2004年度 卒業論文

組み込み向けMINIXの
プロセス管理に関する研究

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　1g01p080-3
氏名　花岡 健介

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 ARM **3**

2.1 ARM の概要 3

2.1.1 ARM のレジスタ 3

2.1.2 Current Program Status Register 4

2.1.3 ARM のパイプライン 7

2.2 Interrupt 8

2.3 Memory Management Unit 11

2.3.1 ページテーブル 11

2.3.2 Translation Lookaside Buffer 13

第3章 開発環境 **16**

3.1 クロスコンパイル環境 16

3.2 CerfCube 16

3.2.1 CerfCube 16

3.2.2 I-Boot 18

3.3 MINIX 19

3.3.1 MINIX の概要 19

3.3.2 MINIX の構成 19

第4章 設計・実装 **21**

4.1 移植の手順 21

4.2 ブートシーケンス 21

4.3 プロセス・タスクの特権レベル 23

4.4 システムコール 23

4.5 割り込み 23

4.5.1 割り込みの初期化 23

4.5.2 レジスタの退避・復元 24

4.6 メモリ保護 24

4.7 デバイスドライバ 26
4.7.1 TTY タスク 26
4.7.2 メモリタスク 26
4.7.3 クロックタスク 26

第 5 章 評価・考察 27
5.1 評価方法 27
5.2 Linux との比較 27
5.3 MINIX におけるオーバーヘッド 28
5.3.1 IPC 28
5.3.2 fork,exec のコストの内訳 29
5.4 改善案 29
5.4.1 コピー 30
5.4.2 IPC 30

第 6 章 関連研究 31
6.1 Fast Address-Space Switching 31
6.2 TLB の共有 31

第 7 章 結論 32

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 パイプライン実行例 8
2.5 IRQ 発生時の動作 9
2.6 L1 page table entries 12
2.7 TTB 13
2.8 L2 page table entries 13
2.9 L1 page table walk 14
2.10 L2 page table walk 14

3.1 CerfCube 17
3.2 MINIX の構成 20

4.1 x86-MINIX のブート 22
4.2 ARM-MINIX のブート 22
4.3 ベクタテーブルの転送 24
4.4 x86 でのメモリ保護 25
4.5 ARM でのメモリ保護 25

5.1 ls 実行時の fork のコストの内訳 29
5.2 ls 実行時の exec のコストの内訳 29
5.3 1byte コピー 30
5.4 4byte コピー 30

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 各例外の優先度，および発生時のモード，ビットの対応 10
2.4 ベクタテーブル 11
2.5 MMU のサポートするページテーブル 11

3.1 CerfCube のハードウェア仕様 17
3.2 CerfCube のメモリマップ 18

4.1 レイヤと特権レベル・動作モード 23

5.1 Linux と MINIX の比較 (単位：μsec) 28
5.2 各コマンドのサイズ (単位：byte) 28
5.3 IPC の発生回数 (単位：回) 28

概 要

近年，PDA や携帯電話などの組み込み機器は非常に高機能化してきている．それに伴って OS も年々高機能化してきているが，同時に構造が複雑化してきている．本研究の目的は，比較的規模の小さな OS である MINIX を ARM アーキテクチャへ移植し，OS の基本的な動作を理解した上で、組み込み機器の持つ要求を満たすために必要な OS の機能について議論することにある．移植を行った MINIX の性能評価として，プロセス管理において重要な fork および exec システムコールのコストを計測し，同じ ARM 上で動作する Linux との比較を行った．MINIX は Linux と比較して 30 倍近く遅いという結果を得たが，MINIX のオーバーヘッドとなっている部分について考察を行い，組み込み機器への適用可能性を議論した．

Abstract

In these years, performance of embedded systems such as PDA or cellular phones are getting higher. Performance of operating system is getting higher, at the same time hits structure is getting complicated. In our research we ported MINIX, which scale is comparatively small, to the ARM architecture. Then we measured the execution time of fork() and exec() system call function which are important for process management, and compared the performance of ported MINIX with original Linux which also runs on the ARM architecture. The result shows that MINIX is about 30 times slower than Linux. We considered about the overhead of MINIX, and discussed its application possibility to the embedded system.

第1章　序論

1.1　背景

従来の組み込み OS はプログラムが一定の時間内に実行を終えることを保証するリアルタイム性や，少ないメモリでプログラムを動作させることに重点を置いてきた．しかし，近年の組み込み機器のめざましい発展により組み込み OS に要求される機能は増えてきている．組み込み機器がインターネットに接続することも一般的になってきた今日では，セキュリティを考慮しなければならない．そこで，これまでのように省資源を最優先にした OS カーネルと全てのアプリケーションを同じプロテクションドメインで動作させるのではなく，互いに保護できるように，各々を異なるプロテクションドメインで動作させる必要がある．そのため，Linux や FreeBSD のような高機能の OS が組み込みシステムでも用いられるようになってきている．

