2004年度 卒業論文

組み込み向け汎用OSにおける プロセス間通信の評価

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 : 1G01P033-1
氏名 : 杵渕 雄樹

目 次

図 目 次

表 目 次

概 要

近年，組み込み機器に求められるアプリケーションの高性能化に伴い，これまで PC などで用いられてきた Linux や FreeBSD などの汎用 OS が組み込み機器においても使用されるようになった．これらの汎用 OS はプロセスをそれぞれ保護されたメモリ領域で動作させることで，これまでの組み込み OS よりも高い堅牢性を実現している．しかしそれによってプロセスのコンテキストスイッチやプロセス間通信のコストが増大する．本研究では，組み込み向けの汎用 OS におけるプロセス間通信の性能改善を行うために，まず実験環境として汎用 OS である MINIX を ARM アーキテクチャへ移植した．そして MINIX におけるプロセス間通信のひとつであるパイプの解析，および測定を行い，特にオーバヘッドとなっている箇所を修正することでパイプの処理時間を短縮した．

Abstract

In recent years, due to the requirement of more sophisticated applications on embedded systems, general-purpose operating systems which have been used on desktop PC's, such as Linux and FreeBSD, are ported to embedded systems. These operating systems support separated memory region, which improves a security of a system, though it decreases the speed of context switch and increases the overhead of interprocess communications. In our research, first we ported MINIX , one of a general-purpose operating system which is fairly small OS, to ARM architecture to measure the processing time of interprocess communication pipe. From the result, we found out the unnecessary initialization of block chache and removed it.

第1章　序論

1.1　背景

近年，携帯電話や PDA などの組み込み機器はますます高機能化，多機能化の傾向にある．例えば無線 LAN や Bluetooth などのネットワーク機能や動画・音声再生などのマルチメディア処理が求められるようになってきた．また携帯電話などにみられるように，外部からアプリケーションをダウンロードして使用可能なシステムの登場により，外部アプリケーションからシステムを保護する仕組みが求められている．

このような要求から，これまで PC などで用いられてきた Linux や FreeBSD などの汎用 OS が組み込み機器へ移植されるようになった．これらの OS を組み込み機器に搭載することで，OS によるネットワーク機能・マルチメディア処理・高度な GUI の提供，これまで PC などで用いられてきた高機能なアプリケーションの組み込み機器での使用，アプリケーションをそれぞれを独立したメモリ保護領域で動作させることによる信頼性・堅牢性の向上などが期待されている．

しかし，このような保護機能を搭載した OS で複数のプログラムを動作させる場合，これまで単一のメモリ保護領域で複数のプログラムを動作させていたシステムに比べ，プロセス間通信のコストが増大することが予想される．

また，これら汎用 OS のソースコードは数 100 万ラインにおよび，OS の構造の理解の大きな妨げとなることが予想される．

1.2　目的

このような背景から本研究ではまず OS 全体の構造を理解するために，比較的小規模な OS である MINIX を ARM アーキテクチャへ移植した．次に移植した MINIX[1] において，プロセス間通信であるパイプの処理時間を測定を行い，その結果に対してパイプ処理の高速化手法を提案する．

1.3　論文の構成

まず第 2 章で本研究の背景技術として，MINIX, ARM アーキテクチャ， 開発環境としてターゲットマシンの CerfCube について述べる．次に第 3 章で MINIX を ARM へ移植する際に修正した箇所について述べる．第 4 章では測定を行うにあたって ARM-MINIX へ追加した機能の解説と pipe の測定結果を提示し，それについての考察を行う．そして第 5 章で本研究の結論を述べる．最後に第**??**で将来課題とパイプの高速化に関連する研究を紹介する．

第2章　背景技術と開発環境

本章では，まず本研究において移植を行った MINIX について述べる．次に移植のターゲットである ARM アーキテクチャおよびターゲットマシンの CerfCube について述べる．

2.1　MINIX

2.1.1　概要

MINIX は Andrew S. Tanenbaum 氏によって開発された UNIX V7 互換[1]の OS である．マイクロカーネル方式を採用したモジュール化された構造になっており，4 つのレイヤから構成される．また，OS の学習を目的として可読性に重点をおいて設計や，数千行のコメントがソースコードに埋め込まれているなど，可読性に優れた OS である．既に Atari, Amiga, Macintosh, SPARC などのアーキテクチャに移植されている[2]．

