第 18 部

オペレーティングシステム

第 1 章

はじめに

WIDE OS ワーキンググループにとって 1993 年度はある意味で非常に重要な年となった。WIDE プロジェクトは 1990 年からカリフォルニア大学バークレー校の Computer Systems Research Group(CSRG) と共同研究を行っている。CSRG による NET/2(Network Release 2) のリリースは、USL との訴訟問題が発生し配布が一時中断されていたものの 1994 年の 2 月に解決、最終的に 4.4BSD-Lite として登場する。これによって我々は良質なオペレーティングシステムのソースコードを再び自由に入手することが可能になった。本論文はまず最初に 4.4BSD を中心に最近の UNIX 系オペレーティングシステムの標準化の動向などについてまとめたものを報告する。

一方で我々を取り巻く計算機環境は大きく変化し従来のワークステーションを中心とした固定的な運用形態に加えて、ノートやサブノートと呼ばれる小型で携帯可能な計算機が登場している。これらの多くは UNIX をサポートできるだけの性能をもっているにもかかわらず、オペレーティングシステム側で十分に生かせるだけの機能を用意していない。携帯型の計算機で UNIX のようなオペレーティングシステムを運用することを考えた場合、従来あまり考える必要のなかった、

- 移動するノードを考慮したネットワーク機能や分散ファイルシステム
- パラレルポートや IC カードに接続するといった付加的なデバイスドライバ
- レジューム機能を用いた運用や、バッテリーによる運用を考慮した細かなパワーマネジメント機能

などが必要だと考えられる。前者は VIP ワーキンググループや分散ファイルシステムのワーキンググループの方で報告があるだろう。本論文ではパワーマネジメントシステムについて報告する。

次にオペレーティングシステムの国際化について考えてみると、現在多くのベンダー製のオペレーティングシステムでは国際標準化案にしたがった国際化機能が用意されている。しかし実際にはそれぞれのベンダー独自の思惑による実装がなされた為か利用するユーザ側の立場が考慮されているとはいえない。またソースコードが入手困難であることも WIDE プロジェクトにとって利用しにくい原因となっている。本論文では我々が開発した ISO C MSE(Multibyte Support Extension) 準拠マルチバイト文字処理関数ライブラリの実装レポートを報告する。

最後に新しいオペレーティングシステム実装の試みとして、Cubix OS を紹介する。これは 64 ビットプロセッサで提供しうる 64 ビットの広大な仮想アドレス空間を積極的に利用した SASOS (Single Address Space OS) のコンセプトから生まれたものである。

第 2 章

BSD Unix 系オペレーティングシステムの動向

1993 年 6 月、最後の BSD Unix であるとかねてから噂されていた 4.4BSD がリリースされた。BSD Unix を開発してきた University of California Berkeley (UCB) の CSRG (Computer Systems Research Group) は、この 4.4BSD をもって BSD Unix の最後のリリースとし、CSRG 自身も解散することを宣言している。ここでは 4.4BSD を中心とした最近の BSD Unix の動向についてレポートする。

2.1 4.4BSD 以前

BSD (Berkely Software Distribution) Unix という名前で本家 AT&T Bell 研究所の Unix から離れ独自の発展を遂げてきた UCB CSRG による Unix は、1986 年に 4.2BSD でネットワーク機能がサポートされたのを機に、研究者の枠を超えて広く利用されるようになった。さらに、その改良版である 4.3BSD (1988 年) はワークステーション用の OS として多くの機種によって採用され、現在のワークステーションの普及の大きな原動力となった [240]。

4.3BSD 以降もマイナーリリースは続けられ、1988 年には VAX に加えて CCI Power という新しいアーキテクチャをサポートした 4.3BSD-Tahoe が配布され、その年の末には BSD Networking Software という名前で 4.3BSD-Tahoe の中のフリー部分が配布されている。

1990 年の 4.3BSD-Reno 版は、マイナーリリースながら OSI ネットワーク、NFS (Network File System) [241] [242]、MFS (Memory File System) [243] などの多くの新機能をサポートした。主な特徴を次に挙げる。

- DEC VAX 8600, 8650, 8200, 8250, MicroVAX II, III サポート
- HP, i386 サポート
- ISO/OSI ネットワークプロトコル
- VFS 仮想ファイルシステム
- NFS ネットワークファイルシステム
- MFS メモリファイルシステム

- FFFS (Fat Fast File System)
- Kerberos 認証機構
- カーネル内汎用メモリアロケータ
- TCP/IP の拡張 (ヘッダ予測、CSLIP、階層的経路制御など)
- 新 QUOTA システム
- IEEE 1003.1

翌 1991 年にリリースされた BSD Networking Software, Release 2 は、やはりライセンスフリー部分のみからなるものだが、それまでの課題であった仮想記憶システムを根本的に入れ替えた大幅なバージョンアップで、内容的には 4.4BSD のプレリリースと呼ぶべきものであった [244]。ただし USL のライセンスに抵触しない部分のみのリリースであったため、カーネルの中の一部のモジュールが欠如していたり、存在しないコマンドもあった。その後このリリースは NET/2, net2 などと呼ばれる。

NET/2 には、もう 1 つ大きな特徴があった。Intel 386/486 アーキテクチャをサポートしたフリーのコードであったという点である。実際にそのコードを実装したのは CSRG ではなく William Jolitz であった。Jolitz は、4.3BSD-Reno がリリースされる前から 386BSD という名前で 4.2BSD の IBM-PC への実装を行っていた。彼はそれを CSRG に寄付し、世の中の多くの PC ユーザが BSD Unix を手に入れられることを望んでいた。NET/2 には彼の寄付したコードが含まれていたが、カーネルの非常に重要な部分が欠如したものであったため、それを元にしてすぐに PC 上で BSD Unix が利用できるユーザは相変わらず少数だった。

NET/2 は未完成な OS だったが、再配布が自由であることから先の BSDI 以外にもそれを元にして OS を開発、配布する動きが活発になった。William Jolitz は独自に 386BSD のサポートを開始し、BSDI 社の BSD/386、 Mach の BSD サーバである BNR2SS などがこれに加わる。現在では 386BSD はさらに NetBSD, FreeBSD などの支流を生み出している。

BSD Unix は、AT&T ベル研究所によって開発されたコードを元にして作られているため、利用に際しては元となるコードの使用権が必要である。現在その権利を持つ USL (Unix System Laboratories) は、 NET/2 を USL のライセンスに無関係に配布することは違法であるという見解を持ち、BSDI とカリフォルニア大学に対して訴訟を起こした。しかし USL が BSDI 社に求めていた配布差し止めの要求は裁判所によって却下され、BSD/386 は正式に出荷されることとなった。Novell による USL の買収、Unix 商標権の X/Open への売却など、USL をめぐる情勢は非常に大きく変化しており、訴訟の行方にも影響を与えると思われていたが、1994 年 2 月になってようやく和解するに至った。

2.2 4.4BSD

1992 年 7 月付で、4.4BSD-Encumbered という名前で 4.4BSD のアルファリリースがアナウンスされた (日本に対しては 1993 年に 2 月頃に配布が始まった)。そのアナウン

スの直前、4.4BSD のリリースを最後に CSRG は解散し、今後 BSD Unix の開発は行わないという発表があった。理由としては、世界的な不景気の影響を受け開発資金の調達が困難になってきたこと、BSD Unix に対する社会的必要性が薄れてきたこと、主力メンバーである Mike Karels を失ったこと (これも原因は資金難である)、などが挙げられている。

1993 年 6 月 1 日付で、CSRG から 4.4BSD 最終リリースの案内があった。冒頭には次のような言葉で、それが CSRG が配布する最後の Unix であることが宣言されている。

> This distribution is the final release that will be done by the Computer Systems Research Group (CSRG).