1.2　目的

Linux や FreeBSD のようにオープンソースな OS が組み込みシステムでも用いられるようになってきたが，コードの量が非常に多く機能も複雑なため，OS の研究は年々難しくなってきている．このような大規模化してきている OS を用いて研究を行うのは困難なため，本研究では比較的規模の小さな OS である MINIX の ARM アーキテクチャへの移植を行った．(以下 ARM 上で動作する MINIX を ARM-MINIX，x86 上で動作する MINIX を x86-MINIX と呼ぶ．) MINIX は規模が小さいだけでなく，マイクロカーネルアーキテクチャを採用したよりモジュール化された設計なので移植性も高い．また，ARM は組み込み用プロセッサとして広く普及しているので，組み込み向け OS の研究に最適である。移植を通して実際にコードを触ることで，理論だけでは学ぶことができない OS の動作を理解することを目的としている．また，プロセス管理の重要な役割であるプロセスの生成にかかるコストおよびプロセスの実行にかかるコストを計測し，ARM-MINIX の性能の評価を行うことで，ARM-MINIX の組み込み機器への適用可能性を議論することを目的とした．

1.3 論文の構成

本論文ではまず第 2 章で移植のターゲットとなる ARM について述べる．そして第 3 章で開発環境として ARM を搭載した CERFCUBE，および移植する OS である MINIX について述べる．第 4 章では x86-MINIX と ARM-MINIX の実装の違いについて解説する．第 5 章では ARM-MINIX の評価および考察を行う．最後に第 7 章で結論を示し，第 8 章で将来課題を述べる．

第2章　ARM

2.1　ARMの概要

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．

2.1.1　ARMのレジスタ

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．プロセッサモードに関しては，2.1.2 節にて詳細に述べる．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．

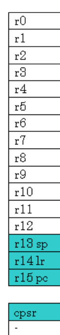

図 2.1: user モードで使用できるレジスタ

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，2.1.2 節にて述べる．

2.1.2 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．

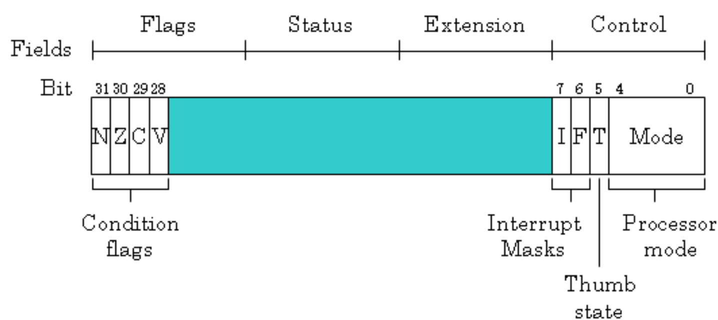

図 2.2: Program status register

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．プロセッサモードによって，使用できるレジスタ群や，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(以下 FIQ), Interrupt Request(以下 IRQ), 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．コンディションフラグに対しては，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

プロセッサモードを変更するには cpsr レジスタを直接書き変えるか，例外や割り込み発生時に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, IRQ, FIQ, Software Interrupt(以下 SWI), 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 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 個のレジスタを示す．この中で影の付いた 20 個のレジスタは，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．

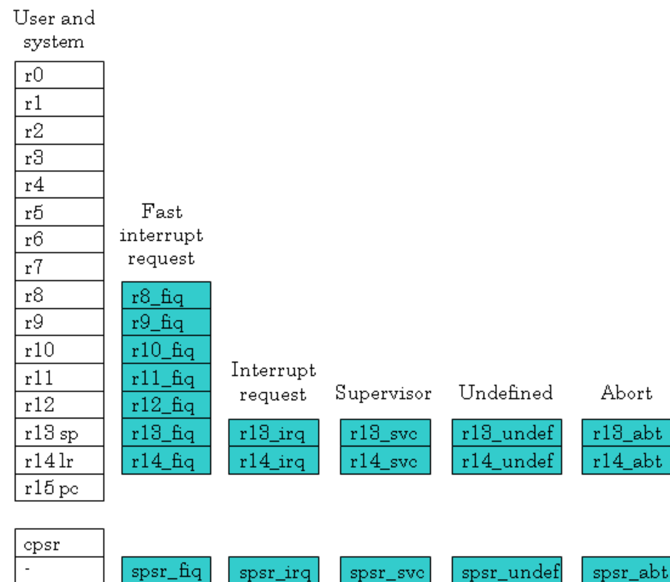

図 2.3: ARM の全レジスタ

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 レジスタを表す．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 IRQ が，F ビットに 1 をセットすると 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

2.1.3　ARM のパイプライン

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．

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．

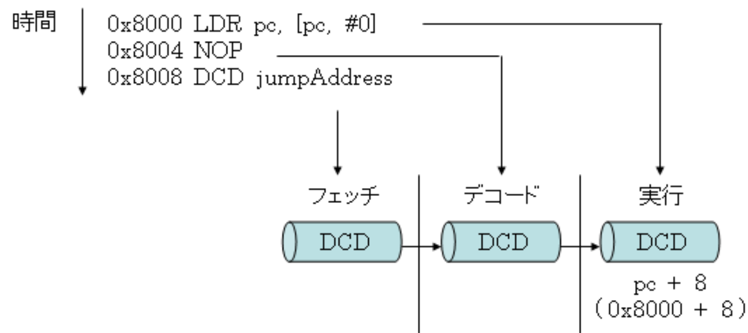

図 2.4: 3 パイプライン実行例

2.2 Interrupt

例外が発生すると ARM は特定のモードへ移行する．例外発生時の動作は,

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

ARM は Reset, Undefined Instruction, SWI, Prefetch Abort, Data Abort, IRQ, 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．

