第 15 部

オペレーティングシステム

第 1 章

はじめに

オペレーティングシステム (以下 OS) は、インターネットの構成要素である計算機を運用する際に不可欠なソフトウェアコンポーネントである。OS ワーキンググループは、WIDE インターネットの活動において、「インターネットと OS」の関わりあいを考察するためのワーキンググループである。

OS と WIDE プロジェクトのかかわりあいとしてはさまざまなものがあるが、ここでは 3 つの例を示す。第 1 には、実験環境としての OS が挙げられる。IPv6 などの新たなプロトコルスタックの実験や設計、運用を行う際には OS の内部構造に関する正確な知識が必要となる。反対にさまざまなプロトコルスタックの実装をとおして、OS 内部におけるプロトコルスタックのよりよい設計実装方法を提案できる可能性がある。第 2 には、携帯計算機などの計算機のあたらしい利用形態のサポートが挙げられる。WIDE プロジェクトでは携帯計算機とネットワークに関する研究が多く行われているが、これらの研究、特にアプリケーション層などの上位層の研究のためには、携帯計算機をサポートする優れた OS が必要となる。例えば、携帯計算機のための遠隔ファイルシステム、移動ホストのための IP プロトコルなどのリアルな実験環境として、携帯計算機とそれをサポートする OS は非常に重要である。しかしながら、現在携帯計算機のための OS 環境はいまだ脆弱であるため、WIDE プロジェクトでは独自に OS の開発・既存 OS の改良/改造を行っている。第 3 に、OS 自体の新たな構成法とネットワークの関わりあいの考察が挙げられる。OS の構成法は、計算機システムの下にあるネットワークや計算機アーキテクチャと不可分な要素である。ネットワークのない時代の OS 構成法と現在の OS 構成法は当然異なるし、今後ギガビット級のネットワークが整備された際の OS 構成法は現在のものとは異なるであろう。このため、新たな OS の構成法を模索することや、他の研究組織の提案している新たな OS 構成法を学ぶことは WIDE プロジェクトにとって大きな異義がある。

本年度、OS ワーキンググループとして大きくわけて以下のような活動を行った:

- 修論生・卒論生などの活動
- Spring/Apertos 輪講 (95 年 9 月福岡ボード合宿)
- WILDBOAR の活動のまとめ (96 年 3 月春合宿)

福岡ボード合宿においては、先進的な技術に関するサーベイが輪講形式で行われた。輪講は OS とネットワークプロトコルの 2 部行われ、OS に関しては Spring と Apertos の論文輪講が行われた。Spring は Sun Microsystems が開発しているオブジェクト指向オペレーティングシステムの名称であり、これは現在研究機関向けの有償配布が行われている (詳細は `http://www.sun.com/tech/projects/spring/spring.html` 参照)。Apertos は Sony CSL が開発しているオブジェクト指向オペレーティングシステムの名称である。

96 年春合宿においては、WILDBOAR プロジェクトに関してこれまで得られた知見をまとめ、以後研究や設計実装を要する分野についてまとめた。WILDBOAR プロジェクトは、BSDI 社 BSD/OS のための携帯計算機サポートパッケージを開発する活動であり、北陸先端大の篠田助教授、フォア・チューン、WIDE project が共同で開発を行っている。

本章の構成は以下のとおりである。第 2章では、携帯計算機の有効な資源利用のための研究に関して述べる (担当: 杉浦)。第 3章は、Spring オペレーティングシステムの概要についてまとめている。これは、福岡ボード合宿での Spring 輪講のまとめである[1]。第 3.1節では Spring オペレーティングシステムの全体像に関して述べる (担当: 砂原)。第 3.2節では Spring オペレーティングシステムにおける名前づけシステムについてまとめている (担当: 岡本)。第 3.3節では Spring オペレーティングシステムにおける RPC システムについて述べている (担当: 伊藤)。第 4章は 96 年春合宿での WILDBOAR プロジェクトに関する議論をまとめたものである (担当: 石井/楯岡)。

[1] Apertos については昨年度の報告書を参照されたい。

第 2 章

携帯型計算機の有効な資源利用

2.1 はじめに

携帯型計算機は、移動運用を主眼として設計され、ある限定された場所に制限されることなく、いつでもどこでも支援環境を利用することができる。このような移動透過性を維持するために、携帯型計算機は小型化、軽量化によってバッテリで運用され、しかしながら、卓上型計算機との完全互換性を保っている [132]。

携帯型計算機は限定された資源の元で運用される。特に充電池を利用した携帯型計算機を特徴づける運用形式は、利用するオペレーティングシステム及びアプリケーションに対して重大な実装問題となる [133]。

本研究では、電力消費に対して抽象化したオペレーティングシステムに着目し、近年にみられる充電池駆動型の携帯型計算機に柔軟に対応できるシステムについて考える。具体的には、携帯型計算機の電池運用方式について調査を行い、実験的な携帯型計算機の節約機構とオペレーティングシステムと協調システムを作成した。これによって既存のオペレーティングシステムには見られない限定された携帯型計算機の資源の有効利用が可能となる。

本節では、費電力管理機構を既存のオペレーティングシステムで対応させる時に生じる様々な問題点について議論し、その解決策として、携帯型計算機で動作する節約機構に対する抽象化に基づいたオペレーティングシステムの設計、について説明する。携帯型計算機に使用されているデバイス毎の電力消費量を測定し、オペレーティングシステムがデバイス毎にどれだけの電力を消費しているかを理解する。

オペレーティングシステムは、システム内の電池容量を把握し、規定されたデバイスの消費電力によって、システムの動作状態に応じて電力消費を管理する。UNIX オペレーティングシステム及び Mach に対して、これらの実験を行い、評価では、携帯型計算機に対して最大 134%の電力節約が実証され、その有効性が示唆された。

2.2 携帯型計算機の節電機構

現在主流となっている PC と同等の機能を持った携帯型計算機は携帯性を確保するために移動時は充電池を用いて運用される。そのため充電池の畜電容量によって、システムを

表 2.1: 携帯型計算機の機構別消費電力分布

機構	ユニット	消費電力分布 ratio(%)
ディスプレイ		
	液晶ドライバ	10%
	バックライト	24%
ハードディスク		
	Logic Unit	8%
	Disk Unit	19%
マザーボード		39%

利用できる時間は限られる。携帯型計算機は、液晶ディスプレイ、ハードディスク機構、CPU、周辺I/O 機構等によって構成され、いずれの機構も省電力化に向けて技術は発展している [134]。これらの構成要素の中で、もっとも消費電力の多い機構は、液晶ディスプレイのバックライト、ディスプレイインターフェース、そしてハードディスク機構である。本実験で、携帯型計算機 (IBM ThinkPad 230Cs) に対して、デバイス別消費電力分布を調査した。

液晶ディスプレイおよび、バックライトの消費電力は、全体の 34%という結果となった (表 2.1参照)。本実験は、携帯型計算機に見られる各種節電機能を全て無効にした状態で行った。

このような消費電力分布に対し、携帯型計算機は節電機能を各デバイスに用いる事によって消費電力を抑えている [135]。例えば、液晶ディスプレイの場合、バックライトを消灯することにより、24%の節電効果が得られる。また、CPU の動作周波数を下げることで、処理能力もそれに比例して下降するものの、消費電力を下げる事が可能となる。このような節電機能によって、表 2.2のように消費電力を節約する事が可能となる。

このような節電機能は、APM(Advanced Power Management) [136] によって管理される。

2.3 APM の仕様

APM は、Intel, Microsoft によって提供された節電管理機構である。APM は、システムの動作状態における消費電力の制御を行う事を目的としている。デバイスがシステムによって利用されていない場合は積極的にそれらの電力を切断し節電を行う。機種依存性の高かった既存の携帯型計算機の節電機構が APM の仕様によって、統一化され、汎用性の高い節電機能を利用する事ができるようになった。APM は、ハードウェア依存のそれぞれ

表 2.2:　Running Devices in Degraded Mode

機構	ユニット	節電状態	サブレベル	消費電力分布 (%)	通常時
ディスプレイ					
	LCD	6%			10%
	バックライト	0%	Normal	24%	24%
			Level 1	21%	
			level 2	20%	
			Level 3	20%	
ディスク					
	Logic Unit	8%	Logic On		8%
			Sleep	3%	
	Disk Unit	0%			19%
マザーボード					
	High	39%			39%
	Slow	25%			
	Slowest	18%			

のデバイスに対して、ソフトウェアインターフェースによって、それぞれの機能を透過している。APM-BIOS によって、機種依存性の問題は吸収される (図 2.1参照)。

APM には、5 つの電力管理状態が設定される。それらは、(1)Full On、(2)APM Enabled、(3)APM Standby、(4)APM Suspend、(5)Off である。これらの状態は APM コマンドによって、移行する (図 2.2参照)。

