7 は，Copy-on-Write を用いて全てのページに対してコピーが発生した場合の実行時間である．Copy-on-Write を用いた場合，全ページのコピーが発生するのは最も最悪なケースである．

| 測定区間 | 所要時間 [clock] | 所要時間 [msec] |
|---|---|---|
| プロセス生成 | 3255.6 | 0.883138021 |
| ディレクトリ検索 | 6785.4 | 1.840657552 |
| ヘッダ読み取り | 47082.4 | 12.7719184 |
| ファイルのロード | 32560.2 | 8.832519531 |
| その他設定 | 1721.6 | 0.467013889 |
| 実行開始 | 428.6 | 0.116265191 |
| CopyOnWite の処理 | 6457.2 | 1.751627604 |
| ページのコピー | 466560.8 | 126.562717 |
| 合計 | 564851.84 | 153.2258572 |

表 5.5: Copy-on-Write (32 page copy) の実行時間の詳細

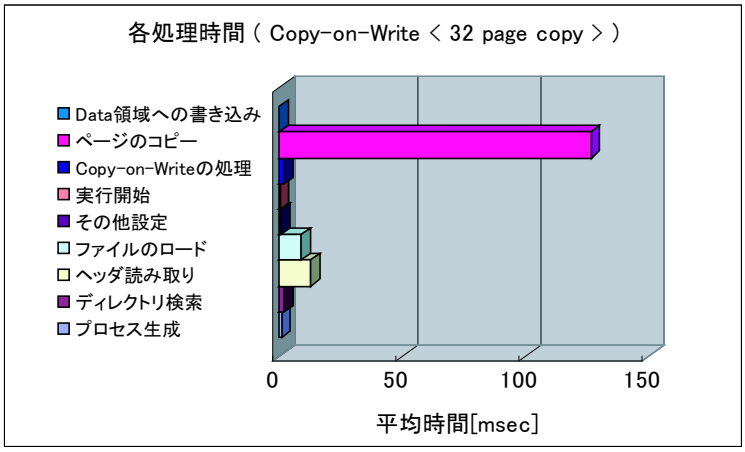


図 5.7: Copy-on-Write (32 page copy) の実行時間

# 第6章　考察

プログラムロード時の高速化にあたり，Copy-Only-Data と Copy-on-Write の 2 つの手法を実装した．計測結果をもとに 2 つの手法の有効性について考察する．

表 6.1 および図 6.1 は，ページコピー発生回数と実行時間の関係を示したものである．ただし，Copy-on-Write 以外の場合は，ページコピーの発生回数を考慮する必要がないので，一定値となっている．

| コピー回数 | 通常の実行<br>実行時間 [msec] | Copy-Only-Data<br>実行時間 [msec] | Copy-on-Write<br>実行時間 [msec] |
|---|---|---|---|
| 0 | | | 24.93391927 |
| 5 | | | 44.96066623 |
| 10 | | | 65.01112196 |
| 15 | | | 85.03678385 |
| 20 | 277.4124891 | 151.1401367 | 105.0857205 |
| 25 | | | 125.1529405 |
| 30 | | | 145.2042101 |
| 31 | | | 149.1930339 |
| 32 | | | 153.2258572 |

表 6.1: ページコピー発生回数と実行時間の関係の詳細

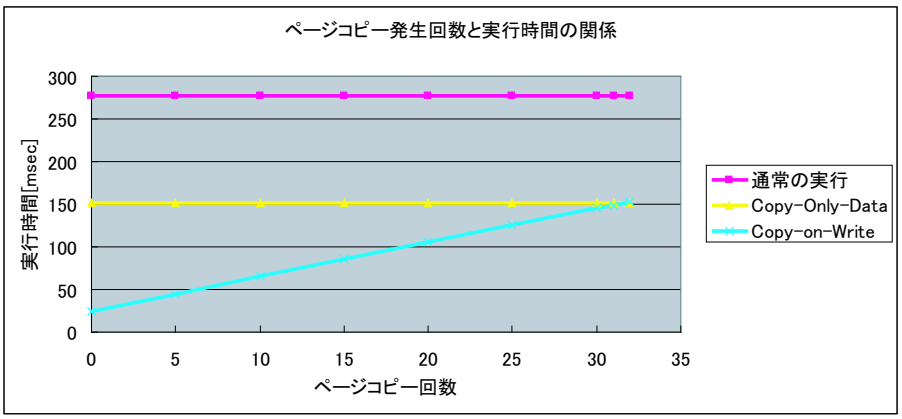


図 6.1: ページコピー発生回数と実行時間の関係

### Copy-Only-Data

最初に，Copy-Only-Data について考察する．表 6.1 および図 6.1 から，通常の実行時間に約 277.4 ミリ秒を要しているのに対して，Copy-Only-Data では約 151.1 ミリ秒に短縮されていることが分かる．つまり，全体で約 1.84 倍に高速化している．