Undefined Instruction は ARM 命令セット，Thumb 命令セット[1]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 2.5 に示す．User モードで IRQ 例外が発生すると，pc の値が IRQ モードのバンクレジスタ pc_irq に，cpsr の値が spsr_irq にコピーされる．同時に cpsr の I フラグが 1 に，モードビットに 10010(IRQ) が設定される．

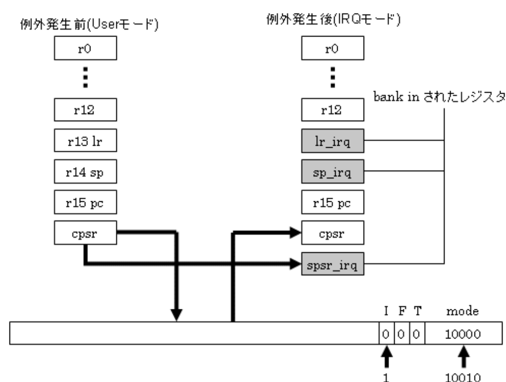

図 2.5: IRQ 発生時の動作

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 2.3 に示す．

[1]16bit に圧縮された命令セット

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 2.3: 各例外の優先度，および発生時のモード，ビットの対応

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 2.4 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．

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]
```

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 2.4: ベクタテーブル