4.4BSD が最終的にサポートしたハードウェアは次の通りである。

hp300	HP 9000/300 シリーズ
pmax	DECstation 3100/5000
sparc	SparcStation 1, 2
luna68k	Omron Luna
news3400	Sony NEWS

HP と Luna は Motorola 68030/40、DEC と Sony は MIPS R2000/R3000、Sun は SPARC をそれぞれ CPU としている。DEC と Sony は同じ MIPS の CPU を使用しているが、バイト順序が異なる。

この他、386BSD や BSD/386 のベースとなっている Intel の 386/486 のコードも入るが、NET/2 のコードがそのまま入っているため、カーネルの他の部分の変更に追随していないのでそのままでは動作しない。VAX と Tahoe のコードも含まれているが、これらは新しい仮想記憶システムに対応していないため、まったく動作しない。Kirk McKusick によれば、VAX の仮想記憶を新しいものに対応させるのは 1ヶ月程度の作業だろうということだった。

CSRG が主な開発環境として使用しているのは、HP-9000 の 300 シリーズである。したがって、このコードが最も安定している。

OMRON Luna は同じ CPU を使用する HP と構成がよく似ていて、共有するコードも多い。HP は現在の BSD のリファレンスマシンであるから、Luna のコードの質もかなり高いと言ってよいだろう。リリース直前に 68040 ベースの Luna II もサポートされた。Sony NEWS のサポートは WIDE プロジェクトの一環として Sony の協力を得て行ったものである。こちらは pmax (DECstation) との共有部分が多く、マシン異存のファイルのうちで pmax へのシンボリックリンクになっているものも少なくない。現在 NWS-3200, NWS-3400 等シングルプロセッサの RISC マシンで動作し、CISC マシンや NWS-3800 等の I/O プロセッサ付きの機種はサポートしていない。

4.4BSD-alpha と 4.4BSD は、機能的にはほとんど違いはない。主な特徴を次に挙げる。

NFS (Lease) NFS (Network File System) は、カナダの Guelph 大学の Rick Macklem が仕様を元に実装したもので、Sun の NFS と相互に接続可能である。NET/2 にない機能として、Lease と呼ばれるキャッシュ管理のメカニズムを備えている [245][246]。

LFS (Log-Structured File System) UCB の Sprite プロジェクトの成果で、ディスク上に連続してデータを書き込むことでディスクアクセスとヘッドのシークをできるだけ減らし、書き込みのスループットを向上させたファイルシステムである。また、クラッシュ時の回復に要する時間も短縮されている。

スタッカブルファイルシステム UCLA の Ficus プロジェクトの成果である。ファイルシステムを階層的に実現することにより、それぞれのファイルシステムの独立性を高め、新たなファイルシステムの開発を容易にし、拡張性を向上する。 [247][248]

64 ビットファイルサイズ 従来の Unix では、ファイルのサイズは 32 ビットで表されていたため、2 ギガ以上のファイルを操作することはできなかった。昨今のハードディスクの大容量化により、この制限が現実の問題となってきたため、4.4BSD ではファイルサイズが 64 ビットに拡張された。

共通デバイスドライバ 他の PC Unix で、generic scsi などと呼ばれる共通の SCSI ドライバの原型。

IEEE 1003 (POSIX) IEEE 1003 の POSIX のシステムインタフェースと、ライブラリ、コマンドの仕様がかなりのレベルでサポートされている。

ファイルサイズが 64 ビットになったことは、思ったよりシステムに対して与える影響が大きかった。リリースが予定よりも遅れた大きな原因になっている。NET/2 までのカーネルでも 64 ビット整数はサポートされていたが、32 ビット整数の配列で代用することができた。しかし、4.4BSD では完全に 64 ビット対応のコンパイラでなければ、カーネルをコンパイルすることすらできなくなっている。現在のところ 4.4BSD を作ることができるのは、おそらく gcc だけだろう。gcc 2.0 以降では、64 ビットのデータ構造が 64 ビットの境界に合わされるようになった。これによって、カーネル内のいくつかの構造の再構成が強要された。

スタッカブルファイルシステムが採用されて、何種類ものファイルシステムが採用されている。/usr/src/sys/fs の下には次のようなディレクトリが存在する。

deadfs umount されたファイルが属する

fdesc プロセスのファイルディスクリプタをアクセスする

fifofs FIFO を実現する

kernfs カーネルに関する情報

lofs ループバックファイルシステム

nullfs Stackable Filesystem のスケルトン

portal Portal Filesystem

specfs スペシャルファイル

umapfs uid のマッピングを行う Stackable Filesystem のサンプル

また、 4.4BSD では新しく LFS (Log-Structured File System) というファイルシステムが採用された。LFS は、UCB Sprite プロジェクトの成果であり、Mendel Rosenblum, John K. Ousterhout 等によって実装された。目的は従来のファイルシステムが持つ次のような問題点を解決することである。

- ディスクのスループットは向上してもアクセス速度は変わらない
- read の効率はキャッシュにより解決されるので write が問題となる
- FFS では小さなファイルに書き込みを行う時には、ディスクのバンド幅の 5% 以下しか利用されなていない

LFS は、以上のような問題点を解決するために、ディスクアクセスのためのヘッドのシークを極力排除するという方針をとっている。そのため、ディスクに書き出すべきデータは連続したブロックに更新情報として書き込まれていく。これが Log-Structured と呼ばれる所以である。Unix では、小さいファイルの作成と消去を繰り返す処理が非常に多い。この処理は inode データの更新を伴うため、物理的に離れた場所にあるデータを更新しなければならず、ディスクヘッドのオーバーヘッドが大きなコストを持ってくる。LFS は主に、小さなファイルの作成とデータの書き込みに焦点を当てたファイルシステムである。

また、ディスクの中で最近更新したデータが局所化されているので、システムがクラッシュした際の復旧の点からも有利である。定期的にチェックポイント情報を更新し、クラッシュした際には、最後のチェックポイントより後に更新されたデータだけを検査すればよい。従来のファイルシステムでは、ファイルシステム全体を走査して整合性の確認をしなければならなかったの対して、検査するデータ量が大幅に少なくなっている。

最終リリースではサポートされると言われていたが、結局 4.4BSD ではサポートされなかった機能がいくつかある。これは次のようなものである。

Bstreams AT&T ベル研究所の 8th Edition Unix の Streams と、BSD の socket を統合した BStreams と呼ばれる新しいインタフェース。

ネットワークの高速化 LBL (Lawrence Berkeley Laboratory) の Van Jaconson による、TCP/IP のネットワークコードの大幅な改良。実装上の階層をできる限り排除することと、ネットワークハードウェアからの割り込みの回数を極力減らすというのが基本的なアプローチである。

バッファキャッシュと仮想記憶の統合 現在はファイルシステムのバッファキャッシュと仮想記憶システムでは、別々のメモリプールを管理している。この 2 つを統合して、互いのメモリを単一のインタフェースで管理するというもの。

2.3 4.4BSD 以降

日本には 1993 年の 9 月に 4.4BSD のテープが到着した。 4.4BSD-alpha の配布手数料は $2000 で、この時 $400 の追加料金を払ったユーザに対しては、自動的に最終リリースのテープが送られてきた。テープには 3 種類あり、HP-300, SparcStation, DECstation のバイナリの中から好きなものを選択することができる。

WIDE Project でも、リリースの到着と同時に 4.4BSD の試用が何ヶ所かで行われているが、十分実用に耐えるという印象を得ている。やはりプラットフォームとしては SparcStation が多く、4.4BSD が Sparc をサポートした影響は非常に大きい。Sparc のコードに関しては、主に開発を行った LBL の Chris Torek がサポートを続けている[1]。

CSRG による BSD Unix は 4.4BSD で最後になるが、NET/2 を基にした OS が非常に活発に活動を続けている。主なものは次の 4 つである。

- 386BSD
- NetBSD
- FreeBSD
- BSD/386

このうち Willam Jolitz の 386BSD は、最近動きが停滞しているようだが、NetBSD, FreeBSD の方は活発に活動を続けているし、BSDI の商品である BSD/386 は独自に開発を行っている。

NET/2 を基にしたフリーの OS としては、現在のところ NetBSD がもっとも進んでいる。NetBSD は、現在 HP, i386, pmax, sparc という 4.4BSD でサポートしているアーキテクチャの他、Amiga, Macintosh, Sun3, ns32532 などでも動作する。

元々は 386BSD のパッチキットを統合する形で始まった NetBSD であるが、現在は次のような NET/2 にはない機能をサポートしている。

PCFS MS-DOS ファイルシステム

BSDI バイナリ互換 BSDI の BSD/386 とのバイナリ互換性

NIS サポート Sun の NIS サポート (現在はクライアントのみ)

loadable kernel module カーネルモジュールの動的リンク

Wine Windows のバイナリ実行環境

共有ライブラリ SunOS4.X と同様の実装方法

[1] ftp.ee.lbl.gov から anonymous ftp 可能である。

リリースされている NetBSD は現在バージョン 0.9 が最新で、開発中のものは NetBSD-current と呼ばれている。これも sup で公開しているため、Internet にアクセスできれば誰でも手に入れることができる。最近ではコンパイルされたバイナリーのスナップショットも定期的に登録されている。また current とは別に magnum というブランチが存在し、より 4.4BSD を統合する方向での開発が進んでいるらしい。