2.4　節電機構の抽象化

節電機能を有効に利用するためには、オペレーティングシステムを新しく設計し直す必要がある。本実験では、消費電力に対して抽象化したオペレーティングシステムのプロトタイプを設計、その一部を実験実装することによって、その有効性を確認した。前述した、消費電力分布の調査を基に、UNIX オペレーティングシステムに対して、消費電力に対する抽象化を行い、節電機能が利用できるように変更を行った。

2.4.1　APM

Mach UNIX システムで汎用性の高い節電機能が利用できるように、APM インターフェースを採用した。UNIX オペレーティングシステムに対して APM を介しての節電機能が利用

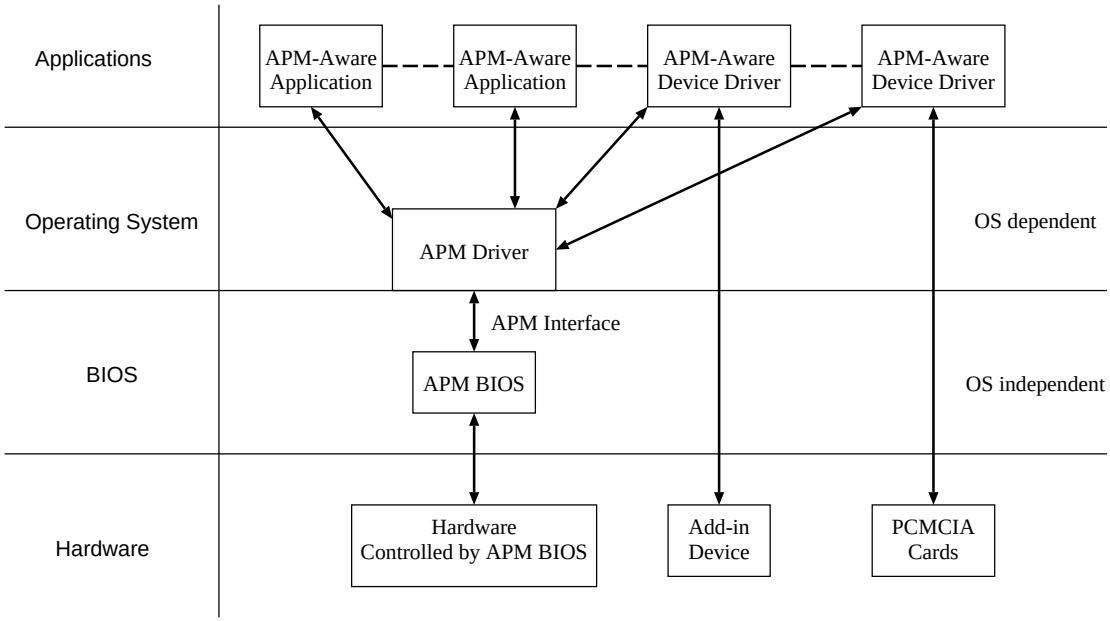

図 2.1: Advanced Power Management System

できるように、システムの変更を行う。APM は、(1)UNIX カーネル自身、(2) アプリケーション、およびユーザ、の両方からアクセスが可能なように設計を行う。具体的には、(1) の解決として、カーネル内の APM I/O インターフェースである *pmcall()* を作成、(2) の解決として、仮想デバイスドライバである**/dev/apm** を作成した。

2.4.2 APM のフック

APM は、UNIX ブート時にフックを行う。システムの電源が入り、セルフテストが終了すると、BIOS Bootstrap ルーチンによって、ディスクからブートコードが読み出され、システムが起動する。第 2 ブートステージで、システムは BIOS コールによって、全てのデバイスの処理、初期化を行い、32 ビット環境であるプロテクトモードに移行し、APM の Probe を行う。APM がブートされる機種に含まれているか、5300H の *APM Installation Check* コールによって検索を行う。*probe_apm()* によって BIOS に APM が搭載されているか否かをチェックする (図 2.3参照)。

2.4.3 APM の I/O インターフェース

APM の I/O インターフェースとして、*pmcall()* ルーチンを作成した。*pmcall()* によって、APM コマンドの入出力が行われ、カーネルでの APM の操作が可能となる。

2.4.4　UNIX の APM デバイスドライバ

APM の入出力をユーザ、アプリケーションからも可能とするために、仮想デバイスドライバである *apm* インターフェースを Mach UX サーバに実装した。UNIX のプロセスおよび、daemon は、**/dev/apm** に対して節電機能のアクセスを行う。APM ドライバは、**apm_open/apm_close** を用いてユーザレベルの利用を可能にしている。図 2.4に apm デバイスドライバを用いた簡単なアプリケーションの例を示す。本アプリケーションで節電機能を無効に設定する。

2.4.5　デバイス毎の節電管理

APM 1.1 では、*Set Power State* によって、システムもしくはデバイスの電力管理状態の変更をデバイス ID によって行う。*Set Power State* によって、定義されているデバイス ID を表 2.3に示す。*Set Power State* を用いることによって、それぞれのデバイスの節電状態を独立に管理することが可能となる。

表 2.3:　デバイスの電力管理 (*Set Power State*)

ID	デバイス
0001H	全てのデバイス
01XXH	ディスプレイ
02XXH	2 次記憶デバイス
03XXH	パラレルポート
04XXH	シリアルポート
05XXH	ネットワーク
06XXH	PCMCIA ソケット
E000H - EFFFH	OEM 予約

実験を行った携帯型計算機によっては、APM によって管理できるデバイスが異なる。APM 1.0 に準拠した携帯型計算機では、このようなデバイス毎の電力管理はできない。

2.4.6　消費電力管理

本システムでは、オペレーティングシステムが、携帯型計算機に搭載している電池の充電容量を把握する。充電容量は **batt_amt** 変数によって保持する。**batt_amt** は、起動時の電池充電容量から算出され、各デバイス毎の電力消費量をもとに、システムの動作状態に置ける消費電力をオペレーティングシステムが管理する。

各デバイスの状況別消費電力は、それぞれの電力管理状態に応じて測定した値として、**pwresource.h** に設定されている。

図 2.5に **batt_amt** の初期化プロセスを示す。

カーネル内で管理される **batt_amt** の電池容量は、UNIX のリスケジューリング時に電力管理状態における各デバイス毎の消費電力を引いていく。それぞれのデバイスの電力管理状態は、apmstat 構造体によって管理される (図 2.6参照)。

batt_amt の値が電池総容量の 10%未満になった場合、カーネル内の電池残量警告が呼び出され、ユーザに対応を求める。

2.4.7 デバイスの節電管理

シリアル、パラレルデバイスは、それぞれのデバイスが open された時のみ電源を供給するように、デバイスドライバに変更を行った。具体的には、*pmcall()* によって、*Set Power State* を呼び出し、デバイス毎の節電管理を行った。

2.5 評価

2.5.1 節電機能の有効性

節電機能が有効に利用できるようなシステム実装がなされたか、完全充電した携帯型計算機を基に完全放電までのシステム動作時間ベンチマークを行った。ベンチマークは、3 つの cron を毎 1、5、15 分おきに実行する。cron は、system カーネルのコンパイル、dhrystone ベンチマーク、fsck コマンドのいずれかを実行する。表 2.4に結果を示す。本実装では 134%の電力節約が実証された。

表 2.4: 節電機能の評価

テスト	実効時間 (minutes)	効果 (%)
節電機能無効	87	100
節電機能有効	110	126
本実装	117	134

表 2.5:　*pmcall()* 実行速度

APM コマンド	**実行速度 (usec)**
APM Driver Version	400
APM Installation Check	600
Get Power Status	700
Get Power State	700
CPU Idle	500
CPU Busy	500

表 2.6: APM コマンド統計

APM コマンド	**総数**	**Percentage (%)**
Get Power State	79968	87
Set Power State	3205	3
CPU Idle	3205	3
CPU Busy	3206	3
Get Power Status	131	—
ETC.	2105	2

2.5.2　batt_amt と実際の電池容量

カーネル内の予測電池容量である **batt_amt**、および、APM の *Get Power Status* によって出力される電池容量との誤差を図 2.7に示す。

2.5.3　オペレーティングシステムオーバーヘッド

APM の I/O インターフェースである pmcall() は、APM BIOS Call を行う。実効される APM コマンドによって、それらのアクセス速度は変化する。表 2.5に本システムで実行する代表的な APM コマンドの実行速度を示す。