2.1.2　構造

MINIX は 4 つのレイヤで構成される．4 つのレイヤを図 2.1 にしめす．

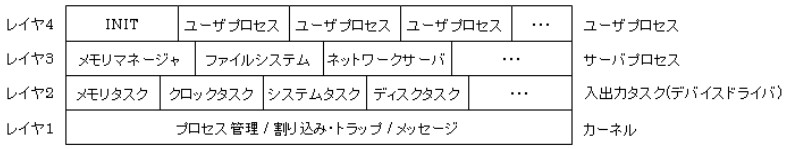

図 2.1: MINIX の構造

レイヤ 1 はプロセスの管理・スケジューリングを行い，割り込みとトラップを捕らえ，プロセス間のメッセージを受け渡す役割を負う．低レベルの割り込みやトラップの処理などはアセンブラで記述されているが，それ以外の部分は C 言語で記述されている．

レイヤ 2 はデバイスタイプごとに用意された入出力プロセスの集まりである．これらのプロセスをタスクという．タスクは他のシステムにおけるデバイスドライバにあたる．タスクは必要に応じて追加・削除することができる．

レイヤ 1 とレイヤ 2 は kernel 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．一部のタスクはサブルーチンを共有するが，それ以外のタスクは独立して動作する．

[1]MINIX バージョン 2.0 は POSIX 規格に基づいている
[2]論文執筆現在 MINIX バージョン 2.0 は x86 と SPARC のみで動作

レイヤ 3 はユーザにサービスを提供するいくつかのプロセスから成る．これらのプロセスをサーバという．サーバは独立したメモリ空間で動作し，デバイスや他のプロセスのメモリ空間にアクセスする権限を持たない．POSIX API などのサービスはサーバとのメッセージのやり取りによって提供される．メモリマネージャは fork や exec といった主にメモリ管理に関するシステムコールを処理し，ファイルシステムは chdir や stat などの主にファイルやデバイスに関するシステムコールを処理する．サーバはデバイスを直接操作することができないため，デバイスに関する処理はメッセージを介してタスクに渡される．

レイヤ 4 は init, sh, ls などのプログラムが動作するユーザプロセスのレイヤである．init プロセスは全てのプロセスの基となるプロセスである．ユーザプロセスはメッセージをファイルシステム，メモリマネージャに渡すことでシステムコールのサービスを受ける．

2.1.3 システムコール

各プロセスやタスクはそれ自体の属するレイヤ，および隣り合うレイヤと通信することができる．例えばレイヤ 4 のユーザプロセスはレイヤ 2 のタスクと直接通信することはできない．

MINIX の提供する POSIX API の実体はメッセージパッシングによるプロセス間通信である．ほとんどの POSIX API は最終的に sendrec システムコールを呼び出し，ファイルシステムかメモリマネージャにメッセージを送信する．実際に MINIX カーネルに備わるシステムコールはメッセージパッシングを行う send, receive, sendrec の 3 種類のみであるが，以下システムコールといった場合は POSIX API を含むものとする．

MINIX はメッセージパッシングにランデブ方式を採用している．ランデブ方式では送受信プロセスの双方が送受信状態になるまでブロックする，

2.2 ARM アーキテクチャ

2.2.1 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．以下，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．

2.2.2 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 2.2 の通りである．User モードとは，アプリケーションが実行するときに，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，2.2.3 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．

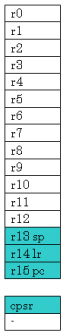

図 2.2: USR モードで使用できるレジスタ

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，2.2.3 にて述べる．

2.2.3 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 **??**に示す．cpsr は，flags, status, extension, control の 4 つのフィールドに分けられる．

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．

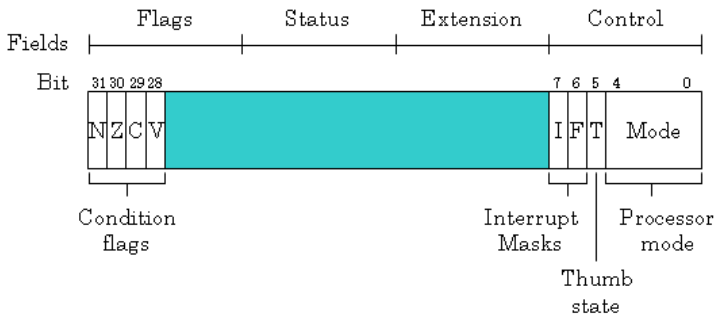

図 2.3: Program Status Register

- 特権モード
  Abort(ABT), Fast Interrupt Request(FIQ), Interrupt Request(IRQ), Supervisor(SVC), System(SYS), Undefined(UND) 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(USR) 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．例外処理についてはで述べる．