```
0x0000001c: sub     lr, lr, #4
         stmdb   sp!, {r0-r3}
         bl      fiq_isr
         ldmdb   sp!, {r0-r3}
         movs    pc, lr
```

2.3 Memory Management Unit

2.3.1 ページテーブル

ARM の MMU は 2 段のページテーブルで構成される．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 Section と呼ばれる 1MB のページへの PTE(Page Table Entry) をもつ．L1 ページテーブルは 4GB の仮想アドレス空間を 1MB のセクションに分割するため，4096 個の PTE をもつ.

表 2.5 に MMU のサポートするページテーブルを示す.

ページ名	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 2.5: 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 種類のエントリによって構成される．エントリの構成を図 2.6 に示す．MMU は PTE の下位 2 ビットによって PTE のタイプを判別する．

Section PTE

Section と呼ばれる 1MB のページのアドレスの先頭 12 ビットを含む．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．

Fine PTE

1024 個のエントリをもつ L2 ページテーブルのアドレスを含む．L2 ページテーブルは 4KB のアラインメントに乗っていけなければならない．

Coarse PTE

256 個のエントリをもつ L2 ページテーブルのアドレスを含む．L2 ページテーブルは 1KB のアラインメントに乗っていけなければならない．

Fault PTE

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．

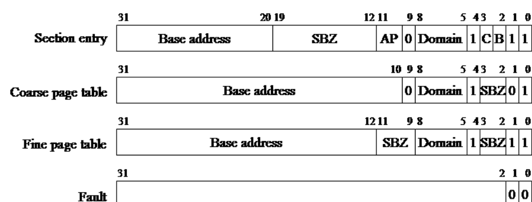

図 2.6: L1 page table entries

L1 ページテーブルのメモリ内での位置は CP15[2]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．CP15 のレジスタ 2 のフォーマットを図 2.7 に示す．

L2 ページテーブルは以下の 4 種類のエントリによって構成される．エントリの構成を図 2.8 に示す．MMU は PTE の下位 2 ビットによって PTE のタイプを判別する．

[2]コプロセッサ

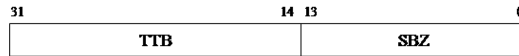

図 2.7: TTB

Large PTE

64KB のページの開始アドレスの先頭 16 ビットを含む．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

4KB のページの開始アドレスの先頭 20 ビットを含む．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

1KB のページの開始アドレスの先頭 22 ビットを含む．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

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．

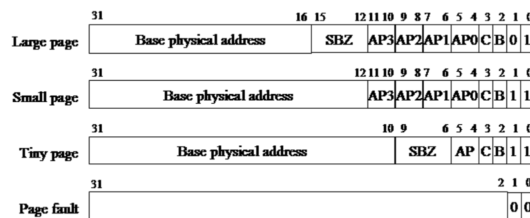

図 2.8: L2 page table entries

2.3.2 Translation Lookaside Buffer

TLB(Translation Lookaside Buffer) は最近使われた PTE を保存するためのキャッシュである．ARM には TLB を操作するコマンドは 2 種類あり，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．

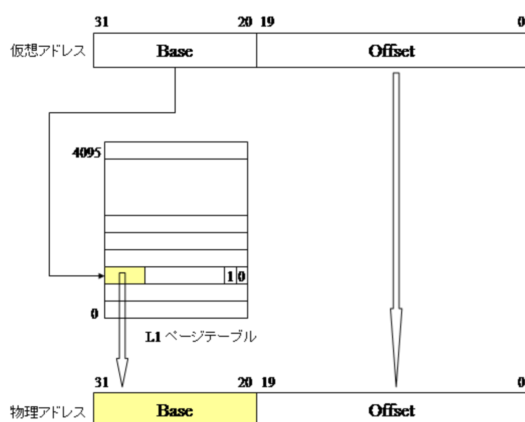

図 2.9: L1 page table walk

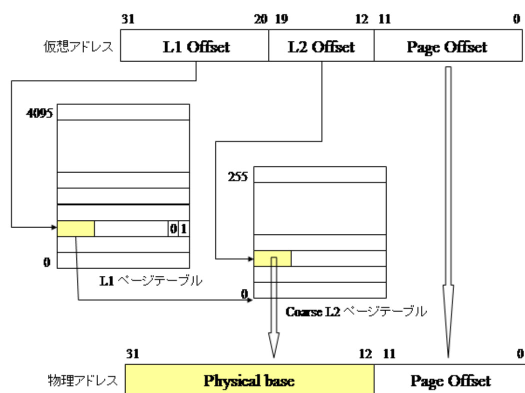

図 2.10: L2 page table walk

MMU が 1MB の Section ページを走査する場合 (図 2.9)，エントリは Master L1 ページテーブルの中にあるので，ページテーブルの走査は 1 段だけです