携帯型計算機を電池で駆動した場合の APM コマンドの統計を表 2.6 に示す。

2.6 まとめ

本節では、携帯型計算機の特徴のひとつである充電池による運用形態において重要となる電力管理機構、節電機構に対し、オペレーティングシステムからの抽象化を用いて対応を行った。実験のためのプロトタイプの実装によって、134%の節電効果が認められた。オペレーティングシステムは、搭載されている電池の容量と、利用するデバイスの消費電力を認識し、ユーザ、アプリケーションの要求に応じて、システムの電力管理状態を変更することが可能となった。また、実装によっては、オペレーティングシステムが自動的にこれらの機構の管理を行う事も可能となる。このような自らをとりまく環境を情報として受け取るとともに、自ら環境に働きかけ環境を創造していくオペレーティングシステムの重要性が本節では示された。

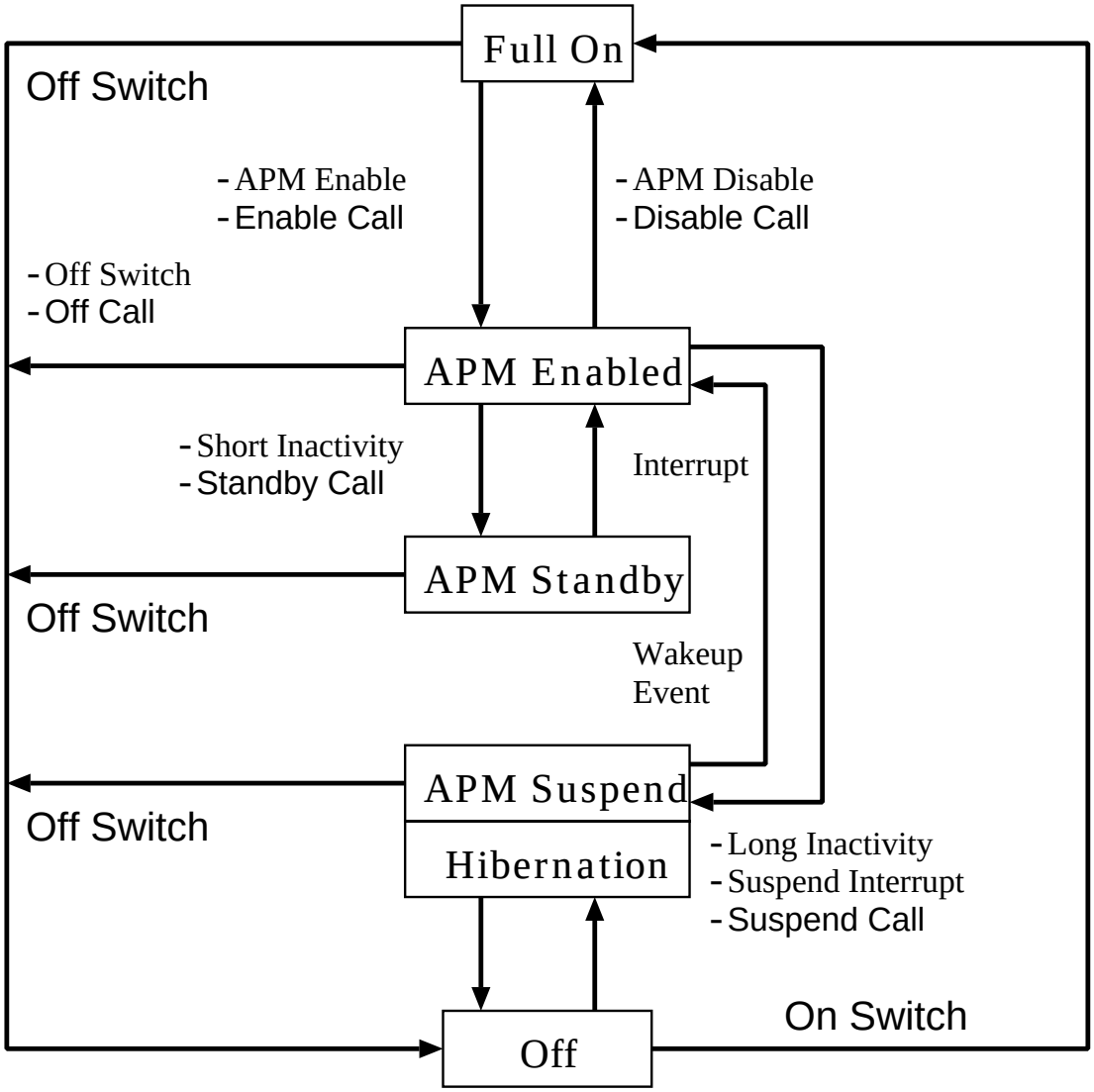

図 2.2: APM System State Transitions

```
void
probe_apm()
{
     struct bios_args ba;
     printf(``Checking APM BIOS...'');
     bzero(&ba, sizeof(ba));
     ba.ba_ax = 0x5300;
       /* APM Installation Check */
     ba.ba_bx = 0;
       /* System Bios */
     bios_call(0x15, &ba);
      if ((ba.ba_flags & PSL_C) != 0) {
         printf(``not found\n'');
         return;
      }
}
```

図 2.3: *probe_apm()*

```
#include <stdio.h>
#include <sys/fcntl.h>
#include <sys/ioctl.h>
#include <machine/apmioctl.h>
main()
{
 int fd;
 int status;
 if ((fd = open(``/dev/apm'', O_RDWR)) < 0) {
   perror(``/dev/apm'');
   fprintf(stderr, ``APM not enabled.\n'');
   exit(1);
  }
 ioctl(fd, PIOCAPMSTOP, 0);
 printf(``APM disabled.\n'');
 close(fd);
 exit(0);
}
```

図 2.4: apm_disable sample programs

```
         :
         :
 pmerr = pmcall(530a, 0, 0, &aret, &bret, &cret);
 /* Get Power Status */
 if (pmerr) {
   panic(‘‘pmerr’’);
 } else {
   astat = (bret >> 8) & 0xff ;
   bstat = bret & 0xff ;
   blife = cret & 0xff ;
   if (astat == 1) {
/* Are you connected to AC? */
      if ( bstat > 3) {
/* Are you sure about the battery amount? */
      batt_amt = batt_amt * blife / 10000 ;
      } else {
      batt_amt = bat_amt*0.5;
/* XXX I can't do anything here...*/
      }
   }
        :
        :
```

図 2.5:　**batt_amt** 初期化プロセス

```
#define          APM_PSTAT_READY         0x0000
#define          APM_PSTAT_STANDBY       0x0001
#define          APM_PSTAT_SUSPEND       0x0002
#define          APM_PSTAT_OFF           0x0003
struct apmstat {
   int display;     /* Display interface */
   int disk;        /* Hard disk drive */
   int par;         /* Parallel ports */
   int ser;         /* Serial ports */
   int network;     /* XXX Network Cards */
   int pcmcia;      /* XXX PCMCIA */
   int oem;         /* XXX OEM related */
};
```