バンクレジスタ

図 2.4 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．

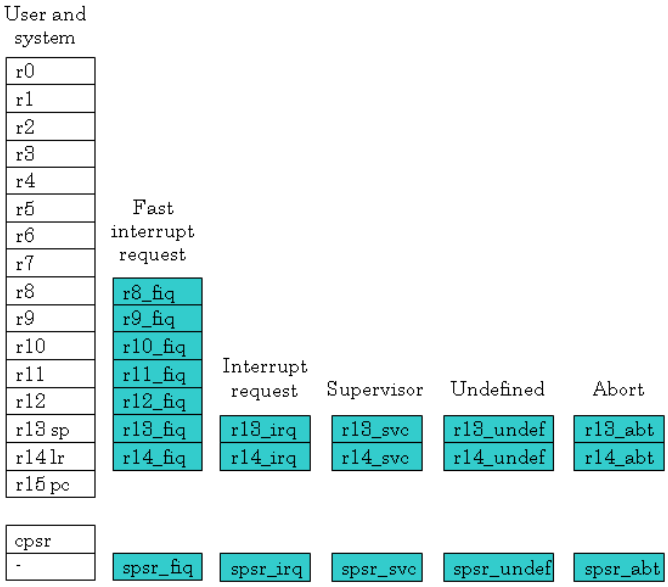

図 2.4: ARM の全レジスタ

USR モードを除いた全てのプロセッサモードでは，cpsr のモード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，SYS モードを除いた全てのプロセッサモードでは，USR 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 USR モードから SVC 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 2.3 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

2.2.4　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 2.5 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．

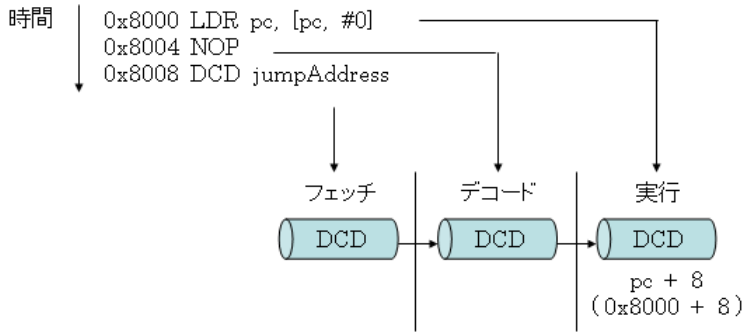

図 2.5: 3 パイプライン実行例

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．

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

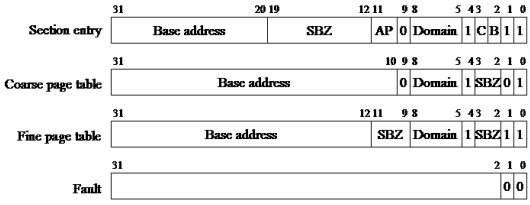

図 2.6: L1 Page Table Entries

2.2.5 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．

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

MMU は PTE の下位 2 ビットによってタイプを判別する (図 2.6 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 2.7 に示す．

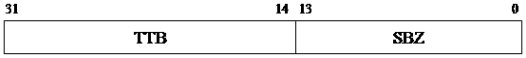

図 2.7: TTB

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 のたページ
Tiny Page	→	1KB のページ
Fault Page	→	アボートを発生させる

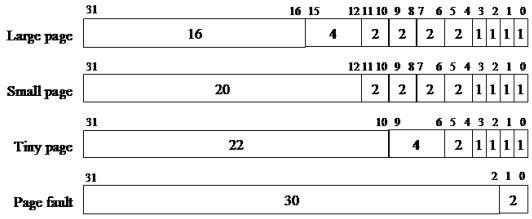

図 2.8: L2 Page Table Entries

MMU は PTE の下位 2 ビットによってタイプを判別する (図 2.8 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．