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 Section 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 2.10) 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 L2 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．

第3章　開発環境

3.1　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．

3.2　CerfCube

3.2.1　CerfCube

CerfCube(図 3.1) 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．

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

図 3.1: CerfCube

CPU	Intel StrongARM SA1110 192MHz
メモリ	Intel Strata フラッシュメモリ (16-bit データバス)
SDRAM	32MB SDRAM (32-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 (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

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 MINIX

3.3.1 MINIX の概要

MINIX は Andrew S. Tanembaum 氏によって開発された UNIX ver.7 と互換性のある OS である．規模が小さいため，mini-UNIX の意味から MINIX と名づけられた．1987 年にバージョン 1.1 が発表され，現在の最新版はバージョン 2.0 である．UNIX と同様に C 言語で書かれているため，Atari，Amiga，Macintosh，SPARC などの各種コンピュータに移植されてきた．

3.3.2 MINIX の構成

MINIX はモジュール化された構造をしている．カーネルが複数のモジュールに分割されていない単一プログラムの UNIX と異なり，MINIX はメッセージパッシングというプロセス間通信を使用して，ユーザプロセス間で通信を行うプロセスの集まりから構成される．モジュール化さているため，カーネルを再コンパイルせずにファイルシステムやメモリマネージャを交換することができるという特徴がある．

MINIX は図 3.2 に示すように 4 つのレイヤで構成される．各レイヤは以下のような役割を担っている．

レイヤ 1 割り込み処理やプロセスのスケジューリングを行い，メッセージを使用して通信するような独立した逐次プロセスのモデルを上位レイヤに提供している．下位レベルの割り込み処理やメッセージのコピーを扱っている部分はアセンブリ言語で記述されているが，それ以外の部分は全て C 言語で記述されている．主にプロセス管理を行うレイヤである．

レイヤ 2 入出力プロセスが各デバイスタイプごとに含まれ，これらのプロセスをタスクと呼ぶ．多くのシステムでは入出力タスクをデバイスドラ

図 3.2: MINIX の構成

イバと呼んでいる．MINIX において “タスク” と “デバイスドライバ” は同義である．レイヤ 2 の全てのタスクとレイヤ 1 の全てのコードは，カーネルという 1 つのバイナリプログラムにまとめてリンクされる．

レイヤ 3 ユーザプロセスに便利なサービスを提供する 2 つのプロセスが含まれる．メモリマネージャは fork，exec，brk などのメモリ管理に関連するシステムコールを発行する．ファイルシステムは read，write，chdir などのファイルに関連するシステムコールを発行する．

レイヤ 4 シェル，エディタ，コンパイラなどのユーザプログラムが動作する．

第4章　設計・実装

4.1　移植の手順

MINIX のソースコードは X86 向けに書かれている．ARM のバイナリを生成しなければならないので，まずクロスコンパイル環境の構築を行った．次に，コンパイラが異なるため Makefile のコンパイラ依存部分の修正，および ARM 上で動作する printk を実装しデバッグを行えるようにした．そして，以下の節で述べるようなアーキテクチャ依存部分の変更を行った．4.2 節でブートシーケンス，4.3 節でプロセス・タスクの特権レベル，4.4 節でシステムコール，4.5 節で割り込み，4.6 節でメモリ保護，4.7 節でデバイスドライバについて実装・設計の変更点の詳細を述べる