図 2.6:　電力管理状態構造体

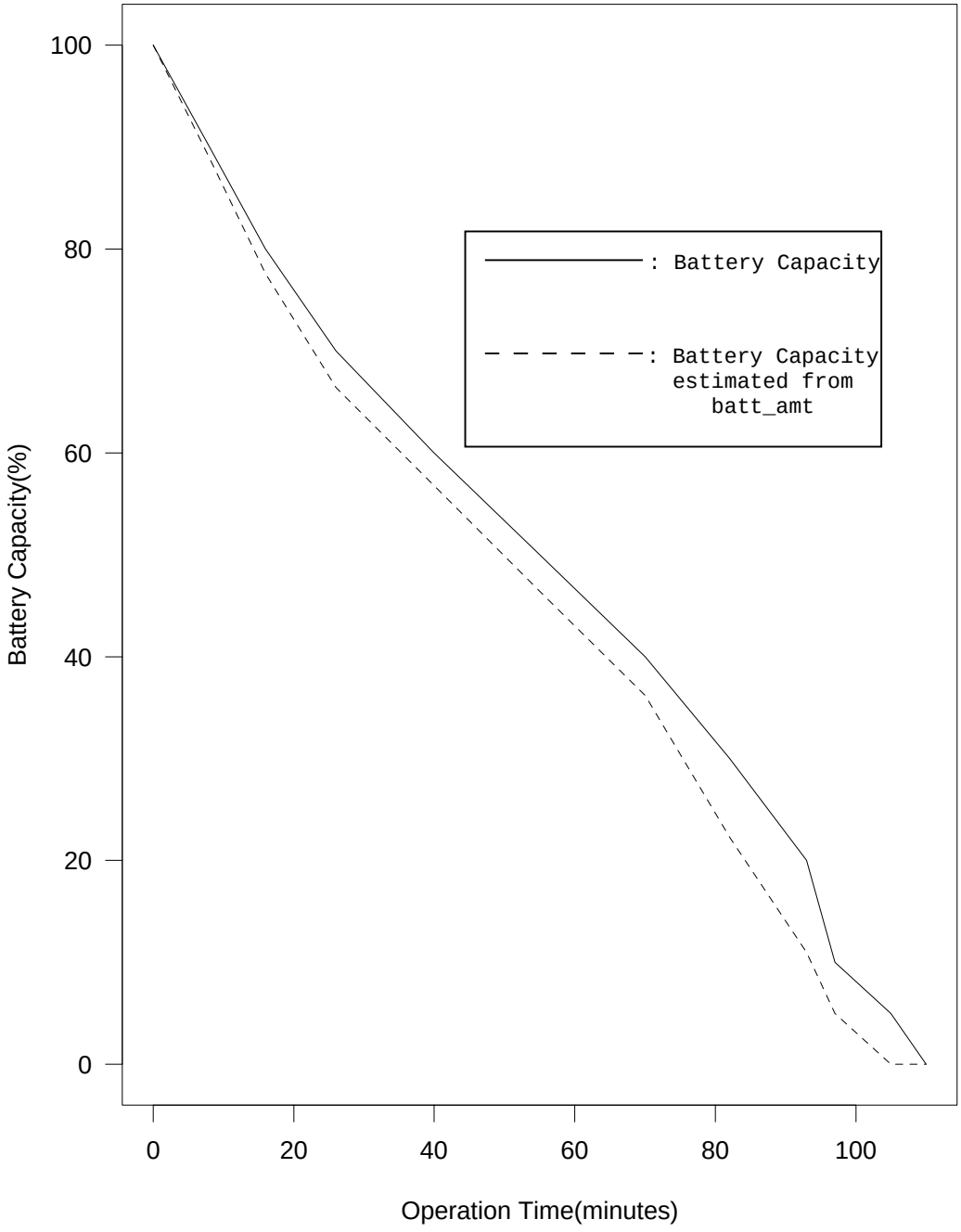

図 2.7: **apm_batt** と実際の電池容量の誤差

第 3 章

Spring オペレーティングシステム

3.1 Spring 概要

Spring オペレーティングシステムは、Sun Microsystems の Mitchell らによって開発された SunOS 互換機能を持つオブジェクト指向オペレーティングシステムである [137]。ここでは、高いモジュール性を持ち分散環境に適したオブジェクト指向オペレーティングシステムを構築することを目標としている。また、従来の SunOS の資産をスムースに引き継ぐため、SunOS が提供する API を持ち、SunOS で動作するアプリケーションを、そのまま Spring 上で利用することができるようになっている。Spring では、Nucleus と呼ばれるマイクロカーネルを基礎に、オブジェクトで構成されるサーバとアプリケーションを動作させることでオペレーティングシステムの機能を提供するようにしている。

Spring では、オブジェクトの実現とは独立したオブジェクトのインターフェイスが重要になるため、これを記述するために記述言語と独立した Interface Definition Language (IDL) を用いている。これは、OMG (Object Management Group) が提案する標準にしたがったものであり、分散型のオブジェクト指向ソフトウェアコンポーネントのインターフェイスを記述するために開発されたものである。

また、オブジェクト指向オペレーティングシステムでは、オブジェクト間でのメッセージ交換やオブジェクト同士のコンテクストスイッチなどによって、処理速度が低下するという問題を持っている。Spring では、基本的にメッセージ交換のメカニズムを Subcontract と呼ばれる仕組みに基づく RPC で行なう Server-based Object によって処理を進めるとともに、サーバを必要としない処理では、RPC を通常の手続き呼び出しとして効果的に処理を行なう Severless Object を積極的に用いるようにしている。

Spring のカーネルは、マイクロカーネルである Nucleus とメモリ管理を行なう Virtual Memory Manager で構成されており、この上にさまざまな機能を提供するサーバ群と、それを利用して動作するアプリケーションが配置されることになる。

各オブジェクトは、Domain と呼ばれるメモリ空間を持ち、Door と呼ばれるものを用いてメッセージ交換をする。ネットワークを介したオブジェクト間の通信は、proxy が仲介することで、ネットワーク上の各計算機の Door 同士を結合し行なうようになっている。

Spring ではセキュリティ機能を提供するため、Access Control List (ACL) を用意し、実

際に処理を行なうオブジェクトの前にメッセージをフロントオブジェクトと呼ばれる ACL の制御を行なうオブジェクトに引き渡し、そこで確認を行なうようにしている。

仮想記憶は、Mach オペレーティングシステムなどと同様に外部ページャの機能が実現できるようになっており、またシステム内に同時に複数の VMM やページャを混在させることができるような機能を有している。これにより、各オブジェクトに適した仮想記憶制御が可能となる。またファイルシステムは、メモリオブジェクトをファイルオブジェクトにマップすることで実現されており、仮想記憶のメカニズムとファイルシステムのメカニズムは、表裏一体のものとなっている。

Spring などのオブジェクト指向オペレーティングシステムで重要な点は、各オブジェクトの名前付けにある。Spring では後で述べる統一的な名前付けサービスを提供しており、それを用いることでシステムに存在するすべてのオブジェクトに名前を与えることができるようになっている。

Spring が提供する UNIX エミュレーション機能は、`libue.so` と呼ばれる共有ライブラリを通常の UNIX オブジェクトにダイナミックリンクすることによって提供される。本ライブラリは、UNIX が提供する各種システムコールを Spring のオブジェクトを用いて実現する機能を提供する。また、UNIX のプロセスのメカニズムは、通常の Spring オブジェクトのドメインとは異なるため、UNIX プロセスサーバが用意されており、これにより UNIX プロセスを実現するようになっている。したがって、UNIX プロセスサーバによって生成された UNIX プロセスは `libue.so` によって UNIX システムコールを実行し、エミュレーションを行なっているのである。

Spring は、IDL で記述されたインターフェイスを持つオブジェクトを組み合わせることでオペレーティングシステムを構成している。これにより、システムのモジュラリティを高くしている。また、Serverless Object などを用いることでオーバヘッドの少ない実装を実現しており、マイクロカーネルアーキテクチャに基づくオペレーティングシステムを効率良く実装したものである。

3.2　Spring における名前づけ

[138] は、Spring における、オブジェクトとそれに付ける名前との対応づけを管理するネームサービスに関する論文である。

Spring のネームサービスの特徴は、単一で拡張性に富むものであり、また、アクセス制御やパーシステンスに対するサポートも行うものである。

3.2.1　基本モデル

コンテキスト (Context) とは、名前とオブジェクトの対応関係をもっているオブジェクトのことをいう。コンテキストからコンテキストへのリンクも可能である。

クライアントオブジェクトは、コンテキストオブジェクトへのリンクを複数保持することができる。これをたどることで、目的のオブジェクトが得られる。

3.2.2　ネーミングポリシー

Spring 自体には、ネーミングのポリシーはないが、現在の実装では、ポリシーは、次の３つある。

- システムごとの共通ネームスペース
- ユーザごとのネームスペース
- ドメイン（クライアントの実行環境）ごとのネームスペース

それぞれ少なくとも１つのコンテキストを親ドメインから渡される。

リゾルブ（名前からオブジェクトを求める）のために、以下のインタフェースが用意されている。

- クライアントインタフェース
- ネームサーバ＝ネームサーバインタフェース
- ネームサーバ＝オブジェクトマネージャインタフェース

3.2.3　ACL&セキュリティ

Spring のネームサービスには、以下のサービスが提供されている。

- 全てのコンテキストパスが、許可されていること
- クライアントが、そのオブジェクトへのパスを許可されていること

- アクセス元へ認証されたオブジェクトを返すこと

Spring のオブジェクトは、ケーパビリティの様に見える。

各コンテキストは、ACL(Access Control Lists) をもち、ネームサーバが、その具体的な意味や実装を規定する。たとえば、サーチを許すか、コンテキストのバインドや削除を許すかなどを規定する。

ネームサービスは、相手を信用していないような関係でリンクすることもできる。必要に応じて、認証して信用できる関係にすることができる。これは、ケーバビリティという形になる。一度信用状態になった際には、これを再利用することで、再度の認証に関わるオーバヘッドを少なくできる。

ネームサービスが、認証とアクセス制御を行うので、これを利用して、クライアントとオブジェクトマネジャーとの間でセキュリティの必要度に応じての認証が可能である。

3.2.4 Spring におけるネームサービスの使い方

Spring 自身には、グローバルネームスペースの概念がないが、Spring のネームサービスがユニバーサルなのでどんなオブジェクトでもネーミングができる。

たとえば、NIS 情報へ Spring ネームスペースからリンクしている。また、Unix の File を Spring の File へ、Unix のディレクトリを Spring のコンテキストへのエンキャプスレーションをしている。