2.4　新 4.4BSD-Lite

4.4BSD-alpha のころから、リリースしたコードのうちフリーにできる部分を NET/2 と同様な形で 4.4BSD-Lite という名前でリリースすることを CSRG はアナウンスしていた。しかし USL との訴訟問題がありこれまで配布されていなかった。

1994 年 2 月 USL と BSDI 社、USL とカリフォルニア大学との訴訟は解決した。合意の内容は次のとおりである。

- NET/2 と現行の 4.4BSD-Lite には、USL の Copyright 含まれているファイルが存在していることに合意。
- UCB は新しい 4.4BSD-Lite を開発し、該当するファイルを新しいものに置き換える。
- BSDI 社は UCB の新しい 4.4BSD-Lite をベースに BSD/386 を開発する。
- 現行の BSD/386(Version 1.1) は NET/2 をベースとしているため、該当するファイルをバイナリで配布する。これは BSD/386 1.1 の出荷に対する限定された措置である。

これによって、UCB は新しい 4.4BSD-Lite を配布可能となった。そして 1994 年 4 月、新しい 4.4BSD-Lite の配布開始が NetNews 等を通じてアナウンスされた。

4.4BSD-Lite は NET/2 がそうであったように、他のフリーな BSD に由来する OS たちに大きな影響を与えることは間違いない。すでに次のリリースから 4.4BSD-Lite を元に開発することを表明しているものも存在している。今後の展開が楽しみである。

2.5　BSD Unix の持つ意味

BSD Unix は、多くの大学や研究機関において、長い間研究開発のプラットフォームとして利用されきた。もちろん、それは BSD がオペレーティングシステムとして優れた機能を持っていたからであるが、それ以外に、すべてのユーザがシステム全体に関するすべてのソースを持ち、それを自由に改変し、情報交換を行うことが可能であったという点は忘れてはならない。多くのユーザの研究成果は、CSRG によって正式リリースに取り入れられ、BSD 自体を改良するために貢献するという再生産の機構がうまく働いていたのである。

しかし、4.2BSD のリリース後、BSD をサポートするワークステーションが市場に現れてきた。これらは BSD がサポートする VAX-11 よりも安価で高性能であったため、ワー

クステーションを開発の対象として使うユーザの数が圧倒的に多くなった。また、メーカが独自にサポートした機能にも優れたものがあり、もはや通常の BSD Unix には戻れなくなってしまった部分もある。たとえば Sun Microsystems の NFS (Network File System) などがそれである。しかし、メーカのサポートする Unix にはソースコードは当然付いていない。Unix ワークステーションの普及は、Unix 文化を広めるという意味では成果を上げたが、反面、すべてのユーザがソースコードを共有するという Unix の大きな特長を失わせてしまったのである。

WIDE Project で 4.4BSD のサポートをはじめたのには、このような背景があった。現在 Unix 利用して教育に携わる人々の多くは、 OS やシステムコマンドのソースコードを参考にしながらプログラミングを学習したに違いない。そして、現在の教育環境に戸惑いと疑問を抱く人も少なくないだろう。4.4BSD や Mach などは教育環境に再び『ソースのある Unix』を再現するという点で、非常に意味のある実用オペレーティングシステムである。なかでも BSD Unix は、pdp-11 や VAX-11 上で教育・研究機関で広く使われてきた馴染みの深いものであり、現在普及しているワークステーションにも BSD Unix に由来する OS を搭載しているものは多い。多くのアーキテクチャをサポートする 4.4BSD によって、再び開かれた Unix が手に入れられる可能性が高くなる。日常使っている環境自体を使って学習できることによる利点は多い。

今日、ネットワークニュースから流れてくる記事を眺めていると、「パスワードを入力するときに画面に表示させないためにはどうするのですか?」といった質問が少なくない。以前であれば、このような疑問を抱いたユーザは、間違いなく passwd コマンドのソースを見つけ出し、stty によってそれが解決することをたちどころに探り当てたはずだ。商業 Unix の普及はこのような優れた教育環境をユーザから奪い、代わりに膨大な量の関連書籍やマニュアルを生み出した。ユーザは多すぎる参考文献の中から自分の探す答えを見つけ出さなければならない。4.4BSD-Lite のリリースは、このような状況を変えるための一石を投じるはずだ。

第 3 章

携帯型計算機における UNIX の利用 (1)

高性能 CPU を搭載した IBM-PC 互換機のノートブックタイプに代表されるような携帯型計算機の登場により、UNIX の運用形態も従来の固定的なものから移動しながらの運用や、バッテリーによる運用などを考慮する必要がでてきた。

携帯型計算機で UNIX を運用する場合問題となるのが、移動時における計算機の状態の保存方法と節電機能のサポートである。一般に UNIX オペレーティングシステムは、起動や停止に時間がかかる場合が多い。また移動するたびにシステムを停止していたのでは連続運用が必要なプログラムなどを動作させることができず、非常に不便である。

そのため携帯型の計算機で運用する場合、必要に応じてシステムの現在の状態を保持したまま休止したり休止以前の状態に戻す機能が必要となる。これは通称レジューム機能と呼ばれており、これによってシステムの連続運用性を確保することが可能となっている。

3.1 レジューム機能について

多くのノートブックタイプの IBM-PC 互換機では、レジューム機能を搭載している。従来レジューム機能は、特別なハードウェアとオペレーティングシステムによるサポートが必要なため機種依存性が高いものであった。しかし最近では、CPU 自体にレジューム機能を用意したものが登場している。[1]

CPU の持つレジューム機能は、オペレーティングシステム及びアプリケーション・ソフトウエアからは完全に見えないようにしてシステム動作を管理する強力な機能を備えている。i486SL シリーズ (i486 Enhanced Series) は、既存のラップトップなどに見られる電源管理機能実装で見られるような不備のない、最大限の電力削減をインテリジェント型電源管理ハードウエアを用いて実現している [249][250]。

そのような電源管理機能を持つ CPU セットを利用したノートブックに UNIX を導入する場合、オペレーティングシステムで機種依存性のある特殊なコードを用意することなくレジューム機能を実装することが可能となるはずである。

[1] 例えば東芝のダイナブックシリーズに搭載されている CPU にも、このような機能が一部内蔵されている

3.2 設計

レジュームは次のような場合に実行される。

- ソフトウエアでサスペンドレジスタに書き込みを行った場合
- 外部装置で外部 SMI(System Management Interrupt) 入力ピンをアサート
- グローバル・スタンバイの自動電源切断オプションを使用する
- ノートブックが用意しているレジュームボタンが押された場合
- バッテリー電圧低下警告を受理した場合

レジュームは、アプリケーションが実行していた情報・メモリイメージを失うことなく再始動できる最低電力を確保したうえで動作する。この場合でも基本的にメモリだけには電力が送られているため、メモリによるバッテリーの消耗は免れない。

サスペンドが実際に起動される前には、システムが DMA などの入出力操作を終了出来るようにサスペンドトリガとサスペンド状態の開始の間にプログラムされた遅延を挿入してある。そこでサスペンドが起こった時点でメモリーのイメージをハードディスクの一部に書きこむことで、メモリによるバッテリーの消耗がない完全なレジュームをサポートすることも可能である。

レジューム機能を UNIX に導入する上で問題となったのが、機種非依存性をどのようにして保つかということである。そのため最低条件として、CPU をレジューム機能を標準で搭載している i486SL(i486SX Enhanced Series) であることとした。

ノートブックがレジュームを行なう際、必ず、SMI が発行される。SMI は、最高優先度のノンマスカブル割り込み (Non Maskable Interrupt — NMI) である。SMI が発行されると、CPU は次のような動作を行なう。

- System Management RAM(SMRAM) を 30000H から 3FFFFH の範囲のメインメモリ空間にマッピングを行なう。SMRAM は、この領域にある既存のメインメモリをオーバーレイし、内部 IO トラップ FILO(System Management FILO) を凍結する。
- SMRAM 空間に CPU の状態を記憶する。このデータは 3FFFFH から格納され、スタック方式で下位アドレスに記録される。
- システム管理モード (System Management Mode) に移行する。
- システムアドレスの 38000H の実装トリガが、プログラムしたシステム管理モードハンドラの実行を開始する。これは、特殊 SMRAM 領域に配置される。
- CPU がシステム管理モードの実行に移り、CPU コアのセレクタが初期化される。