4.2　ブートシーケンス

まず x86-MINIX のブートの流れを解説する．(図 4.1 参照) システムを起動するとハードウェアはディスクの最初のセクタを読み込み，そこにあるコードを実行する．そのセクタには Boot と呼ばれるプログラムが配置される．Boot は MINIX イメージをメモリにロードする作業を行う．MINIX をコンパイルすると Kernel，MM(Memory Manager)，FS(File System)，INIT を異なるバイナリとして生成する．MINIX イメージは Kernel，MM，FS，INIT のバイナリを連結させたものである．各バイナリには a.out 形式のヘッダがついているので，ロードが完了するとヘッダをもとに BSS 領域の初期化やスタックの設定を行い，カーネルのエントリポイントに入る．

ARM-MINIX では各バイナリを 1 つのイメージファイルにしない．ホストマシンから CERFCUBE にイメージを転送する際に，I-Boot の機能によって各バイナリを指定したアドレスに配置することが可能なためである．図 4.2 に示すように Boot を 0xc0008000，Kernel を 0xc0100000，MM を 0xc0200000，FS を 0xc0300000，INIT を 0xc0400000 に転送する．メモリにこれらのバイナリを転送しロードが完了すると，まず 0xc0008000 にある Boot が実行される．Boot のヘッダはホストマシンで削除する．Boot は Kernel，MM，FS，INIT のヘッダを参照して，それぞれのデータ領域を 4K アラインにのせる．4K アラインに乗せるのは，テキスト領域とデータ領域を異なるページにのせてメモリ保護を行うようにするためである．この時，ヘッダを読み込む部分

などで大幅な変更を加えた．x86-MINIX のバイナリのヘッダは a.out 形式であるのに対して，ARM-MINIX の各バイナリのヘッダは a.out 形式ではなく，ELF 形式として出力されるためである．データ領域の移動が終わると，BSS 領域の初期化やスタックの設定を行い，Kernel のエントリポイントにブランチし，ブートシーケンスを終える．

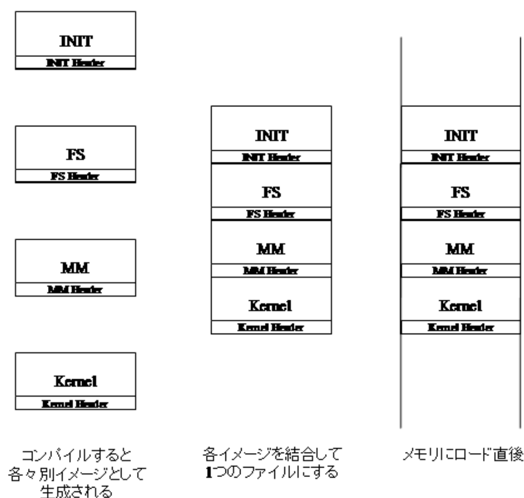

図 4.1: x86-MINIX のブート

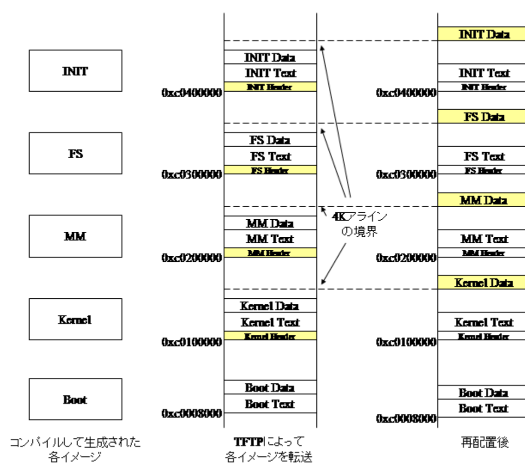

図 4.2: ARM-MINIX のブート

4.3 プロセス・タスクの特権レベル

x86-MINIX ではカーネルは特権レベル 0, タスクは特権レベル 1, サーバとユーザプロセスは特権レベル 3 で動作する．

ARM-MINIX ではカーネルは Supervisor モード，タスクは System モード，サーバとユーザプロセスは共に User モードで動作する．System モードは特権モードであり，なおかつ User モードと全てのレジスタを共有する．このためシステムコールや割り込みの際，レジスタの退避・復元に同じ手続きを用いることが出来る．

各レイヤと動作モード・特権レベルの対応を表 4.1 に示す.

レイヤ	x86-MINIX	ARM-MINIX
レイヤ 1(カーネル)	特権レベル 0	Supervisor モード
レイヤ 2(タスク)	特権レベル 1	System モード
レイヤ 3(サーバ)	特権レベル 2	User モード
レイヤ 4(ユーザプロセス)	特権レベル 3	User モード

表 4.1: レイヤと特権レベル・動作モード

4.4 システムコール

システムコール send, receive, sendrec が呼び出されると，プロセスの処理はトラップ命令によって中断され，カーネルに制御が移る．カーネルはレジスタをプロセスのスタックフレームへ退避してからシステムコールの処理を行う．システムコールの処理が終了すると，レジスタを復元してプロセスを再開する．

カーネルはプロセスの情報を管理するためにプロセス構造体の配列，プロセステーブルを持つ．プロセスのレジスタはプロセスに対応したプロセス構造体のスタックフレームという領域に退避される．レジスタの構成はアーキテクチャごとに異なるため，スタックフレームを修正した.

4.5 割り込み

4.5.1 割り込みの初期化

x86 アーキテクチャは割り込みを有効にする前に割り込みディスクリプタテーブル (IDT[1]) を設定する必要がある．x86-MINIX はブートシーケンスで IDT の初期化を行う.

[1] Interrupt Descriptor Table