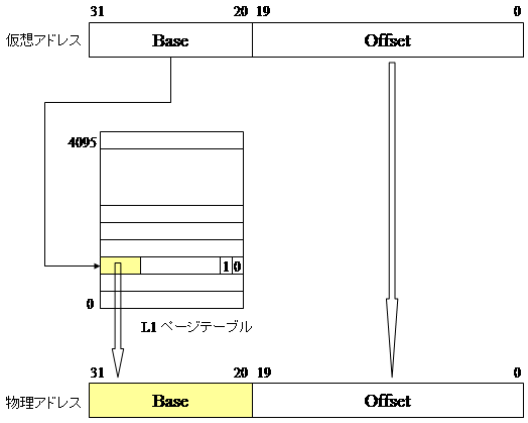

図 2.9: L1 Page Table Walk

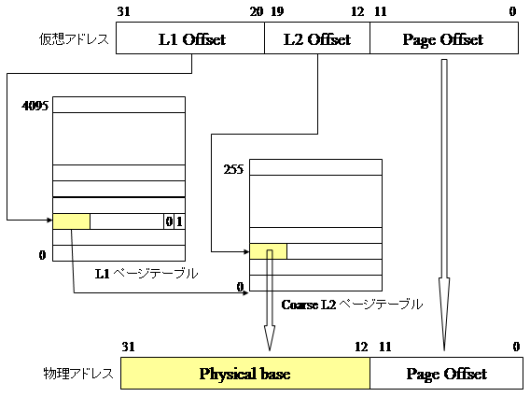

図 2.10: L2 Page Table Walk

MMU が 1MB のセクション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，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．

例外	モード	I ビット	F ビット
RESET	SVC	1	1
DABT	ABT	1	0
FIQ	FIQ	1	1
IRQ	IRQ	1	0
PABT	ABT	1	0
SWI	SVC	1	0
UNDEF	UND	1	0

表 2.4: 例外発生時のモード，ビット

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.

2.2.6 例外処理

ARM プロセッサは Reset(RESET), Data Abort(DABT), Fast Interrupt Request(FIQ), Interrupt Request(IRQ), Prefetch Abort(PABT), Software Interrupt(SWI), Undefined Instruction(UNDEF) の 7 種類の例外を持つ.

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 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，モードビットを設定
4. pc の値を特定のアドレスに設定

各例外発生時のモード，フラグの関係を表 2.4 に示す.

また，例外には優先度が設定されており，より優先度の高い例外が発生している際は低い優先度の例外は発生しない．例外の優先度を表 2.5 に示す.

RESET 例外 はシステムの初期化を行う．RESET 例外は例外の中で最も高い優先度をもつ．また RESET 例外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 SVC モードに設定される.

例外	優先度
RESET	1
DABT	2
FIQ	3
IRQ	4
PABT	5
SWI	6
UNDEF	6

表 2.5: 例外の優先度

DABT 例外 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，USR 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 ABT モードに設定される．

FIQ 例外 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 例外は優先度の高い割り込みである．FIQ 例外は DABT 例外の発生していない場合に発生する．IRQ 例外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 例外は発生しない．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．

IRQ 例外 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 例外は 2 番目に優先度の低い割り込みである．IRQ 例外は FIQ 例外か DABT 例外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 例外は発生しない．発生時に I ビットは 1，モードは IRQ モードに設定される．

PABT 例外 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 ABT モードに設定される．

SWI 例外 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 SVC モードに設定される．

UNDEF 例外 は ARM 命令セット，Thumb 命令セット[3]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 UNDEF 例外が発生する．発生時に I ビットは 1，モードは UND モードに設定される．

[3]16bit に圧縮された命令セット

例として USR モードで動作中に IRQ 例外が発生した場合の動作を図 2.11 に示す．USR 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 lr_irq に，cpsr の値が spsr_irq にコピーされる．同時に cpsr の I ビットが 1 に，モードビットに 10010(IRQ) が設定される．

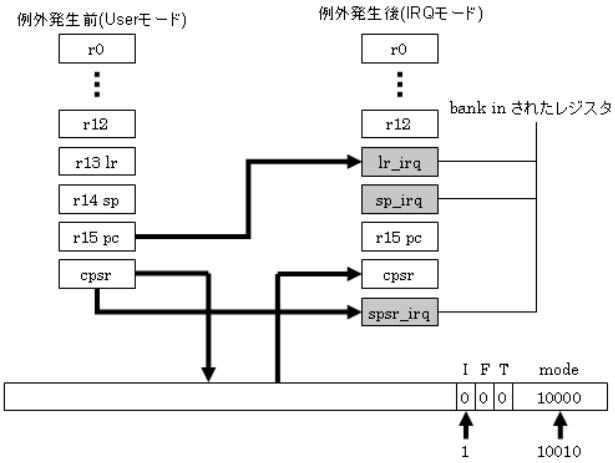

図 2.11: IRQ 発生時の動作

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ジャンプする．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 2.6 に示す．

例外	アドレス	アドレス (高位)
RESET	0x00000000	0xffff0000
UNDEF	0x00000004	0xffff0004
SWI	0x00000008	0xffff0008
PABT	0x0000000c	0xffff000c
DABT	0x00000010	0xffff0010
未使用	0x00000014	0xffff0014
IRQ	0x00000018	0xffff0018
FIQ	0x0000001c	0xffff001c