UNIX が上記状態を認知し、自動的にレジューム処理を行なうことによって標準化が可能となる。

まず、レジュームを UNIX 側でいかに認識させるかが課題となる。例えばダイナブックでは、電源ボタンを押すことで SMI インタラプトが入る仕組みになっている。この電源ボタンはチップセット上のサスペンド・レジュームボタンピンの機能を利用している。このピンはトグル式で、システムがサスペンド状態でなければ、サスペンド・レジュームボタンピンをアサートすることでレジューム機能を開始し、システムがサスペンド状態であるならば、サスペンド・レジュームピンをデアサートすることで、サスペンド状態からの解放が可能となっている。そのため、電源スイッチによってレジュームを動作させる場合、オペレーティングシステム側の処理を確実にサスペンド状態へと移行することが困難であった。

3.3　実装

実装は BSD/386 上で行った、BSD/386 は BSDI 社の開発した POSIX 互換のオペレーティングシステムで University of California Berkeley (UCB) の CSRG (Computer Systems Research Group) の開発した BSD Networking Software, Release 2(NET/2) を基に開発されたものである。BSD/386 はカーネルを含むシステムソフトウェアの殆どがソースコードを含めて入手可能であり、カーネルレベルでの変更が必要になる今回のような実装を行うのに最適なシステムの一つだと考えられる。また対象マシンとしては東芝のダイナブックを用いた。

3.3.1　auto-shutdown

まず、ノートブックを `shutdown` したときに自動的に電源を切る機能を加えた。具体的には、`shutdown` コマンドを実行し、カーネルが `shutdown` された時に、自動的に電源を切るコードを書き加えた (図 3.1) 。

これによって、シャットダウン時にも電源キーを触れることなく自動的に電源を切ることが可能となった。

図 3.2に見られるように `autopowerdown()` は機種依存となっている。これはマシンによって電源を切るアドレスが異なるためである。このため機種を自動的に判別し、相応するアドレスを自動的に認知するようにしている。

3.3.2　resume-key

キーボード上の特殊キー (ホットキー) の組合せによってレジューム機能を立ち上がらせる方法をカーネル上に構築した。この場合、ほぼ、完全な形で機種非依存性を確保できる。特殊キーの組合せによって、ハードディスクの `sync` を行ない、 SMI を起動し、レジュームを実行させ、電源を自動的に切る。図 3.3にそのアルゴリズムを示す。

ここでは `CTRL-ALT-DEL` のキーを同時に押した時にサスペンドもしくは、システムハングが生じるようにカーネルの変更を行っている。すでに `CTRL-ALT-DEL` によるシステムハ

```
  :
  :
  doatshutdown();
    if (システムが完全にシャットダウンされた。) {
       printf(``\nSystem is halted; %s\n\n'',
       ``hit reset, turn power off, or press return to
       reboot'');
#ifdef NOTEBOOK /* ノートブックの場合 */
       autopowerdown();
#endif /* NOTEBOOK */
```

図 3.1: ブートアルゴリズムに追加したルーチン

ングのルーチンがカーネルに装備されているため、この部分に改良を加え、CTRL-ALT-DEL キーを入力すると、カーネルが、システムダンプを行なうか、レジュームを行なうかの選択をするようにした。

メモリーの持つイメージをファイルとしてダンプするノートブックの場合、MSDOS のパーティションをダンプエリアとして確保しなければならない場合がある。現状のダイナブックはメモリに電力を与え、記憶確保を行なうタイプなので、その必要はない。上記タイプのノートブックの場合、実際のメモリーイメージダンプ作業は、ノートブック内の BIOS が制御しているため、機種依存性が生じる。resumestart() 実行の際にも、機種を特定し、機種に応じたサスペンド機能のルーチンを起動しなければならなくなる。

3.3.3 レジュームによる時計の遅れ

レジューム機能によってシステムが停止している間、UNIX の用意している内部時計は CPU 自体が完全に止まってしまうため進まない。これに対しハードウェアクロックはレジューム時も動作しているため、システムをレジュームから復帰したときにハードウェアクロックと UNIX の内部時計の間で時間のずれが生じてしまう場合がある。そのため、レジューム復帰時に UNIX の内部時計をハードウエアクロックと同期させる必要がある。

ここでは復帰時に clockadj() ルーチンを呼び出すことによって、ハードウエアクロックと内部時計の同期を行なっている。(図 3.2の最後の部分)。clockadj() の構造自体はは極めて単純なもので、ハードウエアクロックのデータをそのまま代入し直しているだけである。

3.4 評価

このシステムでは運転状態からレジューム状態に移るまで、平均 3 秒程の時間しかかからない。レジューム状態からの復帰には 1 秒程度かかる。表 3.1にレジューム機能を持たせたシステム起動時間と shutdown の時間の効用を示す。

```
autoshutdown()
{
   switch ( notebookid() ) {
    /* notebook id は、各ノートブック独特の ROM イメージから、機種判定
     * を行ない、機種名を返すルーチンである。
     */
        case DYNABOOKA: /* ダイナブック タイプ A の場合 */
          outw(0x0e8, 0x9a5a);
          break;
        case DYNABOOKB:
            :
            :  /* 機種によって、電源 OFF を行なうルーチンは異なる。*/
        case NOTYPE: /* 機種が特定できない */
          error(機種が特定できない);
          break;
       }
    /* レジュームした時はここより下はいかないはず! */
     sleep(3);
      clockadj(); /* レジューム復帰時にクロックを戻す */
}
```

図 3.2: autoshutdown のアルゴリズム

これに対しレジューム機能を使用しない場合、常に shutdown と、ブートという作業を繰り返すため、起動しているアプリケーションも中断し、再起動しなければならなくなる。 また shutdown には 40 秒、起動には 110 秒の時間が無駄となり、ノートブックの連続運用性が確保されない。さらにシステムへのログインを行ない、アプリケーションの再起動を行なわなければならなくなるため、実質的な時間の無駄がさらに広がる。本研究によりレジューム機能を UNIX で利用できるようになり、ノートブックの連続運用性が確保された。

一方レジューム時のクロック補正は、現在は単純なクロック補正機能によって、解決している。しかし、実際には復帰時にクロック補正を行うためレジューム時間内に動作す

表 3.1: レジューム機能の効用

	システム起動時間	shutdown 時間
レジューム機能あり	1 秒	3 秒
レジューム機能なし	110 秒	40 秒

```
pccons.c /* カーネル上のキーボードデバイス処理部 */
    case RESUMEHOTKEY: /* レジューム のホットキーが入力された場合 */
      resumestart();
      autoshutdown();
        /* ここまではこない */
      break;
```

図 3.3: 特殊キーの組合せによるレジューム

るはずの cron などで時間設定されたプロセスに対しては、それらの動作が無効となってしまう場合があるため、変更を行なう必要があるだろう。また 厳密な意味では、プロセステーブルの実行時間などに影響が現れてしまうため、根本的な UNIX システムの構成を変更しなくてはならなくなる。そのため UNIX の持つ時間軸をゆがめる機能が必要となる。

具体的解決案として、レジューム復帰時に UNIX の持つ時計をハードウエアクロックの時間まで早送りすることによって、レジューム期間中の at ジョブ、cron ジョブの実行を行なうルーチンを作成する必要がある。

また、プロセステーブルに関していえば、resume_time という新たな変数を置くことで、レジューム状態で UNIX システム全体が止まっていた時間を報告するシステムを構築する必要がある。

3.5 まとめ

UNIX でレジューム機能をサポートするために、レジュームを行うキーを対応させ、オペレーティングシステムが、レジューム状態を感知できるよう工夫した。また、シャットダウン時にも自動的に本体の電源が切れるようにするといった配慮を行った。

レジュームによる時計の遅れは、ハードウエアクロックとの動機をとるという安易な解決で終ってしまったが、今後、早送り時計などの開発も必要となるであろう。

第 4 章

携帯型計算機における UNIX の利用 (2)

携帯型計算機においてバッテリーによる期待稼働時間を増加させる方法として、利用していない時は、使っていない部分 (LCD のバックライトや DISK 等) の電源を極力切ることで電力消費を押えていく方法がある。また CPU 自体も内部の動作クロックを止めたり、通常動作時に比べて外部から入力するクロックの周波数を変更することなく内部周波数を下げて、消費電力の低減を計ることが可能なものが存在している。これらは通称、節電機能と呼ばれている。

4.1 節電機能について

節電機能を利用するには計算機が利用されてない (アイドル) 状態であることをシステムが認識できる機構が必要となる。これには通常オペレーティングシステム側のサポートが不可欠である。

MSDOS ではシステムがアイドル状態であるかどうかを検出する手法として、`power.exe` という、デバイスドライバを利用している。`power.exe` はオペレーティングシステムや BIOS に発する割り込みなどの発生状況を調べる。この結果パソコンがアイドル状態だと判断すると待機状態にはいっても構わないことを MSDOS の電源管理インターフェースである APM インターフェースを介して直接制御 BIOS に知らせることで節電機能を利用可能にする。

