プロジェクト B−Free 〜 *Introduction*

内藤隆一

1994,1995

目 次

第1章　B−Free プロジェクトとは何か

— そのときはわからなかったが、これはそれまで私が手掛けた中でもっとも大胆なオペレーティングシステムのプロジェクトになるのだった。
Helen Custer 「INSIDE WINDOWS NT」

1.1　目的

B−Free は、フリーな BTRON を作成することを目的としたプロジェクトです。

すなわち、B−Free プロジェクトは、他のＯＳを使わずスタンドアローンで動作する BTRON。BTRON−OS の上で動作するマネージャ群。そして基本的なアプリケーションなどから構成される完全なＢＴＲＯＮ環境を提供します。

現在のところ、BTRON 仕様の OS は商業的な目的に作成されたものだけでした。BTRON−OS に興味を持っている人は、BTRON 仕様 OS を購入して使うしか方法はありませんでした。もちろん、この場合興味があってもソースなどは見ることができません。しかし、B−Free プロジェクトは商業的な目的で作成するものではありません。自由に使用することができ、興味があるならばソースを見ることはもちろん、プログラムを変更することもできます。

このようなプログラムを自由にながめ、変更できるような環境を作成することを目的としたプロジェクトとして GNU プロジェクトがあります。基本的な目的は、B−Free プロジェクトと GNU プロジェクトは似ています。しかし、GNU プロジェクトが (ある程度改良は施されているとはいえ) UN*X のような (like UN*X) ＯＳの作成を目標としているのに対し、B−Free では、BTRON に基づいた環境を作成することで異なっています。

1.2　対象とする仕様

1994 年現在、BTRON は、次の 3 つの仕様が決められ (あるいは決められようとし) ています

- BTRON1 16 ビット (正確には 80286 プロセッサ) を対象とした BTRON 。構造は一枚岩的なもの (?) だが、比較的貧弱なハードウェアで実用的なマルチタスク / マルチウィンドウ OS を実現している。
- BTRON2 TRON チップ仕様プロセッサあるいは同等の性能をもつ 32 ビットプロセッサを対象とした OS 。 マイクロカーネル構造を採用しており、保守性および拡張性に優れている。
- BTRON3 BTRON1 と BTRON2 の経験に基づき定義された BTRON 仕様の最新バージョン。BTRON1 と同様にマイクロカーネル構造をもつ。1994 年前半に仕様が fix される予定。

これらの BTRON 仕様のうち、B−Free プロジェクトで作成する BTRON は、BTRON [13] を対象とします。

ただし、BTRON [13] 仕様のうちマシン依存になっている部分については変更される場合もあります。たとえば、80286 を対象とした仕様などについては変更されるでしょう。また、ＯＳの内部構造については BTRON 1 仕様では定められていないため、独自に決めることとします。