また，測定結果を表 5.2 と図 5.1 および表 5.3 と図 5.5 から，Copy-Only-Data の手法を用いると，プログラムロードに要する時間が約 261.3 ミリ秒から約 135.1 ミリ秒へ約半分 (52%) に減少している．これは，実行プログラムの Text 領域と Data 領域の大きさがそれぞれ同じサイズであることに由来する．つまり，Copy-Only-Data の手法により Text 領域のコピーを行わなくなったので，全コピー量に占める Text 領域の割合分 (50%) だけ実行時間が減少したと考えられる．

以上より，プログラムの実行に関して Copy-Only-Data の手法は有効であるといえる．また，プログラムに占める Text 領域の割合が大きくなるほど，この手法がより有利になると考えられる．

### Copy-on-Write

次に，Copy-on-Write について考察する．表 6.1 および図 6.1 から，通常の実行時間に約 277.4 ミリ秒を要しているのに対して，Copy-on-Write でページのコピーが全く発生しなかった場合では約 25.0 ミリ秒に大幅に短縮されていることが分かる．つまり，全体で約 11.1 倍に高速化している．一方，全ページのコピーが発生した場合は，約 153.2 ミリ秒の時間を要している．つまり，全体で約 1.81 倍に高速化している．以上より，通常のプログラムの実行と比較すると，Copy-on-Write 最悪なケースであっても十分有効な手法であるといえる．また，Data 領域に対する書き込みによって発生するページのコピー回数が少ないほど，この手法がより有利になると考えられる．

Copy-on-Write を用いた手法では，Data 領域のページのコピー回数が増加するほどパフォーマンスが低下する．そこで，Copy-Only-Data と Copy-on-Write を比較し，Copy-on-Write のオーバヘッドを検証する．

Copy-on-Write でページのコピーが全く発生しない場合は，Copy-Only-Data と比べ約 5.4 倍高速である．このケースでは，Copy-on-Write を用いた手法が最速であり，Copy-on-Write のオーバヘッドは全く問題にならないといえる．一方，31 ページのコピーを行う場合は約 149.2 ミリ秒であり，Copy-Only-Data よりも未だ高速である．全ページ (32 ページ) のコピーが発生する場合は約 153.2 ミリ秒であり，Copy-Only-Data よりも低速である．しかし，実行時間の差は約 2.1 ミリ秒であり，この差は全実行時間の約 1.39%$(2.1 \div 151.1 \times 100)$ に過ぎない．これより，Copy-on-Write のオーバヘッドはほとんど無視できるほどに小さいといえる．

以上から，プログラムの実行時における Copy-on-Write 方式の利用は，非常に有効であるといえる．

# 第7章　結論

本研究では，教育用 OS である OS/161 を ARM プロセッサ向けに移植を行った．移植ではメモリ管理や割り込み処理といったプロセッサ依存の箇所書き換える必要があるため，OS の核となる部分に関して理解を深めることができるということ分かった．

また，プログラム実行の計測を行い，最大のオーバヘッドはプログラムのロードであるということが分かった．そこで，プログラムのロードを高速化する手法として，Copy-on-Write 方式を実装した．Copy-on-Write を用いた方法は，通常のプログラムロードに比べ最大 11 倍高速化され，最悪のケースでもオーバヘッドは 1.39%と無視できるほど小さいということが分かった．

以上から，プログラム実行を高速化する手法として Copy-On-Write は非常に有効であるといえる．

# 謝辞

本研究の機会を与えてくださり，懇切丁寧なご指導を賜りました中島達夫教授に心より感謝，御礼申し上げます．また，いつも丁寧に的確な助言をして下さった追川修一助教授，研究室で大変お世話になった岩崎 匡寿氏，菅谷みどり氏，茂田井寛隆氏に深く感謝致します．最後に，共に励まし合い本研究を行った杵渕雄樹氏，花岡健介氏，そして腰前秀成氏に感謝します．

# 関連図書


[1] Andrew N. Sloss, Dominic Symes, Chris Wright, John Rayfield 著: ”Arm System Developer’s Guide -Designing and Optimizing System Software”, Morgan Kaufmann Publishers

[2] 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 (2001)

[3] David A. Holland, Ada T. Lim,1 and Margo I. Seltzer: ”A New Instructional Operating System”, Harvard University

[4] The udos project
http://www.sde.cs.titech.ac.jp/ gondow/udos/

[5] はじめて読む 486
蒲地 輝尚 著
株式会社アスキー 2002

[6] 詳細リナックスカーネル (Understanding the Linux Kernel 邦訳)
Daniel P.Bovet, Marco Cesati 著, 高橋 浩和, 早川 仁 監訳, 岡島 順治郎, 田宮 まや, 三浦 広志 訳
O’REILLY (株式会社オライリー・ジャパン) 2002

[7] 第 2 版デバイスドライバ (Linux Device Driver 邦訳)
Alessandro Rubini, Jonathan Corbet 著, 山崎 康宏, 山崎 邦子, 長原 宏治, 長原 陽子 訳
O’REILLY (株式会社オライリー・ジャパン) 2002