これに対し UNIX では、BIOS を介さずに、直接ハードウエアを制御しているため、MSDOS と同様の検出方法を用いることはできない。そこで別の方法でアイドル状態を検出する方法を考えた。

4.2 実装

本研究では、節電ユーティリティとして次のようなコマンドを用意した。ここでは CPU のアイドル状態の検出に X サーバが標準で用意しているスクリーンセーバを用いている。これは厳密な意味でアイドル状態の検出になっていない部分もあるが携帯型計算機の実際の利用形態を考えた場合、十分実用できであると判断した。

4.2.1 xsetsuden

xsetsuden は、X ウインドウ上で動作するユーティリティである。xsetsuden は、X ウインドウ実行時に起動する X ウインドウのスクリーンセーバと連動し、液晶のバックライトの消灯、液晶そのものの電源カット、ハードディスクのアイドルモード、CPU スリープ を行なう。X サーバーに改良をし、X サーバー内の関数 XSetScreenSaver のタイムアウトによって、節電機能を作動させる [251]。

xsetsuden によって動作する節電機能は次のような流れで行なわれる。

- X サーバーのスクリーンセーバが起動する。
- 同時に液晶のバックライトを消す。
- スクリーンセーバが起動してから 30 秒後にハードディスクを sync、ハードディスクをアイドルモードにする。(ハードディスクのモータを止める)
- ハードディスクの電源を切ってから 30 秒後に、CPU をスタンバイモードに移行させる。

xsetsuden の起動間隔は、xset コマンドによって設定できる。

4.2.2 backlight

backlight コマンドで任意にバックライトの消灯・点灯を行うことができる。日常の明るさではバックライトを点灯しなくても、液晶の表示が判読可能な場合が多い。このような時は、バックライトコマンドでバックライトの点灯、消灯を行なうことができる。バックライトを消灯した場合、画面はかなり見辛くなるが、稼働時間は劇的に増える事になる。

4.2.3 評価

xsetsuden のタイムアウトを 30 秒で設定し、実際にバッテリー警告灯が点灯するまでノートブックで日常作業を行ってみた (本論文の執筆作業を行った)。通常はバッテリーの充電から 110 分程度の作業が可能だが、xsetsuden による節電効果によって、145 分まで作業時間を伸ばす事ができた。この実験では、作業体系は、ほぼ連続していて、あまりアイドル時間が増えなかったので、稼働時間を効果的に伸ばす事が出来なかったが、節電機能は有効であるというのがわかる。

xsetsuden は X ウインドウ上でしか動作しないため、コンソール状態での節電ユーティリティには対応しない。また根本的な UNIX オペレーティングシステムに節電機能を採り入れているわけではないので、その点については改良を行なう必要がある。

また backlight コマンドによって、110 分程度だった稼働時間を 260 分まで伸ばす事が出来た。バックライトによる電力消費を抑えることによって、ノートブックの稼働時間を大幅に増やす事が可能であるというのがわかる。

4.3　まとめ

節電ユーティリティを X ウインドウに対応させる事で、バッテリーの有効利用が出来るようになった。またユーザが意識することなく節電が行えるよう配慮した。また携帯型計算機での節電に限らず、固定的に運用する通常のワークステーションでも節電機能は重要である。最近は計算機システムもエコロジーへの配慮が必要という意見があり、現にアメリカではエナジースターと呼ばれる節電のための規格がすでに存在している。UNIX もこれに対応する必要が迫られるだろう。

現在、節電機能については、まだ UNIX システムそのものがそれに基づいて考えられていない。今後、新しいシステムを構築する上で、カーネルレベルでの節電—例えば、ハードディスクを出来る限りアクセスしないシステムの開発— 等が重要であると考えられる。そのためには機器の抽象化とオペレーティングシステムとの融合についても考慮する必要があるだろう。

第 5 章

国際化対応マルチバイト文字処理関数ライブラリの開発

今日、ソフトウエアの国際化は大きな課題の一つとなっている。しかし世界中で使用されている文字コードにはさまざまなエンコーディングがある (日本語だけ考えてもロックシフトを用いた JIS コード、シングルシフトを用いた EUC、「すき間」を利用したシフト JIS、さらには近い将来 JIS 化されそうな Unicode がある)。これらすべてに対応するようにプログラミングすることは困難である。

これに対し、多くのシステムでは複数のエンコーディングに対応するマルチバイト文字処理のためのライブラリが提供されている。しかし多くは対応しているマルチバイト文字コードは EUC などごく一部に限られ、ステートフルの文字コードに対応しているものはまれである。またソースが公開されておらず、システムごとのライブラリの仕様の相違が大きいため、国際化対応アプリケーションの移植性を悪いものにしている。

以上のような現状をふまえ、

- 現在使われているほとんどの文字コードに対応できる柔軟性がある
- 特定のシステムに依存しない
- フリーソフトウエアとして公開できる

という基本的要求を満たすことのできるマルチバイト文字処理関数ライブラリを開発した。

5.1 仕様

実装したライブラリ関数は ISO C マルチバイト拡張仕様案 [252] (以下 MSE と略す。MSE:Multibyte Support Extension) に従うことにした。MSE の特徴として以下の点があげられる。

- ANSI C 標準ライブラリ関数とほぼ同様のインタフェース
- ステートフルエンコーディングのマルチバイト文字の処理に対応できるように、文字列やストリームごとのステートを保存する方法が規定されている。
- 文字分類/変換関数 (標準ライブラリでいう is*、to*) は分類/変換クラスが拡張可能になっている。

5.2　設計

5.2.1　基本構成

ライブラリを構成する関数群を以下のように分類する。

setlocale 関数 環境変数などから現在のロケールを取得し、ロケール情報 (このライブラリでは文字コード、文字分類に関する情報) をファイルからロードする。

ワイド 文字入出力関数群 動作はロケール情報に基づいて行われる。入力関数はマルチバイト文字のストリームからバイト列を読み込み、ワイド 文字に変換して格納する。出力関数はワイド 文字をマルチバイト文字列に変換してストリームに書き出す。

ワイド 文字処理関数群 標準ライブラリの ctype, string 関数のワイド 文字対応版。ctype 関数はロケール情報に基づいて動作を変える。

これらの関数の関係を図 5.1に示す。

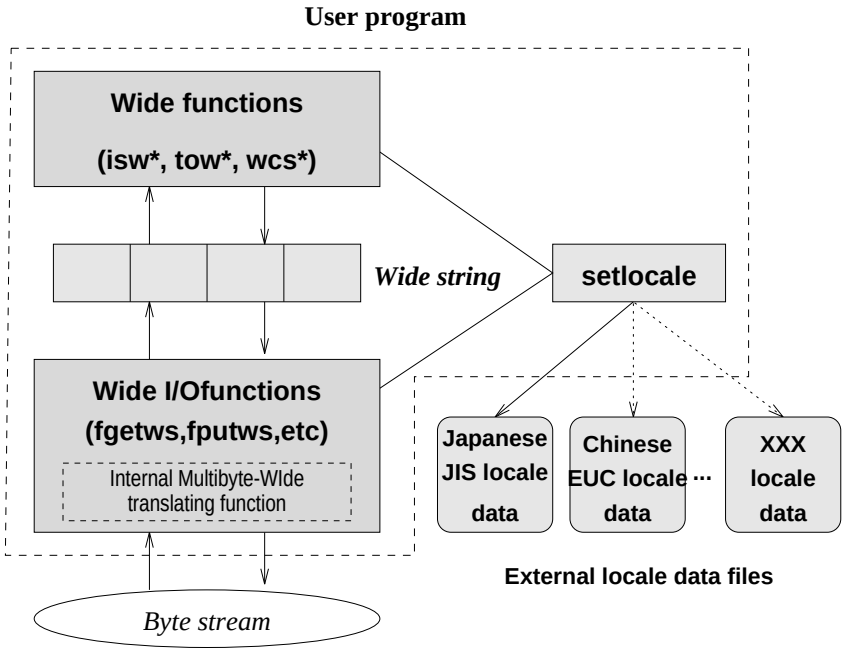

図 5.1: 国際化対応文字処理ライブラリ関数の階層

5.2.2　マルチバイト 文字コード

既存の文字コードは次のようなエンコーディング方式に分類することができる。

ステートレスなもの EUC、S-JIS、BIG-5 など

iso2022 ロックシフトに基づいたステートフルなもの JIS、ISO-2022-JP など

iso10646/unicode に基づいたもの UTF-2 など

iso10646[253]/unicode[254] をそのままサポートすることはあきらめた。なぜならこれらは現在の C 言語の仕様にはなじまないからである。c 言語では文字はバイトの列からなり、値が 0 のバイトはヌルバイトで文字列の終端を表すことになっているが、これらは 1 文字 16 ビット (または 32 ビット) であり、これを 2 バイト (または 4 バイト) からなるマルチバイト文字であると考えるべきではない。例えば 'a' は unicode では 16 進数で 0041 という値となるが、これでは文字列の途中でヌルバイトが入ってしまうことになる。