表 2.6: ベクタテーブル

通常，ベクタテーブルのエントリには次に示すような分岐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分岐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．FIQ 例外が発生すると，アドレス 0x0000001c から始まるハンドラが即座に実行される．このハンドラは分岐を行わないため，より高速に処理される．

```
0x00000000: ldr     pc, [pc, #reset_handler]
0x00000004: ldr     pc, [pc, #undef_handler]
0x00000008: ldr     pc, [pc, #swi_handler]
0x0000000c: ldr     pc, [pc, #pabt_handler]
0x00000010: ldr     pc, [pc, #dabt_handler]
0x00000014: ldr     pc, [pc, #none_handler]
0x00000018: ldr     pc, [pc, #irq_handler]
0x0000001c: sub     lr, lr, #4
            stmdb   sp!, {r0-r3}
            bl      fiq_isr
            ldmdb   sp!, {r0-r3}
            movs    pc, lr
```

2.3 CerfCube

2.3.1 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，CurfCube 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 (CurfCube の詳細なハードウェア仕様については表 2.7 を，メモリマップについては表 2.8 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 2.7: 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 2.8: 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．

2.3.2 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

第3章　設計と実装

3.1　概要

本章では移植を行った箇所の設計と ARM における実装について述べる．以下，オリジナルの x86 アーキテクチャ向けの MINIX を x86-MINIX, 本研究において ARM アーキテクチャへ移植した MINIX を ARM-MINIX と呼ぶ.

本研究では主に x86-MINIX の次の箇所を修正することで移植を行った.

1. ブートシーケンス
2. プロセス・タスクの特権レベル
3. メモリ保護
4. システムコール
5. 割り込み
6. デバイスドライバ
   (a) メモリタスク
   (b) クロックタスク
   (c) TTY タスク

また，評価に関する項目として,

1. ファイルシステム
2. パイプ

について述べる.

3.2　ブートシーケンス

まず x86-MINIX のブートの流れを解説する．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

を連結させたにすぎない．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 領域の初期化などを行い，カーネルのエントリポイントに入る．

移植を行った 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 TFTP で取得した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 3.2 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ロードするとまず boot が実行される．CerfCube では 0xc0008000 にあるコードから実行されるので，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．これ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．コンパイラが違うため，ARM 上で動作する MINIX の各バイナリのヘッダは ELF 形式として出力される．x86 上で動作する MINIX のバイナリのヘッダは a.out 形式なので，ヘッダを読み込む部分などで大幅な変更を加えた．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ジャンプし，ブートシーケンスを終える．

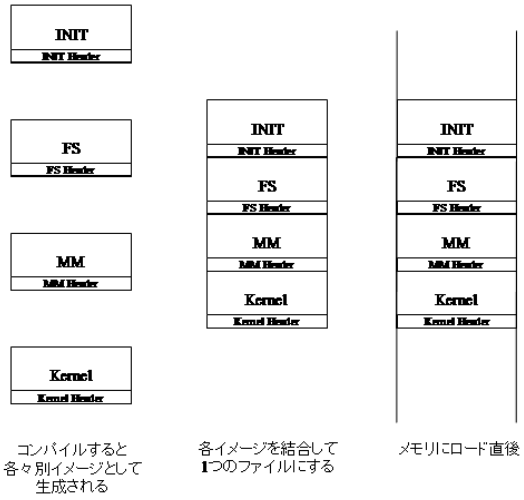

図 3.1: x86 のブート

3.3 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う．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 3.3 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ため，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

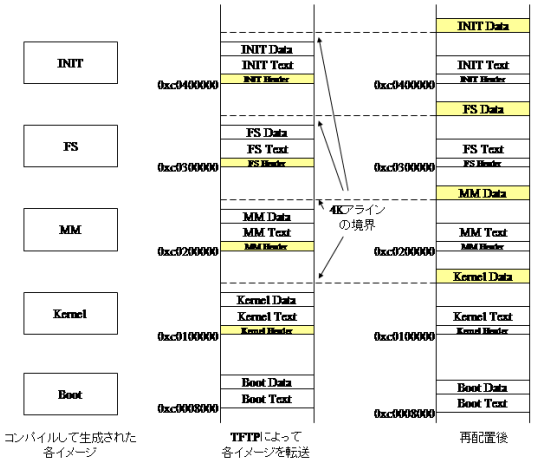

図 3.2: ARM のブート

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用意されているため，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．