ARM アーキテクチャでは割り込みを有効にする前に，割り込みベクタテーブルを適切なアドレスに配置する必要がある．ベクタテーブルはアドレス 0x00000000 もしくは 0xffff0000 に配置されなければならない．開発環境の CerfCube ではこれらのアドレスに RAM が存在しないため，MMU の設定によって RAM を 0xffff0000 へマッピングしてからベクタテーブルを転送する．ベクタテーブルの転送を図 4.3 に示す．

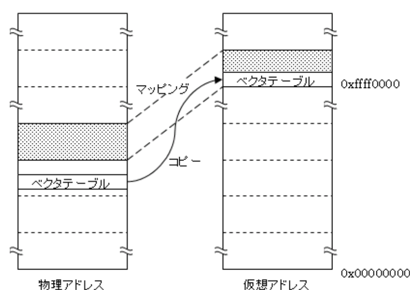

図 4.3: ベクタテーブルの転送

4.5.2 レジスタの退避・復元

割り込みが発生するとカーネルはレジスタを退避してから割り込みハンドラを呼び出す．ハンドラが終了するとレジスタを復元して直前の処理を続行する．

割り込みはプロセスを実行中であってもカーネルを実行中であっても発生する．ARM-MINIX はカーネルとプロセスを異なるモードで実行するため，割り込みが発生した際のモードによって退避すべきレジスタが異なる．ARM-MINIX は直前の処理がカーネルかプロセスかによって，割り込みハンドラが異なる処理を行う．

4.6 メモリ保護

x86-MINIX はセグメント機構を用いてメモリ保護を行っている．セグメント機構はメモリ空間を可変長に区切りメモリ保護を行うことができるのが特徴である．図 4.4 で示すようにディスクリプタテーブルというディスクリプタを格納するためのテーブルを用意する．ディスクリプタはベースアドレス，オフセットアドレス，属性によって構成される．ディスクリプタテーブルにはセレクタ値という識別子がついているので，これをレジスタに設定するとそのディスクリプタが指し示すメモリへのアクセスが有効になる．ディスク

リプタが指し示すアドレス以外へのアクセスや属性に違反するアクセスは禁止される．ディスクリプタはテキスト，データ，スタックの各領域ごとにあるので，他の領域のメモリを破壊することはない．

一方 ARM にはセグメント機構がないので，ページング機構を用いてメモリ保護を行うように変更を行った．各プロセスは仮想アドレスの 0 番地から開始するようにし，コンテキストスイッチの際は L1 ページテーブルの PTE を変更することでメモリ空間の切り替えを行うようにした．各プロセスは独自の仮想メモリを持つことになるので，あるプロセスが動作中に他のプロセスのメモリを破壊することはない．また，PTE のアクセスパーミッションを設定することで不正なアクセスも防いでいる．MINIX は移植性を高めるために仮想記憶を実現していないので，ページフォルト時のスワップイン・スワップアウトは実装していない．

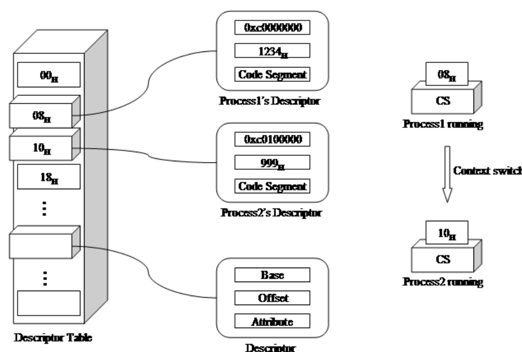

図 4.4: x86 でのメモリ保護

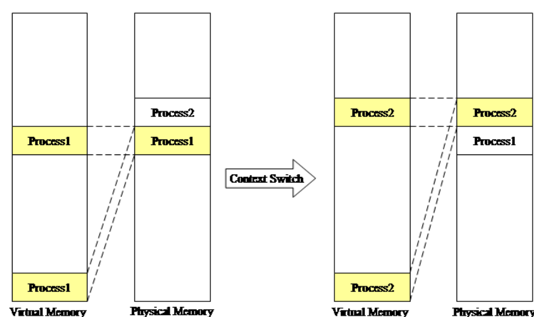

図 4.5: ARM でのメモリ保護

4.7 デバイスドライバ

4.7.1 TTY タスク

TTY(Tele-TYpewriter) とはキャラクタ端末のことで，キーボードやプリンタから構成される．x86-MINIX は TTY のデバイスドライバを割り込みによって実装している．ARM-MINIX では実装を容易にするためにポーリングによる実装へ変更を行った．

4.7.2 メモリタスク

メモリタスクは RAM ディスクの機能を提供する．実験環境ではまず I-Boot の TFTP 機能によってホストマシンから RAM ディスクイメージを CerfCube の RAM 上に転送する．ARM-MINIX は RAM 上のデータをルートディスクとしてマウントする．

4.7.3 クロックタスク

クロックタスクはタイマ割り込みを処理し，時間に関する POSIX API を扱う．

開発環境の CerfCube は OS Timer Counter Register(以下 OSCR) の値が OS Timer Match Register(以下 OSMR) の値と一致したときに，割り込みが発生する．繰り返し割り込みを発生させるために，クロックハンドラは割り込みの度に OSMR の値をインクリメントする．

第5章　評価・考察

今回はARM-MINIXにおいて，プロセス管理に関する重要なシステムコールであるfork，execの評価を行った．測定を行ったCerfCubeの仕様は表3.1に示した通りである．以下の節で，評価結果および評価に対する考察を述べる．

5.1　評価方法

時間の計測を行うために，TICK_TIMEマクロおよびinitticks, dumpticksシステムコールの追加を行った．

TICK_TIMEは次のように定義される．