5.2.3 ワイド文字コード

ワイド文字コードは以下の条件を満たしていることが望ましいと考えられる。

- ステートレス
- 1 文字の長さが固定
- サポートするすべてのマルチバイト文字コードのスーパーセット

現段階でこれらの条件を最も満たしている文字コードとして、本研究では icode[255] を選択した。これは iso10646/unicode を基にしたものだが日中韓の漢字の区別のためのビットが付加されている。icode は結合文字が存在するため 1 文字固定長とはいえない面もあるが、結合文字を使用することは少なく、文字処理に影響することはあまりないだろうと考えた。

5.3 実装

実装は BSD/386 で行った。BSD/386 は BSDI 社の開発した POSIX 互換のオペレーティングシステムで University of California Berkeley (UCB) の CSRG (Computer Systems Research Group) の開発した BSD Networking Software, Release 2(NET/2) を基に開発されたものである。BSD/386 はカーネルを含むシステムソフトウェアの殆どがソースコードを含めて入手可能であり、今回実装を行うのに最適なシステムの一つだと考えられる。

5.3.1 マルチバイト↔ワイド文字変換

このライブラリでもっとも重要な部分は入出力関数、拡張マルチバイト/ワイド文字変換関数内部で用いられるマルチバイト↔ワイド文字変換ルーチンである。

変換アルゴリズムとしては大きく分けて

1. 一定の演算やビット操作によって変換する方法
2. マルチバイト文字とワイド文字とのマッピングテーブルによって変換する方法

が考えられるが、このライブラリの場合ワイド文字コードが icode のため EUC などとはコードの並びが全く異なる。そのため演算やビット操作による方法では効率よく変換することができない。それでマッピングテーブルによる方法を用いることにした。マッピングテーブルは各ロケールのロケール定義ファイルに記述され、setlocale 関数の呼び出しで読み込まれる。

マッピングテーブルを用いる場合、メモリ消費が問題となる。2 バイトコードの値をそのままテーブルのエントリ番号 (つまり配列の添字) として与えるとすると、$2^{16} = 65536$ エントリが必要になる。ワイド文字 (wchar_t) は 4 バイト長なので、1 つのテーブル全体の大きさは 65536×4 バイト=256K バイトとなる (マルチバイト→ワイド変換とワイド→マルチバイト変換の 2 つのテーブルが必要である)。さらにマルチバイト文字が 3 バイトコードの場合は 64 メガバイト、4 バイトコードなら 4 ギガバイトとなる。ここまでいくととても現実的な値ではなくなる。

しかし実際にはコードのすべての部分が使われているわけではない。例えば日本語 EUC コードは最大 3 バイトであるが、文字が割り当てられている部分を各バイトをそれぞれ xyz 軸に割り当てて 3 次元の空間として表現するとほとんどが空きであることが分かる (図 5.2)。

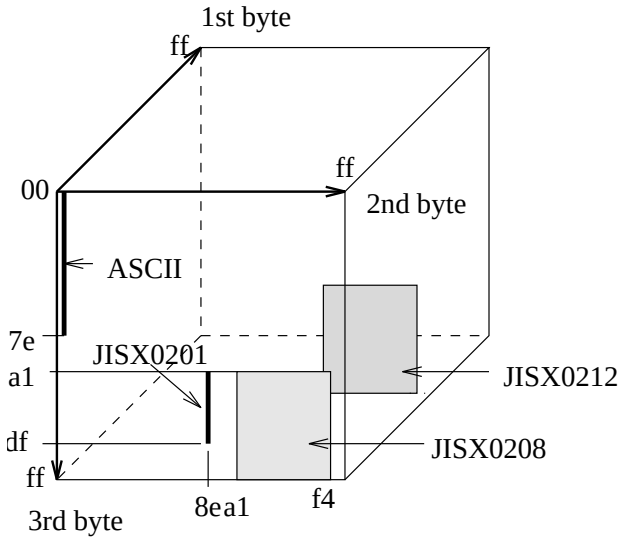

図 5.2: 日本語 EUC のコード割り当て空間

それでテーブルは文字が連続的に割り当てられている範囲ごとに作成することにした。このようにすると日本語 EUC のマルチバイト→ワイド文字変換テーブルの大きさは 60K バイト弱となった。また compoundtext の場合でも約 110K バイト、UTF-2 だと約 118K バイトになった。

ワイド→マルチバイト文字変換テーブルもほぼ同様の考え方で作成することにしたが、ワイド文字コード (icode) ではもともと unicode で日中韓の漢字が統合されて割り当てら

れていたものに日中韓の識別ビットをつけてそれぞれの国の漢字を判別するようになっている。そのため unicde では日本の漢字にはなくても中国や韓国の漢字には存在する文字に割り当てられているコードが、icode で日本の漢字が割り当てられている部分では空きになってしまう。このように不連続な空きが多いため、上手にテーブルを分割することができなかった。そのためワイド→マルチバイト文字変換テーブルはマルチバイト文字コードに日中韓のいずれかの漢字コードが含まれる場合、200 K バイトを越える大きさになってしまった。これを解決するためにはワイド文字コードとテーブルのデータ構造の変更が必要である。

5.3.2 ワイド入出力関数

入出力関数は、内部でマルチバイト→ワイド文字変換を行なうとともに、ストリームごとのステートを保存しなければならない。ステートの保存のために mbstate_t という新しい型が定義されている。ストリームごとにステートを保存するためには file 構造体に mbstate_t 型のメンバを付け加えるのがもっともスマートな方法だと考えられるが、こうすると標準ライブラリ関数にまで影響が及ぶのでファイルディスクリプタ番号をエントリ番号にとる、ステート保存用の別のテーブルを用意した。

5.3.3 ワイド文字分類・変換関数

MSE では文字分類・変換クラスの拡張ができるように、wctype、iswctype、wctrans、towctrans という 4 つの関数を定義している。これらは例えば以下のように用いることができる。

```
wchar_t wc, wc2;
int n;

n = iswctype(wc, wctype("hiragana")); /* wc がひらがなだったら真 */
/* wc がひらがなだったら、wc2 にはカタカナが入る */
wc2 = towctrans(wc, wctrans("tokatakana"));
```

この例の “hiragana”、“tokatakana” というクラス名とその対象となるコードの範囲、変換先はロケール定義ファイルに記述される。

5.3.4 ワイド文字列処理関数

ワイド文字列処理関数はロケールには影響されない (wcscoll、wcsxfrm、wcsftime を除く) ので標準ライブラリの関数の名前を str*だったものを wcs*に変え、char 型だった引数の型をワイド文字型 (wchar_t) に変えるだけでよい (実際にはスクラッチから書いた)。

5.3.5　ロケール定義ファイル

文字コードに依存する情報はほとんどロケール定義ファイルに記述することができる。現在用意されているのは 5.3.6節にあげたものだけだが、ロケール定義ファイルを用意することによって新たな文字コードに対応させることができる。

ロケール定義ファイルは${LOCALEDBPATH}/{ロケール名}/LC_CTYPE というファイル名になる。$LOCALEDBPATH はデフォルトでは/usr/local/lib/locale である。ロケール定義ファイルには以下のような情報を記述する。

- エンコーディングはステートフルかステートレスか
- ステートフルの場合はエスケープシーケンスに関する情報
- コードセットごとのコード範囲と 1 文字のバイト数
- マルチバイト↔ワイド文字変換マッピングテーブル
- 文字分類・変換クラス名とそのクラスに関する情報

ロケール定義ファイルはテキスト形式で記述できるが、読み込みの高速化のため、バイナリ形式に変換しておくことにした。そのために mklcctype というユーティリティを用意した。

5.3.6　対応できた文字コード

現在ロケールデータが作成済みのマルチバイト文字コードは以下のとおりである。

ロケール名	文字コード
ja_jp.jis7	日本語 7 ビット jis
ja_jp.eucjp	日本語 EUC
zh_cn.gb2312	中国語 gb2312
zh_cn.eucch	中国語 EUC
ko_kr.ksc5601	韓国語 ksc5601
ko_kr.euckr	韓国語 EUC
ja_jp.utf2	UTF-2