ポリシーとしては、次の３つの組み合わせで実現している。

- システムごとの共通ネームスペース
- ユーザごとのネームスペース
- ドメインごとのネームスペース

Spring では、各マシンはビレッジ (villages) と呼ぶ、設置場所や管理などに応じたグループに属している。ビレッジ内の資源の共有は多いが、ビレッジ間は少ない。

各マシンは、１つのローカルネームスペースをもち、ビレッジは、各マシン間で共有する。ビレッジネームスペースは、エンタープライズレベルネームスペースとさらにつながっている。

各ユーザは、Unix のホームディレクトリの概念に近い、個人のネームスペースをもっている。Unix と異なるのは、個人ごとに自由にネームスペースを接続できることである。

各ドメイン（クライアントオブジェクトの実行空間、Unix のプロセスのようなもの）は、独自のネームスペースをもち、それは、典型的には、マシンローカルとビレッジのネームスペースの一部と、実行したユーザのネームスペースを保持しているだろう。ちょうど、Unix の環境変数のような感じである。

RPC や、プロセスマイグレーションなどで、ドメインが切り替わると、ビューは替わるかもしれない。

ドメインのプライベートネームスペースには、典型的には以下のネームスペースを持つ。

- Private name bindings
  環境変数や、入出力のためのもの
- Shared name spaces
  well-known 名前（~home directory), user directory, device 等）のためのもの
- Generic name spaces containing standard system objects
  system オブジェクト（bin・lib）と service(service/authentication など）のためのもの

子ドメインのプライベートネームスペースには、親ドメインのコンテキストが通常引き継がれる (init ドメインの機能)。

ネームサービスは、いくつかのネームサーバから構成されている。通常、machine/domain 共通のネームサーバと、ビレッジ内に１つあるビレッジネームサーバである。ドメインのネームスペースから、これらが差される。各マシンのネームスペースは、ビレッジネームスペースから差される。

また、Unix の世界と Unix ファイルシステムへ、Spring のネームスペースから差せるように、いくつかファイルサーバが実装されている。

ネームサービスの機能をうまくつかうと、複数のネームスペースのセットを並行して利用することができて便利である。例えば、開発用とリリース用の２つのオブジェクトで、一部のオブジェクトだけ置き換えただけのネームスペースをそれぞれ利用できる。

3.3 Subcontract: Spring RPC のサブシステム

Spring オペレーティングシステムでは、Spring の c++オブジェクト間で使われる RPC ベースの呼出しのために、Subcontract と呼ばれる機構を用いている。ここでは論文 [139] に基づき、Subcontract の概要について述べる。

3.3.1 問題点

Spring のオブジェクトは CORBA におけるオブジェクトと同様、単一のプロセス (Spring 用語では Domain) 内に存在する c++オブジェクトである。プロセス間オブジェクト通信やリモートオブジェクト通信には RPC ベースのメカニズムが用いられる。

RPC はこれまで手続き呼出の位置透過性を保つために広く利用されているが、以下のような不足点がある:

- RPC は通常、単一の通信セマンティクスしか実現しない

  replication/トランザクション/オブジェクト移送/永続性、を実現するような通信セマンティクスはどうやって実現すべきか?

- アプリケーションの要求に基づいて RPC のメカニズムを変更できない

  新しい効率のいいオブジェクト管理機構 (例: replicated object) が登場した場合、もとの RPC システムを変えずにこれを利用したい。

このためにとるべきアプローチには (1) 単一の単一の RPC メカニズムで全てを包含するようなものを導入する、(2) 複数の RPC メカニズムを実現できるような枠組を提供する、のふたつが考えられる。しかしながら、(1) は実行コストや一般性の点で疑問があり、現実的でない。Spring では (2) のアプローチをとり、複数の RPC メカニズムの導入を許し整合性と互換性を保つための機構として subcontract を提案する。

3.3.2 Subcontract の概要

Subcontract とは、「オブジェクト呼び出しと引数渡しの基本的なメカニズムを実現する交換可能なモジュール」である。Subcontract は RPC スタブの OS に近い部分を実現し、Subcontract の RPC スタブのオブジェクトに近い部分に対するインタフェースは標準化されている。このため、Subcontract はオブジェクト自体のインタフェースや実現とは独立に実現することができる。また、Subcontract のインタフェースが守られるならば Subcontract はオブジェクト、および RPC スタブのオブジェクトに近い部分とは独立に変更することが可能である (図 3.1)。

3.3.3　Subcontract のインタフェース

CORBA および Spring のオブジェクト間通信では、通信を行う 2 者の間にクライアントとサーバの区別がある。サーバはファイルサーバなどの役割を果たす側で、ファイルの内部情報などを外部から通信を行うことのできる c++オブジェクトの形で抽象化する。クライアントはサーバの公開したオブジェクトを利用し計算を行う。

ここで、オブジェクトは内部状態、メソッドテーブルと Subcontract descriptor からなる。

クライアント側での Subcontract のインタフェースは以下のとおりである:

- `marshal`

  オブジェクトを marshal して、通信バッファに格納しサーバへ送信する。変換の済んだオブジェクトを送り終ったら、送り元のオブジェクトを消去する。

- `unmarshal`

  `marshal` で変換されたオブジェクトを通信バッファから取り、オブジェクトを再構成する。再構成のためには内部状態の復元に加え、メソッドテーブルと subcontract descriptor の設定が必要である。

- `invoke`

  オブジェクトのメソッドを実行する。

- `invoke_preamble`

  Subcontract が `invoke` によってメソッド実行を処理する前に、通信バッファのサイズ調整や、共有メモリによる通信の場合のバッファ割り当てなどを行う必要がある場合がある。このために、Subcontract は `invoke_preamble` によって前もって呼び出される必要がある。

- `marshal_copy`

  レプリカ等をつくるためにオブジェクトをコピーしてから `marshal` すると結構無駄が大きかった、しかも利用率が高かったので、まとめて扱うインタフェースを作っておいた。

- そのほか

  `copy` と `delete` の操作がある。

サーバ側では Subcontract とサーバオブジェクトのプログラムの依存関係が強くならざるを得ないため、インタフェースは Subcontract ごとに異なる。

サーバプログラム fileserver が singleton subcontract を用い、file object を外部のクライアントに提供する場合、以下のような処理が行われる。

- クライアント側の要求により、Spring object を作成する

  作成の前にオブジェクトの内部状態、メソッドテーブル、サーバ側用スタブが必要である。subcontract が通信ポートである Door を生成して、外部からアクセス可能な Spring object をつくる。

- オブジェクトをサーバの Domain からクライアントの Domain へ移送する

  Subcontract のオペレーション `marshal` と `unmarshal` が使われる。

- オブジェクトのメソッドを起動する

  1. クライアント: メソッドテーブルを見てメソッド (実際にはスタブルーチンを) を起動
  2. クライアントスタブ: `invoke_preamble` を起動 (実際にはなにもしない)
  3. クライアントスタブ: 引数を marshal
  4. クライアントスタブ: `invoke` を起動
  5. クライアント subcontract: Door を介してサーバへ処理要求
  6. サーバ subcontract: サーバ側のスタブを起動
  7. サーバスタブ: サーバ側にある言語オブジェクトのメソッドを起動

- クライアント側でオブジェクトをコピーする

  subcontract の `copy` を呼ぶ。実際にはメソッド table と subcontract descriptor、それから内容 (Door ID くらい?) がコピーされる。

- クライアント側でオブジェクトを使い終ったら

  subcontract の `consume` を呼ぶ。Door ID を消去。Door が完全に不要になったら Door そのものも消去。

このように、Subcontract はオブジェクトの重要な活動全てに介入している。

3.3.4 Subcontract 間の互換性

通信を行うクライアントとサーバの間で異なる Subcontract が利用されていた場合、Subcontract の ID を用いて互換性チェックを行う。もしふたつの Subcontract 間でメッセージの marshalling/unmarshalling が可能な場合には互換性がある、としてそのままで通信を行う。互換性がない場合や Subcontract のプログラムコードがまだメモリ内にない場合には、Subcontract ID を用いて Subcontract のプログラムコードをダイナミックリンクする。

Subcontract の導入によりオブジェクト間通信のセマンティクスは多様になるが、セマンティクスの差異は型システムの導入により検査・解決する。

3.3.5　Subcontract の実例

Subcontract の実例としては、(a) 複製オブジェクトを利用して通信回数を削減する Replicon subcontract、(b) オブジェクト間通信に使う通信ポート (Spring 用語では Door) を複数オブジェクトで共有して、通信ポートの数を節約する Cluster subcontract、(c) オブジェクトのキャッシングをサポートする Caching subcontract、(d) 永続オブジェクトのために通信切断時の再接続をサポートする Reconnectable subcontract、などがある。将来的にはトランザクション処理のための Subcontract やビデオデータ通信を行うオブジェクトのための Subcontract を設計実装する予定である。