```
#define TICK_TIME asm("mcr p0, 0, r0, c0, c0": : : "r0")
```

mcrはARM命令セットのコプロセッサ-レジスタ間転送命令である．ARMプロセッサは存在しないコプロセッサへのアクセスを要求されるとUNDEF例外を発行する．ARM-MINIXはUNDEF例外が発行されるとOSCRの値を配列tick timeに保存してから直前の処理を再開する．TICK_TIMEには約1.4μsecのオーバヘッドが存在する．

initticksシステムコールは配列tick timeを初期化する．dumpticksシステムコールは配列tick timeに記録された値をprintk関数を用いて全て出力する．

5.2　Linuxとの比較

ARM-MINIXのfork，execの評価を行い，同じCERFCUBE上で動作するLinuxとの比較を行った．各々のOSのシェルでlsを実行した際の時間を比較した．評価結果を表5.1に示す．

評価結果より，MINIXはLinuxと比べてfork，execともに30倍近く遅いことが分かった．マイクロカーネルアーキテクチャ方式を採用したMINIXと，様々な最適化が施されたモノリシックカーネル方式のLinuxを比較した当然の結果と言える．以下の節ではMINIXのオーバーヘッドとなっている箇所の評価・考察を行う．

	Linux	MINIX
fork	0.568	18.181
exec	2.660	74.500

表 5.1: Linux と MINIX の比較 (単位：μsec)

5.3 MINIX におけるオーバーヘッド

5.3.1 IPC

マイクロカーネルアーキテクチャ方式を採用した MINIX はプロセス間通信 (IPC) の手段としてメッセージパッシングを用いている．MINIX における IPC(InterProcess Communication) とは，あるプロセスがメッセージを送り，コンテキストスイッチをし，別のプロセスがメッセージを受け取るまでを言う．IPC のオーバヘッドを計測するために，IPC1 回あたりのコストと fork, exec 中に発生する IPC の回数を計測した．

IPC のコストの計測を行ったところ，IPC1 回あたりのコストは 91(μ sec) であった．

次に fork，exec の実行中に発生する IPC の回数を測定した．各コマンドのサイズを表 5.2 に，IPC の発生回数を表 5.3 に示す．表より実行するバイナリのサイズが大きいほど IPC の回数が増えるという結果になっている．IPC によるオーバーヘッドが大きいことが分かる．

	text size	data size
ls	34256	9376
pwd	4976	432
sample	6732	320
sh	35772	4480

表 5.2: 各コマンドのサイズ (単位：byte)

	fork	exec
ls	9	182
pwd	9	104
sample	9	106
sh	9	180

表 5.3: IPC の発生回数 (単位：回)

5.3.2　fork,exec のコストの内訳

MINIX におけるオーバーヘッドを調べるため，ls 実行時の fork，exec のコストの内訳を計測した．計測結果は図 5.1，図 5.2 に示す通りである．fork では約 90%をコピーに，exec では約 70%をコピーやメモリの初期化に使っている．メモリへの書き込みや読み出しに時間がかかっていることが分かる．また、この内訳の表からも IPC がオーバーヘッドになっていることが明らかである．

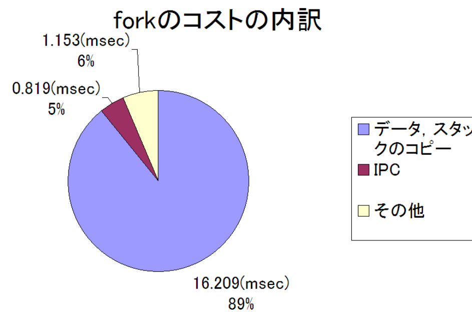

図 5.1: ls 実行時の fork のコストの内訳

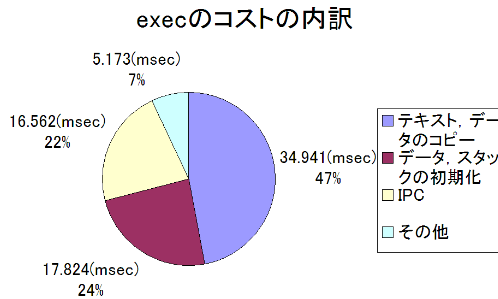

図 5.2: ls 実行時の exec のコストの内訳

5.4　改善案

前節より fork や exec の際のオーバーヘッドは，コピーや IPC であることが分かった．本節ではこれらのオーバヘッドの現状の問題点を述べ、それらに対する改善のための改良案を示す．

5.4.1 コピー

5.3.2 節より，データのコピーがオーバーヘッドになっていることが分かった．現状のコピーの問題点は図 5.3 に示すように、データを 1byte ずつコピーしている点である．これを図 5.4 に示すように，コピーの最初と最後を 4byte アラインメントのために 1byte コピーを行い，間の部分を 4byte コピーにすると高速になる．

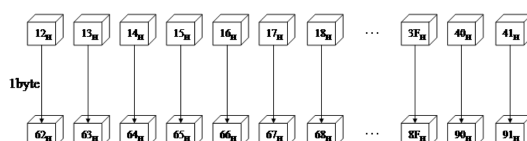

図 5.3: 1byte コピー

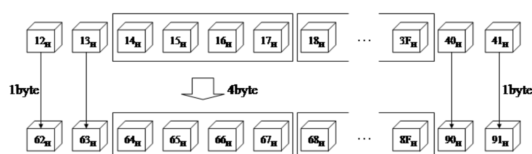

図 5.4: 4byte コピー

5.4.2 IPC

IPC を高速化するには，キャッシュミスを減らすなどのアーキテクチャ依存の改良を施さなければならない．IPC の高速化手法については過去に様々な研究がなされてきた．6 章の関連研究にて詳細に述べる．

第6章　関連研究

マイクロカーネル方式の OS において，システムのサービスを利用するには IPC を使わなければならない．IPC が発生する度に，コンテキストスイッチが起きる．それゆえにマイクロカーネル方式の OS において，コンテキストスイッチの速度は重要である．本章では Strong ARM においてコンテキストスイッチを高速化する手法として，’Fast Address-Space Switching’ および’TLB の共有’ を紹介する．これらの手法は L4 などのマイクロカーネル方式の OS で有効である．

6.1　Fast Address-Space Switching

コンテキストスイッチが発生するたびに，CPU のキャッシュや TLB をフラッシュするのはかなりのコストがかかる．そこで ARM 特有のドメインを用いて，アドレス空間の識別を行い，アドレス変換のマッピングの変更がなされるまでキャッシュをフラッシュしないようにする．これによってコンテキストスイッチのオーバーヘッドはかなり改善する．

6.2　TLB の共有

プロセス間で共有されたページのエントリを共有するというものである．この手法はコンテキストスイッチの高速化のもっとも一般的な手法である．しかし Strong ARM においてこの手法はあまり有効的でなく，ある特定の条件でのみ威力を発揮する．

第7章　結論

本研究では，MINIX の ARM への移植を行った．そして，プロセス管理に関する評価をとるために，ARM-MINIX における fork，exec のコストの計測を行った．ARM-MINIX の性能はよくないことが分かったが，オーバーヘッドとなっている箇所について議論を行い，問題点に対する改善案を示すことで，ARM-MINIX もある程度は組み込み機器で使えるという結論を得た．また，OS の動作を理解するという目的は達成された．

謝辞

本研究を始めるきっかを与えてくださった中島達夫教授に心より御礼申し上げます．また，度々の質問に懇切丁寧に答えてくださった追川修一助教授に心より御礼申し上げます．様々なアドバイスをしてくださったグループの先輩である，岩崎匡寿氏，菅谷みどり氏，茂田井寛隆氏に感謝します．そして，共同に研究を行った杵渕雄樹氏をはじめとする同輩に感謝します．

参考文献

[1] MINIX : http://www.cs.vu.nl/ ast/minix.html

[2] Andrew N.SLOSS, Dominic SYMES, Chris WRIGHT : “ARM System Devloper’s Guide”, MORGAN KAUFMANN(2004)

[3] A・S・タネンバウム, A・S・ウッドハル : “オペレーティングシステム 第 2 版　設計と理論および MINIX による実装”, ピアソン・エデュケーション (2003)

[4] John R.Levine : “Linkers & Loaders”, オーム社 (2001)

[5] 蒲地輝尚 : “はじめて読む 486”, アスキー出版局 (2002)

[6] Adam Wiggins, Harvey Tuch, Volkmar Uhlig and Gernot Heiser : “Implementation of Fast Address-Space Switching and TLB Sharing on the StrongARM Processor”

[7] Volkmar Uhlig, Uwe Dannowski, Espen Skoglund, Andreas Haeberlen : “Performance of Address-Space Multiplexing on the Pentium”

[8] Intel Corporation : “Intel StrongARM SA-1110 Microprocessor Developer’s Manual”