また compoundtext を使うロケールデータもほぼ完成している

5.3.7　用意されている関数

MSE に規定されている関数のうち、図 5.1、5.2、5.3のものが実装済みである。また現時点では図 5.4のものはまだ未実装である。

表 5.1: 実装済みの関数–1

ヘッダ<iso646.h> 代用スペル	
マクロ	意味
and	&& の代用
and_eq	&= の代用
bitand	& の代用
bitor	| の代用
compl	~ の代用
not	! の代用
not_eq	!= の代用
or	|| の代用
or_eq	|= の代用
xor	^ の代用
xor_eq	^= の代用
ヘッダ<wctype.h> ワイド文字分類、変換関数群	
型	意味
wctrans_t	ロケール特有の文字変換を表現
wctype_t	ロケール特有の文字分類を表現
wint_t	WEOF を保持できる型
マクロ	意味
WEOF	end-of-file
関数	機能
int iswalnum(wint_t wc);	isalnum のワイド文字版
int iswalpha(wint_t wc);	isalpha のワイド文字版
int iswcntrl(wint_t wc);	iscntrl のワイド文字版
int iswdigit(wint_t wc);	isdigit のワイド文字版
int iswgraph(wint_t wc);	isgraph のワイド文字版
int iswlower(wint_t wc);	islower のワイド文字版
int iswprint(wint_t wc);	isprint のワイド文字版
int iswpunct(wint_t wc);	ispunct のワイド文字版
int iswspace(wint_t wc);	isspace のワイド文字版
int iswupper(wint_t wc);	isupper のワイド文字版
int iswxdigit(wint_t wc);	isxdigit のワイド文字版
wctype_t wctype(const char *property);	ワイド文字分類クラスの値を返す
int iswctype(wint_t wc, wctype_t desc);	拡張ワイド文字分類関数
wint_t towlower(wint_t wc);	tolower のワイド文字版
wint_t towupper(wint_t wc);	toupper のワイド文字版
wctrans_t wctrans(const char *property);	ワイド文字変換クラスの値を返す
wint_t towctrans(wint_t wc, desc);	拡張ワイド文字変換関数

表 5.2: 実装済みの関数–2

ヘッダ <wchar.h> 拡張マルチバイト /ワイド 文字関数群	
型	意味
mbstate_t	変換ステートの保持
マクロ	意味
WCHAR_MAX	wchar_t の最大値
WCHAR_MIN	wchar_t の最小値
関数	機能
・ワイド 文字入出力関数	
wint_t fgetwc (FILE *stream);	fgetc のワイド 文字版
wchar_t *fgetws (wchar_t *s, int n, FILE *stream);	fgets のワイド 文字版
wint_t fputwc (wint_t c, FILE *stream);	fputc のワイド 文字版
int fputws (const wchar_t *s, FILE *stream);	fputs のワイド 文字版
wint_t getwc (FILE *stream);	getc のワイド 文字版
wint_t getwchar(void);	getchar のワイド 文字版
wint_t putwc (wint_t c, FILE *stream);	putc のワイド 文字版
wint_t putwchar(wint_t c);	putchar のワイド 文字版
wint_t ungetwc (wint_t c, FILE *stream);	ungetc のワイド 文字版
int fwide (FILE *stream, int mode);	ストリームのオリエンテーション変更
・拡張マルチバイト/ワイド 文字変換関数	
wint_t btowc (int c);	1 バイトをワイド 文字へ変換
int wctob (wint_t c);	ワイド 文字を 1 バイト文字へ変換
int mbsinit (const mbstate_t *ps);	ps が初期状態か調べる
size_t mbrlen (const char *s, size_t n, mbstate_t *ps);	mblen の “restartable” 版
size_t mbrtowc (wchar_t *pwc, const char *s, size_t n, mbstate_t *ps);	mbtowc の “restartable” 版
size_t wcrtomb (char *s, wchar_t wc, mbstate_t *ps);	mctomb の “restartable” 版
size_t mbsrtowcs(wchar_t *dst, const char **src, size_t len, mbstate_t *ps);	mbstowcs の “restartable” 版
size_t wcsrtombs(char *dst, const wchar_t **src, size_t len, mbstate_t *ps);	wcstombs の “restartable” 版

5.4 問題点

- ロケールデータの保持のためのメモリ消費量が大きい。JIS コード用で約 280K バイト、UTF-2 用で約 460K バイト消費する。このうちの 70〜80%がワイド→マルチバイト文字変換テーブルに消費されている。
- システム付属のマルチバイト文字処理関数と比べてマルチバイト→ワイド文字変換速度が遅い。これはシステム付属のものがたいていマルチバイト文字コードが EUC、ワイド文字コードも EUC の変形でほとんど変換に手間がかからないのに対し、このライブラリでは変換にかなりの処理が必要なためだと考えられる。
- 対応しきれないマルチバイト文字コードがある。たとえば ISO-2022-INT-1 は GB2312 中国語と CNS11643 中国語の文字コードを区別しているが、ICODE ではこれらは統合されて中国語のコードとして割り当てられているためワイド文字に変換すると区別できなくなってしまう。

5.5 今後の課題

まずまだ実装を終えていない関数の実装と BSD/386 以外のシステムで利用できることの確認を行ない、公開する予定である。ライブラリの次の課題としては以下のようなものがあげられるだろう。

よりよいワイド文字の設計 日本語、中国語、韓国語を扱う場合、無駄が多いことなどを考えるとワイド文字コードの改良が必要である。また結合文字への対応も考慮したい。

アプリケーションへの適用 このライブラリを利用した応用アプリケーションの開発を行い、有用性を検証する必要がある。

高レベルライブラリの国際化 curses や regex、dbm などをこのライブラリに対応可能なように改良する。

コンパイラへの適用 プログラム中にワイド文字定数を利用可能にし、プログラムの可読性を上げる。

表 5.3: 実装済みの関数–3

関数	機能
・ワイド文字列関数	
wchar_t *wcscpy (wchar_t *s1, const wchar_t *s2);	strcpy のワイド文字版
wchar_t *wcsncpy (wchar_t *s1, const wchar_t *s2, size_t n);	strncpy のワイド文字版
wchar_t *wcscat (wchar_t *s1, const wchar_t *s2);	strcat のワイド文字版
wchar_t *wcsncat (wchar_t *s1, const wchar_t *s2, size_t n);	strncat のワイド文字版
int wcscmp (const wchar_t *s1, const wchar_t *s2);	strcmp のワイド文字版
int wcsncmp (const wchar_t *s1, const wchar_t *s2, size_t n);	strncmp のワイド文字版
wchar_t *wcschr (const wchar_t *s, wint_t c);	strchr のワイド文字版
size_t wcscspn (const wchar_t *s1, const wchar_t *s2);	strcspn のワイド文字版
wchar_t *wcspbrk (const wchar_t *s1, const wchar_t *s2);	strpbrk のワイド文字版
wchar_t *wcsrchr (const wchar_t *s, wint_t c);	strrchr のワイド文字版
size_t wcsspn (const wchar_t *s1, const wchar_t *s2);	strspn のワイド文字版
wchar_t *wcsstr (const wchar_t *s1, const wchar_t *s2);	strstr のワイド文字版
wchar_t *wcstok (wchar_t *s1, const wchar_t *s2, wchar_t **ptr);	strtok のワイド文字版
size_t wcslen (const wchar_t *s);	strlen のワイド文字版