3.3.6　Spring オペレーティングシステムと Subcontract

Subcontract はオペレーティングシステムと独立に利用できるものである。Spring では Door という効率的な IPC メカニズムが存在するため、これを Subcontract 実現のために有効に利用することができる。他のオペレーティングシステムでは Door のかわりに IP ネットワークパケットや各オペレーティングシステムの IPC システムを利用することができる。

3.3.7　実行性能

Subcontract を導入することにより通常の RPC システムに比べて実行時間に加わる損失は、オブジェクト呼出 (関数ポインタ経由の関数呼出 2 回) と marshall/unmarshall(関数ポインタ経由の関数呼出 1 回) である。

中程度の粒度のオブジェクト間通信による評価では、subcontract によるオーバヘッドは RPC コストの 5%から 10%であった。(ただし、プロセス間 RPC 全体の実行時間は 10μsec 程度である。であるからオーバヘッドは 0.5μsec から 1μsec である)。しかし、このコストはリモート通信の場合にはネットワーク上の伝搬遅延で無視することができる。また、Subcontract 導入により各種の最適化が可能なため、オーバヘッドが存在するからといってそれが即損失であるとは言えない。

3.3.8　関連研究

関連研究としては Argus や Eden などの RPC に関する研究、Smalltalk や 3-KRS などの自己反映計算に関する研究、CORBA における object adapter があるが、これらの詳細はここでは述べない。

3.3.9　結論

Subcontract は RPC スタブを全て変更することなしに、各種のオブジェクト実行メカニズムを用いるための機構である。Subcontract の導入により、各オブジェクトは好みの実行

メカニズムを自由に選択できる。また、セキュリティ/性能/頑強性/アクセス制御などのさまざまな機構を、単一のメカニズムで提供できる。

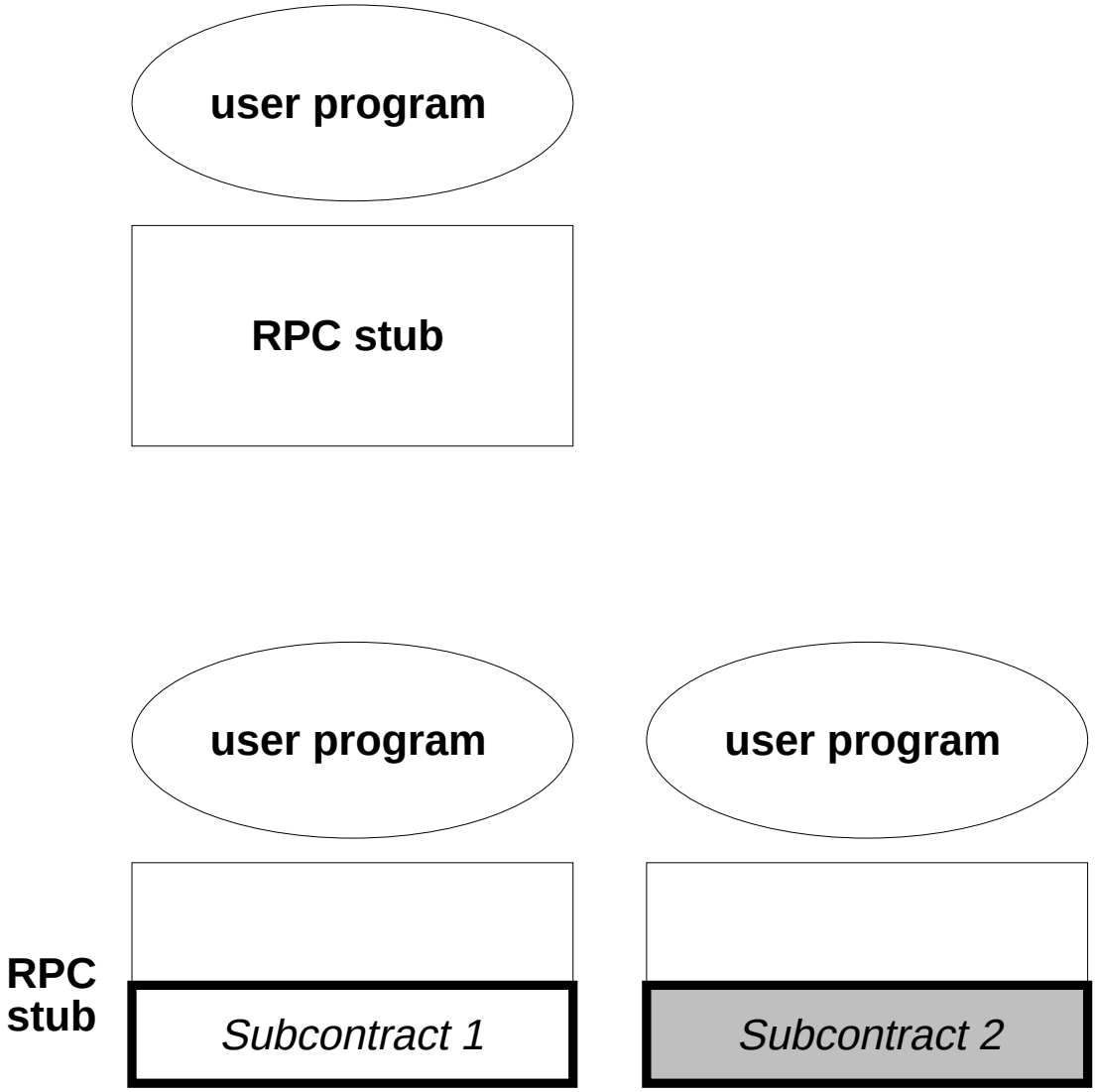

図 3.1: 従来の RPC stub と Subcontract 導入後の RPC stub。Subcontract では、Subcontract と RPC stub 上位部の間のインタフェースを標準化することにより、Subcontract の交換を可能としている。

第 4 章

モーバイルコンピューティングにまつわる諸問題

4.1 背景

近年の小型の携帯型コンピュータの普及により携帯しながら使うというモーバイルコンピューティングが現実のものとなってきている。しかし現在の UNIX ベースのオペレーティングシステムはそういった使われ方は全く想定されていないため、利用に際してさまざまな問題点があらわれてきている。

OS ワーキンググループでは、後述する WILDBOAR というパッケージを基にして、携帯しながら利用した場合のさまざまな問題点を洗い出し、その問題点に対する解決策や、モーバイルコンピューティングにおける OS のさまざまな機能のありかたを議論した。

なお本節はモーバイルコンピューティングを考慮した新しい OS などを作成する場合に参考資料となるべくまとめられたものである。今後のモーバイルコンピューティングを考える時に以下の議論が役にたてば幸いである。

4.2 WILDBOAR とは

WILDBOAR とは BSD/OS もしくは NetBSD 上で、APM(Advanced Power Management) と PCMCIA カードデバイスのサポートをおこなうためのパッケージである。

以下に WILDBOAR によって可能になることを示す。

- PCMCIA の規格に則ったさまざまなインターフェイスカードを使用することができる
  - モデム
  - イーサネット
  - ATA フラッシュディスク
  - SCSI カード
  - ビデオキャプチャカード

- 電源を入れたまま、必要な時にコンピュータに PCMCIA カードを挿して使用することができ、さらに必要がなくなれば抜くこともできる
- 多くの携帯型コンピュータにおいて *suspend*(一時停止) と *resume*(一時停止からの復帰) ができる

なお WILDBOAR パッケージの開発には WIDE プロジェクトのメンバーが多数参加し、協力しているが、OS ワーキンググループの正式な活動ではないということをここに断っておく。

4.3 WILDBOAR の現状

1993 年末から開発がはじまった WILDBOAR だが、当初はカードの抜き挿しが検出できるだけであった。その後クライアントのドライバが開発され起動時にカードを挿しておけば使用できるようになった。さらにバッテリを節約するために、*suspend* 中にもカードに電力が供給されてしまうという状態を防ぐ必要がでてきた。このために APM を利用して機種依存性のない電力コントロールが実現され、最終的には動作中のカードを抜き挿しして使用できるようになった。現時点での最新版 (1996 年 5 月現在) の WILDBOAR ではさらに以下のようなことが可能になっている。

- バージョン 1 タプルの管理がカーネル外でおこなえる
- ユーザプロセスがイベントを認識することができる

バージョン 1 タプルとは各 PCMCIA カードの不揮発性メモリに記録されたさまざまなパラメータである。従来は挿入されたカードを識別するためのパーサがデバイスドライバ内に直接書かれていたが、現在では必要なパラメータをファイルに書いておいて、それをカーネルに読み込ませることが可能になっている。