ブロックキャッシュ

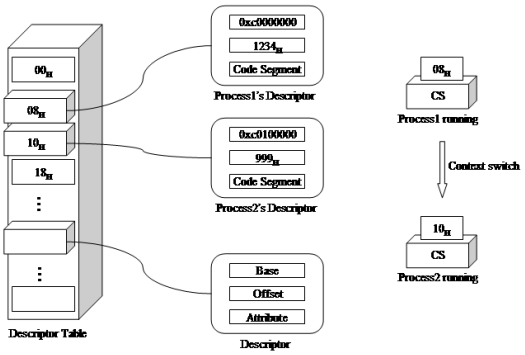

図 3.3: x86 でのメモリ保護

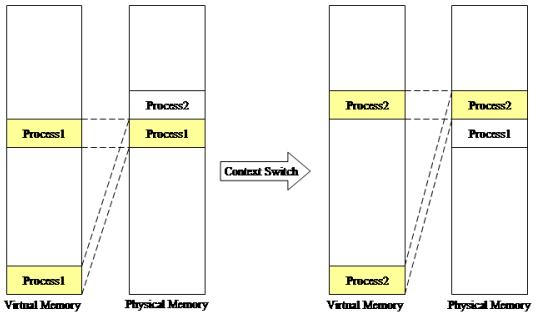

図 3.4: ARM でのメモリ保護

3.4 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 SVC モード，タスクは SYS モード，サーバとユーザプロセスは共に USR モードで動作する．SYS モードは特権モードであり，なおかつ USR 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 3.1 に示す.

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

表 3.1: レイヤと特権レベル・動作モード

3.5 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.

3.6 割り込み

3.6.1 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．

ARM-MINIX はブートシーケンスにおいて割り込みベクタテーブルの初期化を行う．ARM アーキテクチャにおいては割り込み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 3.5 に示す．

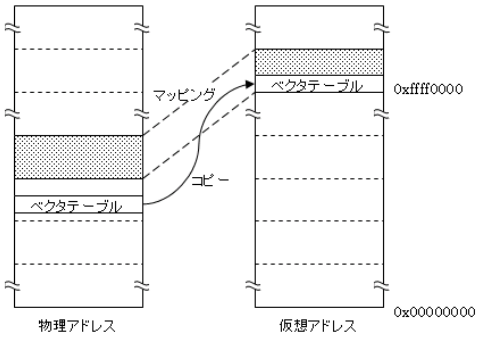

図 3.5: ベクタテーブルの転送

3.6.2 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，割り込み発生時の動作モードによって，割り込みハンドラは異なるレジスタを退避する．

3.7 デバイスドライバ

3.7.1 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．

[1]Interrupt Descriptor Table

3.7.2 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，時間に関するシステムコールを扱う．

開発環境の CerfCube では，OS Timer Counter Register(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．

3.7.3 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 上で動作する 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 へ移植した 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．

3.8 ファイルシステム

3.8.1 ブロックキャッシュ

MINIX のファイルシステム自身は直接デバイスの操作を行わず，実際のデバイス I/O は MINIX レイヤ 2 のデバイスドライバによって行われる．デバイス I/O はメモリアクセスよりも遅いため，ファイルシステムはその内部にデバイス上のデータを一時的に格納するためのブロックキャッシュを持つ (図 3.6).

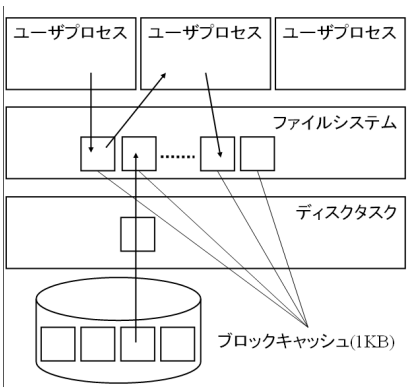

図 3.6: ブロックキャッシュ

3.9 パイプ

パイプはプロセス間通信に用いられる仮想的なデバイスである．本節ではパイプを使用する際に用いられる pipe システムコール, write/read システムコールについて述べる．

3.9.1 pipe システムコール

ユーザプロセスはパイプを使用するために，まず pipe システムコールを呼び出す．ファイルシステムが pipe システムコールのメッセージを受け取ると，メッセージの送信元プロセスにファイルディスクリプタの対を返す．これらのディスクリプタはそれぞれパイプへの書き込み/読み込みに用いられる．ディスクリプタを取得したプロセスとその子プロセス同士はパイプを通してプロセス間通信を行うことができる．