表 5.4: 未実装の関数

関数	機能
・ワイド文字入出力関数	
int fwprintf (FILE *stream, const wchar_t *format, ...);	fprintf のワイド文字版
int fwscanf (FILE *stream, const wchar_t *format, ...);	fscanf のワイド文字版
int wprintf (const wchar_t *format, ...);	printf のワイド文字版
int swscanf (const wchar_t *s, const wchar_t *format, ...);	sscanf のワイド文字版
int wscanf (const wchar_t *format, ...);	scanf のワイド文字版
int swprintf (wchar_t *s, size_t n, const wchar_t *format, ...);	sprintf のワイド文字版
int vfwprintf(FILE *stream, const wchar_t *format, va_list arg);	vfprintf のワイド文字版
int vwprintf (const wchar_t *format, va_list arg);	vprintf のワイド文字版
int vswprintf(wchar_t *s, size_t n, const wchar_t *format, va_list);	vsprintf のワイド文字版
・ワイド文字列数値変換関数	
double wcstod (const wchar_t *nptr, wchar_t **endptr);	strtod のワイド文字版
long int wcstol (const wchar_t *nptr, wchar_t **endptr, int base);	strtol のワイド文字版
unsigned long int wcstoul(const wchar_t *nptr, wchar_t **endptr, int base);	strtoul のワイド文字版
・ワイド文字列関数	
int wcscoll (const wchar_t *s1, const wchar_t *s2);	strcoll のワイド文字版
int wcsxfrm (const wchar_t *s1, const wchar_t *s2, size_t n);	strxfrm のワイド文字版
size_t *wmemchr (const wchar_t *s, wchar_t , size_t n);	memchr のワイド文字版
int wmemcmp (const wchar_t *s1, const wchar_t *s2, size_t n);	memcmp のワイド文字版
wchar_t *wmemcpy (wchar_t *s1, const wchar_t *s2, size_t n);	memcpy のワイド文字版
wchar_t *wmemmove(wchar_t *s1, const wchar_t *s2, size_t n);	memmove のワイド文字版
wchar_t *wmemset (wchar_t *s, wchar_t, size_t n);	memset のワイド文字版
size_t wcsftime(wchar_t *s, size_t max, const wchar_t *format, const struct tm *timeptr);	strftime のワイド文字版

第 6 章

新しいコンセプトの OS の紹介

現在、標準 OS として広く普及している UNIX は、仮想メモリ機能、TCP/IP ネットワーク機能などの様々な機能が加えられ発達してきたが、基本設計が 20 年以上も前になされたもののため、新しいハードウェア、ソフトウェアモデルに必ずしも合わなくなってきてきる。

そこで新しいコンセプトにもとづく OS が提案されている。例えば、Sun 社の Spring をはじめとするオブジェクト指向 OS などが研究レベルで提案されている。

今回紹介するのは、64 ビットプロセッサで提供しうる 64 ビットの広大な仮想アドレス空間を積極的に利用した SASOS(Single Address Space OS) と呼ばれる OS である。

6.1 Cubix

Cubix[256, 257] は 64 ビットプロセッサでの利用を想定し、単一の広大なアドレス空間 (2^{64} = 16 エクサバイト) を利用し、かつ、プログラム (UNIX でいうところのプロセス) とファイルを区別することなく、その空間中にすべて配置するメモリモデルが特徴である。このモデルによって、従来システムコールを介して行われる IPC (プログラム間通信) やファイルアクセス (read, write など) を、Cubix ではメモリアクセスの形で直接行なえるので、OS のオーバヘッドが低下し、高速なプログラム実行を可能とする場が提供できる。

また、従来プログラムごとに独立な仮想記憶空間を割当て、空間の独立性でプログラムを保護してきたが、Cubix では空間とは独立した保護概念を導入し、同一アドレス空間内でもメモリ保護を可能とした。これは、ページ単位の保護機構で、メモリ管理機能 (MMU) に多少の拡張をすることによって、より高速に機能するように考えられている。

Cubix は以下の特徴をもつ。

1. **単一仮想記憶空間 (SVS) の採用**

   UNIX などの OS では、プログラムごとに仮想記憶空間が割り当てられ、互いに空間が独立している多重仮想記憶空間 (MVS) 方式であるが、Cubix ではすべてのプログラムやデータが共通の単一仮想記憶空間中に存在する（図 1)。

   この SVS の導入により、プログラム間の通信は、メモリポインタの受け渡しとサブルーチン呼び出しで行える。

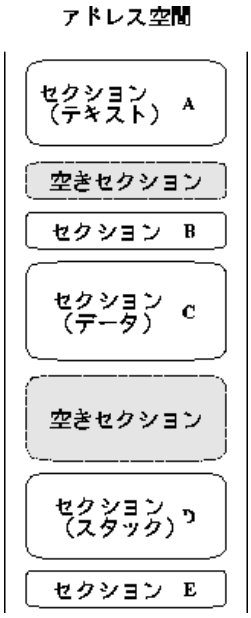

図 6.1: 仮想空間構成概念図

2. **高機能メモリ保護機構を用いたアクセス保護**

   仮想空間は複数のメモリセクションと呼ぶ領域から構成される。各メモリセクション毎に「スレッド ID」と「現在スレッドが実行しているプログラムの存在するメモリセクション ID」の２つをキーとして readable/writable/executable の各属性を指定することにより、メモリセクションに対するアクセス保護を設定する。ここでスレッド ID がキーとして加わっているため、共通の単一仮想記憶空間でありながら、スレッドごとに参照可能なメモリセクションが異なり、スレッド間のアクセス保護が実現できている。

3. **一元化記憶域 (OLS) の採用**

   ディスク上のデータとメモリ中のデータを区別なくアクセスできるように、すべてを仮想記憶空間中に存在させる。

   ディスクに格納されているファイルは、Cubix ではメモリセクションに対するバッキングストアとなる。物理メモリは、実体に対するキャッシュとして扱う。この点で Cubix のメモリセクションは Mach[258] におけるメモリオブジェクトと類似の概念である。

4. **マイクロカーネルアーキテクチャ**

   Cubix は、マイクロカーネルアーキテクチャ構成を採用している。したがって、Cubix マイクロカーネルの上に UNIX サーバなどの各種システムサーバをのせ、柔軟な OS の構成ができるようにした。この形で既存の各種 OS との互換性を提供する。Cubix の全体構成を図 6.1に示す。

6.2 Opal

ワシントン大学では、Opal という Cubix と同様な単一仮想記憶空間方式の OS を開発している [259]。

保護方式は、単一仮想空間上にプロテクション・ドメインと呼ぶ保護単位の領域を多重に重ねることにより、スレッド、プログラムの保護を行うものである。

Cubix のモデルでは、保護をより柔軟にするためにスレッド ID とメモリセクション ID の 2 つの保護キーにより保護を実現しているが、Opal では、従来のコンテクスト ID と同様な考え方で、一つのキー（プロテクション・ドメイン ID）で保護している。したがって、OS の実装にあたっては、特殊なハードウェアを仮定していないがそのままでは UNIX の fork の概念がサポートできないという欠点がある。

6.3 おわりに

以上、Cubix を中心にして、簡単に SASOS について報告した。Cubix は、現在。機能検証モデルとして、32 ビットの DOS マシン上にプロトタイプが動作している [260]。今後は、高速ネットワークに適合した OS として発展させる予定でいる。

携帯型の個人情報機器、マルチメディア、高速ネットなどの OS をとりまく環境は、時代とともに変化している。今後とも、新しい計算機環境に適した OS がいろいろ登場するだろう。UNIX をはじめ、このような OS の動きにも注目し、積極的に WIDE 環境に取り入れていくことが必要であろう。

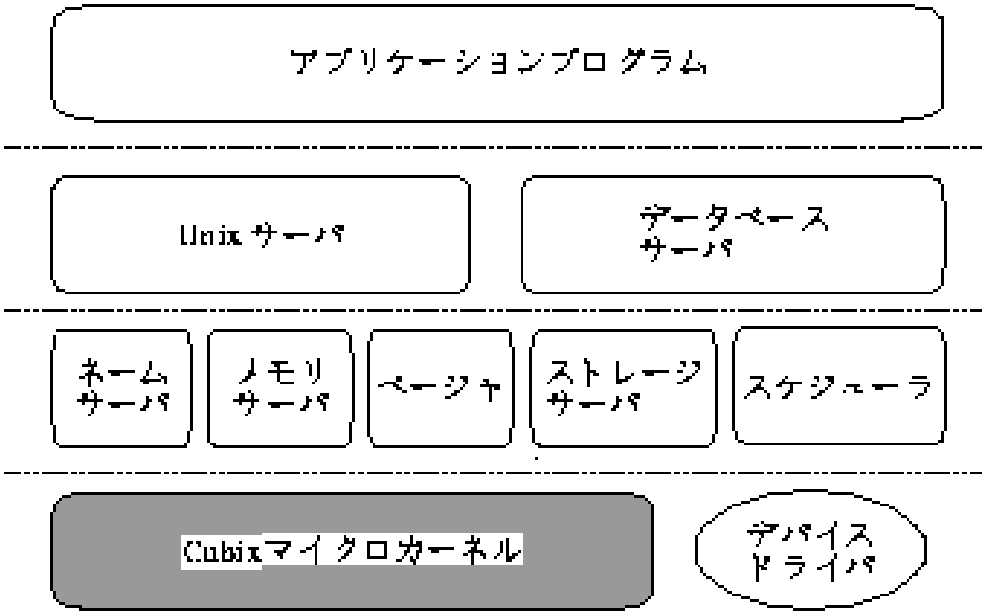

図 6.2: Cubix 構成概念図