さらにあるデバイスを経由して、ユーザプロセスがカードの抜き差しと *suspend* と *resume* のイベントを検出できるようになった。この機能により、例えば *resume* 時に何かコマンドを実行することが可能になった。またイベント時に実行されるコマンドは通常のファイルに記述できるので、ユーザの利用形態によってさまざまな応用が可能になる。

残念ながら、現時点では最近市場に出回り始めた一枚で複数の機能を持つマルチファンクションカードはサポートされていない。これは既存の OS において複数のデバイスドライバが割込みを共有するという状況が全く考慮されていないためである。

4.4 WILDBOAR 利用時の問題点

ここでは WILDBOAR を利用していた場合のネットワーク関係と時間管理の問題点について議論する。ファイルシステムに関する問題は節をあらためて考察する。

4.5 ネットワーク関係

コンピュータを持ち歩きつつネットワークに接続して使用している場合には、以下にあげるようなことが問題になる。

1. 古い ARP キャッシュが残る
2. DHCP によるパラメータ設定では不十分
3. PCMCIA の同じ種類のイーサネットカードが 2 枚あった場合、同じインターフェイスとして扱いたい場合と違うインターフェイスとして扱いたい場合がある。
4. PCMCIA の違う種類のイーサネットカードがそれぞれ 1 枚づつあった場合、同じインターフェイスとして扱いたい場合と違うインターフェイスとして扱いたい場合がある。

ARP キャッシュ

コンピュータを携帯しながら移動した場合に、直前に接続していたネットワーク上のホストの ARP キャッシュが残ってしまうことがまず問題としてあげられる。この場合、移動した後に直前まで接続していたネットワーク上のホストの MAC アドレスが ARP キャッシュに残っていると、そのホストとはすぐに通信することができない。ARP キャッシュに MAC アドレスが存在するとその通信相手が現在接続しているネットワークにあると思ってしまうからである。現状では解決策として ARP キャッシュの保持時間 (20 分) が経過して自動的にそのエントリが消去されるのを待つか、root になって該当する ARP キャッシュのエントリを手動で削除するという手段が考えられるが、こういった方法ではあまりに効率が悪い。

より良い解決方法としては現在の WILDBOAR では *resume* した時にコマンドを実行することができるようになったので、arp コマンドを改造して ARP キャッシュを全てクリアするオプションを用意して実行することにより、移動時に古い ARP キャッシュを消去することが可能である。

しかし移動する時に *suspend* しないで、起動したまま移動することも考えられるが、こういった場合には WILDBOAR の範疇では解決できない。単純な解決方法としては、イーサネットのキャリアが検出できなくなったらカーネル内部で ARP キャッシュを無条件でク

リアするという処理を追加するということが考えられる。この場合一時的に HUB やカードの調子が悪くなった場合にも ARP キャッシュがクリアされてしまうが、再び ARP をおこなっても、そのオーバヘッドは許容できる程度であると考えられる。

また他の解決方法として、イーサネットのキャリアが検出できなくなると何らかの方法でプロセスへ通知するという仕組みを作成することが考えられる。この仕組みを用いて、プロセスが ARP テーブルをクリアするという方法も考えられる。しかしキャリアの検出はイーサネットコントローラによっては不可能なものもある。

このように様々な方法が考えられるが、基本的にイーサネットに依存しているため、他のデバイスに対してはまた別のアプローチが必要になってしまう。普遍的な解決方法としては、デバイスの状態をカーネルに通知するというアプローチが有効であると考えられる。

DHCP

あるホストが移動して別のネットワークに接続した場合、通信をおこなうためには新規に IP アドレスを取得しなければならない。そのために開発されたプロトコルが DHCP(Dynamic Host Configuration Protocol) である。WIDE プロジェクトで開発された DHCP クライアントでは新たにアドレスを取得しなおすためには、デーモンプロセスとして動作している DHCP クライアントにシグナルを送らなければならない。移動する際に *suspend* すれば、WILDBOAR では *resume* した時になんらかのコマンドを実行することが可能なので、DHCP クライアントにシグナルを送るというコマンドを作成すればアドレスの再取得を実現できる。しかし *suspend* せずに起動したまま移動した場合は、アドレスを取得するために手動で DHCP クライアントにシグナルを送る必要がある。

この問題の解決策としては、前述したイーサネットのキャリアを検出する仕組みがあれば、その応用として DHCP クライアントと連動させるという方法があげられる。イーサネットのキャリアを検出しなくなってから一定時間が経過し、再びキャリアを検出した場合を移動したとみなせば、完全に自動で IP アドレスをはじめとする各種パラメータの再設定が可能になる。またイーサネット以外に、無線 LAN の使用時にローミングが起きた場合にもこういった仕組みは有効である。

別の問題として、DHCP によってネームサーバのアドレスは取得可能であるが、一般的な UNIX ベースの OS においてはネームサーバのアドレスは`/etc/resolv.conf` というファイルに直接記述されている。そのため DHCP クライアントがネームサーバのアドレスを取得しても`/etc/resolv.conf` ファイルを書き換えなければならない。設定ファイルを直接書き換えるというのはかなり乱暴な方法であるため、新たな仕組みを模索中である。具体的には`/etc/resolv.conf` と等価のファイルを指定して、もしそのファイルが存在した場合は`/etc/resolv.conf` より前に参照するようにライブラリを改造するなどの方法が考えられている。

ネットワークインターフェイスの抽象化

まずここに 3c589(デバイス名は ef) というイーサネットカードが 2 枚あるとする。1 枚目 (ef0) を挿して使って抜いてから、もう 1 枚を挿したとする。この場合新しく挿した方は ef1 というデバイスになるが、これが ef0 になって欲しい場合もあれば、現状通り ef1 になって欲しい場合もある。

別の例として 3c589 と XJ10BT(デバイス名は mz) というイーサネットカードがあるとする。3c589 を抜いて XJ10BT を挿した場合、それぞれ ef0 と mz0 となるが、同じインターフェイスとして勝手に切り替わってそれまでおこなっていた通信を継続して欲しい場合もあれば、別のインターフェイスとして認識して欲しい場合もある。

こういった要望に答えるためには通常のネットワークインターフェイスの上位に、図 4.1 のような仮想的なネットワークインターフェイス (デバイス名は vif) を設ける。

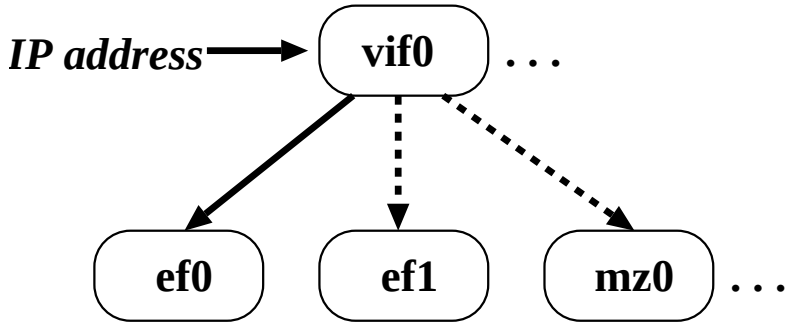

図 4.1: 仮想ネットワークインターフェイス

このように仮想的なネットワークインターフェイスと実際のイーサネットカードを 1 対 1 で対応させ、その対応を動的に変更可能な仕組みを作成すれば、上述した利用者の要望に応じることが可能になる。しかしそのためには、イーサネットカードの切替えとともに、カーネル内部の経路表の書き換えが必要になる。またイーサネットカードが切り替わるということは、IP アドレスと MAC アドレスの組合せが変わるということなので、その時点で通信している相手の ARP キャッシュを書き換えなければならない。これは ARP レゾルブパケットをブロードキャストして、同じネットワークに接続されているホストの ARP キャッシュを上書きすることによって解決できる。

また上記のような機構を用いれば、単にイーサネットカードを取り換えた場合だけでなく、無線 LAN とイーサネットを切替えた場合などにも連続的な通信をおこなうという応用が可能になる。

4.6 *suspend* と時間管理

ホストの *suspend/resume* が可能になったことにより、従来のように電源を切る前には shutdown し、電源を入れるたびに起動に時間をとられるということがなくなり、利用者に

とっては非常に便利に使えるようになった。しかしこの *suspend/resume* にも問題点はある。長時間 *suspend* していた場合、*resume* した瞬間に内部時間を急速に進めて現在の時刻にあわせようとする。そのため *suspend* していた間にたまった、`cron` コマンドによって実行される job が連続して発生してしまう。例えば 3 日間 *suspend* していた場合、3 日分の daily job(`fsck` など) をおこなうため、短時間に 3 回もの `fsck` などがおこなわれてしまう。実際には 3 回もの連続した `fsck` は必要ではないうえに、ホストにかなりの負荷がかかり利用者が快適に利用できるようになるまでにかなりの時間がかかってしまうことになる。