3.9.2 write/read システムコール

パイプへの書き込み/読み込みには write/read システムコールを用いる．write/read システムコールの処理を図 3.7 に示す．

write/read システムコールのメッセージを受け取ったファイルシステムは，まずパイプの使用するブロックキャッシュを探す．パイプにブロックキャッシュが割り当てられていない場合，ファイルシステムは新たなブロックキャッシュを用意する．パイプに書き込まれたデータが増加するに従って，新たなブロックキャッシュが割り当てられる．ファイルシステムは新しいブロックキャッシュに対して書き込みが行われる際に，ブロックキャッシュを 0 で初期化する．

次にファイルシステムはユーザプロセス，ブロックキャッシュ間のデータ転送を行う．第 2 章で述べた通り，ユーザプロセスとサーバは独立したプロセスとして動作しているため，互いのメモリ領域に直接データを書き込む (読み込む) ことはできない．そのため，ファイルシステムはシステムタスクに対して，プロセス間のデータ転送 (sys_copy) を要求する．1KB を超えるデータ転送は 1KB 単位に区切られて実行される．例えば 2.345KB のデータのは 1KB, 1KB, 0.345KB に区切られて 3 回の sys_copy によってユーザ領域からキャッシュブロックへ転送される．

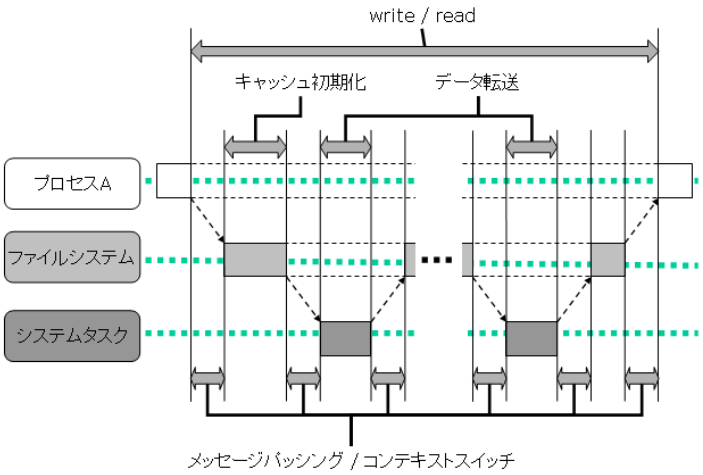

図 3.7: パイプの write/read

3.10 評価のための実装

3.10.1 時間計測

測定を行うために TICK_TIME マクロと initticks, dumpticks システムコールを追加した.

TICK_TIME の次のように定義されている.

```
#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.

第4章　評価と考察

4.1　評価 (1)

まずパイプに対しての write/read システムコールの書き込み/読み込みバイト数に応じた呼び出しから復帰までの時間を計測する．結果を図 4.1 に示す．

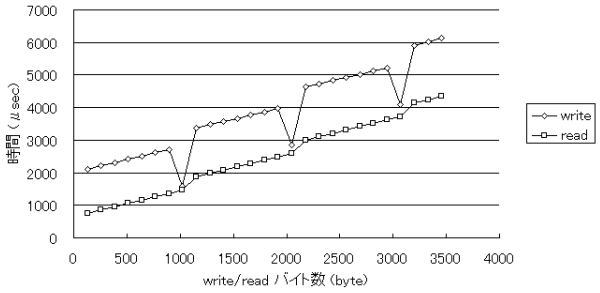

図 4.1: write/read システムコールの処理時間 (1)

転送バイト数が 1024 の倍数のときに write の処理時間が極端に短くなる．転送バイト数が 256, 512, 768, 1024 の際の処理時間の内訳を表 4.1 に示す．

転送バイト量	256	512	768	1024
メッセージパッシング /コンテキストスイッチ	423	423	423	423
キャッシュ初期化	1228	1228	1228	2
データ転送	243	445	648	849
その他	328	328	328	328

表 4.1: write システムコール処理時間 (μsec) の内訳

1024 バイト以外の書き込みにおいては，キャッシュの初期化に最も長い時間がかけられている．データ転送の処理時間は転送バイト数に比例して増加する．メッセージパッシングのオーバヘッドは 1024 バイトまでの転送においては，sys_copy の呼び出しは一度だけのため同じ値になっている．

次にデータ書き込みバイト数が 512 のときに最も処理時間が長いのは，キャッシュの初期化処理である．ブロックキャッシュの初期化に大きな時間が費やされているが，書き込みバイト数が 1024 のときにはブロックキャッシュの初期化が行われていない．

ブロックキャッシュの初期化が行われるのは，通常ファイルを操作するときに lseek システムコールなどを用いて，ファイルの終端を超えた位置にオフセットが設定された場合，それまでのファイル終端とオフセットの空隙が 0 で埋められていなければならないためである．

パイプへの書き込み/読み込みは FIFO(First In First Out) 方式で行われるため，lseek システムコールなどを用いて読み書きオフセット位置を変更することはできない[1]．そのためブロックキャッシュを 0 で初期化する必要はない．そこでパイプが新たなブロックキャッシュへ書き込みを行う場合でも，ブロックキャッシュの初期化を行わないように変更を加えた．

4.2 評価 (2)

変更を加えたファイルシステムで前節同様の測定を行う．結果を図 4.2 に示す．

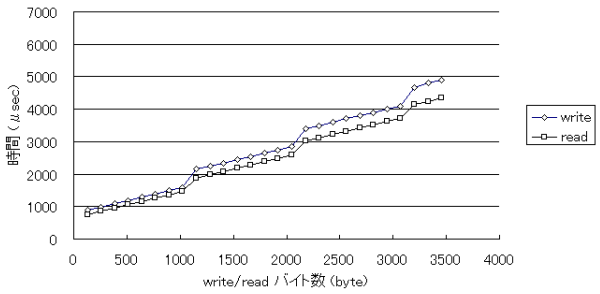

図 4.2: write/read システムコールの処理時間 (2)

4.3 考察

変更を加えた write システムコールで次に大きなオーバヘッドとなるのは，メッセージパッシングによる遅延である．データ転送量が 1KB 増加するごとにシステムタスクとのメッセージのやり取りは増加する．ARM-MINIX のパイプには 7KB までの書き込みを行えるため，1 回のパイプに対する write システムコールのメッセージのやり取りは最大 16 回まで増加する．データ転送量の増加に伴う 1 バイトあたりの時間を図 4.3 に示す．データ転送量が増加するに従い，メッセージパッシングの処理時間の合計は増加するが，1 バイトあたりに要する時間は減少する．

[1]パイプデバイスに対して lseek を行うと ESPIPE エラーが返される

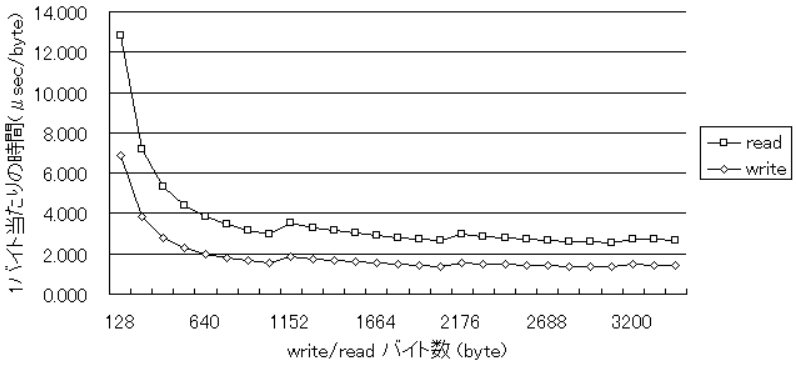

図 4.3: 1 バイトあたりの処理時間

第5章　結論

本研究では ARM アーキテクチャへ MINIX を移植し，パイプへの書き込みの処理時間を測定した．測定結果から ARM-MINIX はパイプデバイスに不要な初期化を行っていることが判明したため，パイプデバイスに対する書き込みを行う際はこの処理を行わないように修正した．その結果，パイプへの write システムコールは約 1220μsec 短縮された．

ARM-MINIX のパイプはデータ転送量を増加させることによってメッセージパッシングの処理時間の合計は増加するが，バイト数あたりの転送に要する時間は短縮されることがわかった．

謝辞

本研究の機会を与えてくださり，常に私の質問に的確なアドバイスを下さった，中島達夫教授，追川修一助教授に厚く御礼申し上げます．また，多くの叱咤激励を頂いた岩崎匡寿氏，お忙しい中で丁寧なアドバイスと多くの書籍をお貸しいただいた菅谷みどり氏，常に親身になって相談に乗って下さった茂田井寛隆氏に深く感謝します．そして，本研究を通じて苦楽を共にした同輩の腰前秀成氏，花岡健介氏，村田雄氏に改めて感謝の意を表します．

関連図書

[1] 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 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, 1998.

[2] 蒲地 輝尚: “はじめて読む 486”, アスキー出版局, 1994.

[3] Andrew N. SLOSS: “ARM System Developer’s Guide”, Elsevier, 2004.

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