この解決策として考えられる方法としては、`cron` コマンドに *suspend* していたら無視するなどのサービスの区別する仕組みがあればよい。しかし daily job のように、3 日 *suspend* していたら 3 回分はおこなう必要はないにしても、1 回分はやっておいたほうがよい場合もある。よって `cron` コマンドは時間経過にしたがって、全ておこなうべき job と 1 回だけおこなうべき job、そしておこなわなくてもよい job というように各 job を区別して、その通りに実行できれば理想的である。そのためには `cron` コマンドが *resume* というイベントを認識し、通常の時間経過の動作とは別の動作をおこなうように改造する必要がある。

4.7 記憶装置とファイルシステム

PCMCIA カードは機械的に利用者による自由な抜き挿しが可能なため、PCMCIA タイプのディスク (ハードディスクや ATA Flash Memory) を使用していた場合には、突然ディスクが抜かれてしまうという事態が考えられる。従来の UNIX ベースの OS においては、動作中にディスクが突然存在しなくなるという状況はまったく考えられていなかった。しかし PCMCIA タイプのディスクはあらゆるタイミングで利用者によって突然に抜かれる可能性がある。

もし mount 中のディスクが突然抜かれた場合には、以下にあげるような問題が生じる。

1. 該当ディスクへのアクセスがブロックされる
2. 計算機上にディスクに記録できていないデータが残る
3. ディスク上に不完全なデータが残る
4. ハードディスクの場合、動作中に突然電源が切られることになり、機械部品に悪影響がでる

これらによって、以下のような現象が生じる。

- ファイルアクセスを行なったプロセスがブロックされる
- update デーモンや sync コマンドがディスクに書き込めなくてハングアップする
- ディスク上のファイルシステムが不完全な状態になる
- ディスクに記録される前のデータが失われてしまう
- ハードウェアが破損する

ここでは、これらの問題の解決方法について考察する。

ディスクへのアクセスがブロックされる

抜いたディスクにアクセスしてしまうという問題の解決策として考えられる方法としては、それが PCMCIA スロットから抜かれたというイベント情報の通知と、そのディスクのマウント情報から、ディスクを利用しているファイルシステムへのアクセスを無効にすることがあげられる。

具体的にはそのディスクそのものへのアクセスをエラー終了にする方法や、ファイルシステムへのアクセスの段階でエラーにする方法などが考えられる。

ディスクに記録できていないデータが残る

実際に書き込まれないうちにディスクが抜かれてしまうという問題については、現在ディスク I/O を高速化するために用いられているバッファキャッシュという仕組みを適応せず

に、writeの度にきちんとディスクへ書き込むことにすればある程度解決できる。この場合、ディスクアクセスの効率がかなり悪化することが予想されるが、最低限ファイルを close した際には必ず書き込みを実行するなどの工夫によって改善できる。これによって、現状に比べれば実際にディスクに書き込まれないという事態は減少するはずである。

しかしこの場合でも、利用者のディスクを抜くタイミングによっては不完全な書き込みになる。任意のタイミングでディスクが抜かれる環境下では、システムによる自動的で完全なこの問題の解決は事実上不可能である。したがって、システムが書き込みを完了できない場合などには、利用者による操作が必要となる。例えば、書き込みが終了する前にディスクを抜かれた場合には、ビープ音などで利用者に警告を発し、利用者がその警告を受けてその時抜いたディスクを再び挿せば、書き込みを完了させるなどの方法が考えられる。利用者に警告を出す場合、ディスクが抜かれてからどの程度の間書き込みが終了していないデータを保持するのか、他のディスクが挿された場合にどうするかなどの問題が残る。

利用者が再度挿さないディスクのデータを保持する場合、該当するデータを保持するメモリを解放できないため、OS のカーネルの記憶領域を圧迫する。この領域は一般的には仮想記憶を利用していないため、一般的な解放できないデータよりも問題は深刻である。

また、同じディスクが挿された場合には、バッファの内容を書き出すする機構が必要だが、異なるディスクが挿された場合には当然ながらその内容を書き出してはいけない。そのためにはディスクの個体を識別をしなければならない。識別の方法としては disklabel と super block などのカーネル内部にあるデータとタイムスタンプを比較すれば実用上は十分であると考えられるが、super block 等の更新される領域を更新中に、ディスクを抜かれた場合にはこの方法では対処できない。

また、*suspend* 中にディスクを抜かれば場合には、それを *resume* 後に検出することになり、利用者による対処が難しくなる。

ディスク上に不完全なデータが残る

前途したような方法により、全てのデータをディスクに書き込ませることができればこの問題は解決する。しかし、実際には不完全なデータが残る可能性は通常のディスクよりも高いと考えられるので、より強力なファイルシステムの整合性のチェック機構が必要となる。

突然の電源断による悪影響

ハードディスク等の機械的な部分を持つカードにおいては、動作中の電源切断が機械部品に悪影響を与えることが予想される。これに関しても、ディスクが自由に抜かれるため、抜かれる前にディスクヘッドを安全な状態にするなどの前処理を行なうことができない。

これに関しても、ある程度の時間アクセスがなければモータを止める等の制御が考えら

れるが、完全ではない。

キャッシュファイルシステムによる解決

ディスクカード上のファイルシステムを直接利用するのでなく、内蔵ハードディスク等にキャッシュを用意し、大容量のキャッシュを持った仮想的なファイルシステムを経由して利用する方法も考えられる。この場合、書き込みが終了していなくてもキャッシュのみで動作し、再度挿入されてから書き戻すように設計することにより、ファイルシステムを利用しているアプリケーションからディスクの抜き挿しを隠蔽することが可能になる。

しかし、この方法でもディスクの識別の問題などは残る。

制限による解決

これまで述べてきた方法は、すべて PCMCIA カードが任意のタイミングで抜かれることを前提にしていた。これらの方法に対し、Microsoft Windows95 で採用されているように抜くタイミングを制限する方法がある。すなわち、利用者が PCMCIA カードを抜く前にシステムに対して予告を行い、システムがカードを抜くことを許可した後に、利用者が実際に抜くという方法である。

この方法を用いればディスクが抜かれる前にファイルの書き込み等の処理が可能になるので、前記の問題点の殆どは解決できる。しかし、利用者は不注意などにより必ずしも予告を行うとは限らないため、利用者の不正な抜き差しを検出し、利用者に警告、警告に従うことにより復帰するような機構が必要になると考えられる。

4.8 まとめ

これまでに述べたように、コンピュータを携帯しながら利用する場合には、現在の OS の仕組みそのままでは解決できない様々な問題があることがわかった。例えばネットワークを利用する場合には、現状の DHCP 以上にさまざまな設定が必要であり、ネットワークインターフェイスのありかたも見直しが必要であることが判明した。さらに *suspend* という従来には考えられなかった動作がサポートされることによって、OS における時間の扱いかたにも新たな観点が必要であることも表出した。また PCMCIA カードは任意のタイミングで抜くことができるため、PCMCIA タイプのディスクを利用すると様々な問題が生じてしまう。こういったファイルシステムとキャッシュの関係の問題を完全に解決することは難しく、利用者の利便性などを一部犠牲にする必要があることも判明した。これらをどのように解決するべきかは今後の課題であり、今後新たに OS を設計する場合などには、ここで議論したような問題をも考慮すべきである。

第 5 章

おわりに

本章の最後に、次年度以降の活動について簡単に述べる。次年度へ向けてのワーキンググループ体制の改正に伴い、比較的短期間で明確な成果を持つプロジェクト単位での活動が求められている。

これを受け、96 年 3 月の WIDE 合宿で議論した結果、OS ワーキンググループとしては次年度以下のように活動することで合意した。

- OS ワーキンググループとしての大枠の活動は行わない。より小さな単位でのワーキンググループ活動へと移行する。
- WIDE プロジェクトにとって各種研究用 OS/商用 OS について見識を深めておくことは重要であると思われる。そのような活動は積極的に WIDE メイリングリスト (`wide@wide.ad.jp`) で行うこととする。

報告書執筆の時点では、小さな単位のワーキンググループとして以下が活動準備段階にある:

- Apertos で遊ぼうワーキンググループ (仮称)
  Sony CSL で開発されている Apertos オペレーティングシステムを用いて新しいプロトコルの実験環境を構築し、これを利用・改良する。
- x-kernel で遊ぼうワーキンググループ (仮称)
  x-kernel を用いて新しいプロトコルの実験環境を構築し、これを利用・改良する。
- router ワーキンググループ (仮称)
  WIDE 独自の IP ルータを製作しようとするグループである。ルータ向け OS の設計開発などが望まれる。

これまで OS ワーキンググループとしての活動にご協力いただいた皆様にこの場を借りて感謝の意を表したいと存じます。これからも各小ワーキンググループとしての活動をご支援頂ければ幸いです。