1.3　残りの内容

本ドキュメントの残りの章は、次のような構成になっています。

Chapter 2　一般ユーザから見た B−Free OS の説明です。

Chapter 3　B−Free OS の構造を概略を説明します。

Chapter 4　B−Free OS の最もハードウェア寄りの部分 — 中心核について説明します。

Chapter 5　ユーザプログラムから見た API を提供する層である LOWLIB について説明します。

Chapter 6　B−Free OS の周辺核について説明します。

Chapter 7　B−Free OS と周辺装置との間を取りもつソフトウェアであるデバイスドライバについて説明する章です。

Chapter 8　B−Free OS の中で一番ユーザプログラムに近い層である、外核について説明します。

Chapter 10 B−Free のシステムインタフェース (API) のひとつ、POSIX 環境について説明します。

第2章　ユーザから見た B−Free OS

— とんでもない、ワトスン、きみには何もかも見えているんだよ。
コナン・ドイル 「青い紅玉」

B−Free OS は、ユーザからはシングルユーザ・マルチタスクの OS として見えます。

B−Free が起動すると、画面がクリアされ、*root* **ウィンドウが表示されます**。

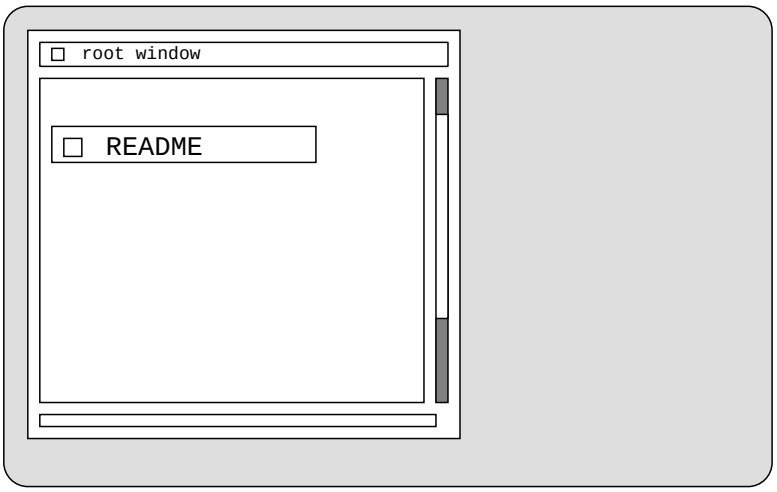

図 2.1: BTRON root ウィンドウ

この状態で、ユーザは root ウィンドウ上に表示されている仮身を操作することができます。

2.1　B−Free のファイルシステム

ここでは、B−Free のファイルシステムについて説明します。

最初にシステムをインストールした時点では、B−Free システムのファイルシステムは、図 2.2 のようになっています。

図 2.2: インストールした直後の B−Free のファイルシステム

この図のなかで、ユーザが直接見ることができるファイルは、網箱で囲んだ部分だけです。

root ファイルシステムは、中心核や各種マネージャなどのシステムで使用する実身を含んでいます。

- SYSTEM この実身には *KERNEL* という実身がひとつあるだけです。*KERNEL* という実身は、中心核の実行実身がそのまま入っています。
- MANAGER この実身には、システムで使用する各種マネージャ(周辺核および外核が含まれています。
- $$PROGRAM.BOX この実身には、アプリケーション群へのリンクが含まれています。
- $$RELATION.BOX この実身には、続柄の情報が入ります。
- $$小物入れ この実身には、小物 (アクセサリー的なアプリケーション群へのリンクが入ります。
- USR ユーザ自身が使用する実身が含まれています。立ち上げ直後、ディスプレイに表示されるウィンドウは、この *USR* 実身の内容です。

2.2 ユーザの情報

B−Free OS は、シングルユーザ向けの OS です。しかし、多くの BTRON マシンがつながったネットワークを構築したような場合、ネットワーク全体では複数のユーザが資源 (実身・プリンタなど) を共有することができます。そのため、ユーザごとに特有の情報を記録する必要があります。

B−Free OS では、ユーザ情報として以下の情報を管理します。

ユーザ名

ユーザの名前です。

所属グループ名

B−Free OS では、何人かのユーザが集まってグループを作ることができます。所属することができるグループは、4 つまでです。ユーザ情報では、ユーザが所属するグループの名前を記録します。

特権レベル

B−Free OS は、ユーザを 0 から 16 の特権レベルに分けています。B−Free での実身には、読み書きできる特権レベルを記録しています。

特権レベルでは、レベル 0 が最も高い特権をもち、すべての実身を読み書きできます。レベル 0 のユーザに対して、実身を読み書きできないような指定はできません。

第3章　B−Free OS の構造

— われわれの間では、この物体を一応 “SS” のコードネームでよんでいる。” スーパーシップ” または” スーパー・ストラクチャ” の意味だ。
小松左京 「虚無回廊」

3.1　一枚岩的な OS vs マイクロカーネル

OS は、一枚岩的 な構造をもつものが多数作られてきました。一枚岩的な OS は、その名のとおり OS がひとつの巨大なプログラムとなっています。

すなわちコンピュータは、2 つの動作モード — カーネルモードとユーザモードをもち、カーネルモードで走るプログラムこそが OS という考えがその根底にあります。

ユーザモードで走るプログラムは、あくまでもユーザが作成したプログラムであり、OS 的な機能はありません。

一枚岩的な OS の代表は、UN*X です (図 3.1)。

UNIX の世界では、一枚岩的な OS — カーネルと呼びます、とユーザプログラムという 2 種類のプログラムしかありません [1]

一枚岩的な構造をもつカーネルの場合、カーネルを変更するには大変な努力が必要となります。それは、ある小さな変更をするだけでも他の (関係のなさそうな) 部分に影響が及ぶ可能性があるからです。

一枚岩的な構造をもつオペレーティングシステムは今でも主流です。しかし、一枚岩的な構造には、保守性や拡張性に問題があることが徐々に分かってきました。そのため新しく生まれた考えかたがマイクロカーネルという考えです。

マイクロカーネルという考えでは、ハードウェアに密着した部分などを小さなモジュール (マイクロカーネル) にまとめます。そして、一枚岩的なオペレーティングではカーネルがやっていたほとんどの仕事をマイクロカーネルの外へ追い出します。

マイクロカーネルの代表は Mach オペレーテイングシステムです。正確にいうと、Mach オペレーティングシステムは、マイクロカーネルアーキテクチャをとるシステムで使用するためのマイクロカーネルにすぎません。OS と

[1] デーモンと呼ばれる特殊なプログラムもありますが、これもまたユーザプログラムの一つです。

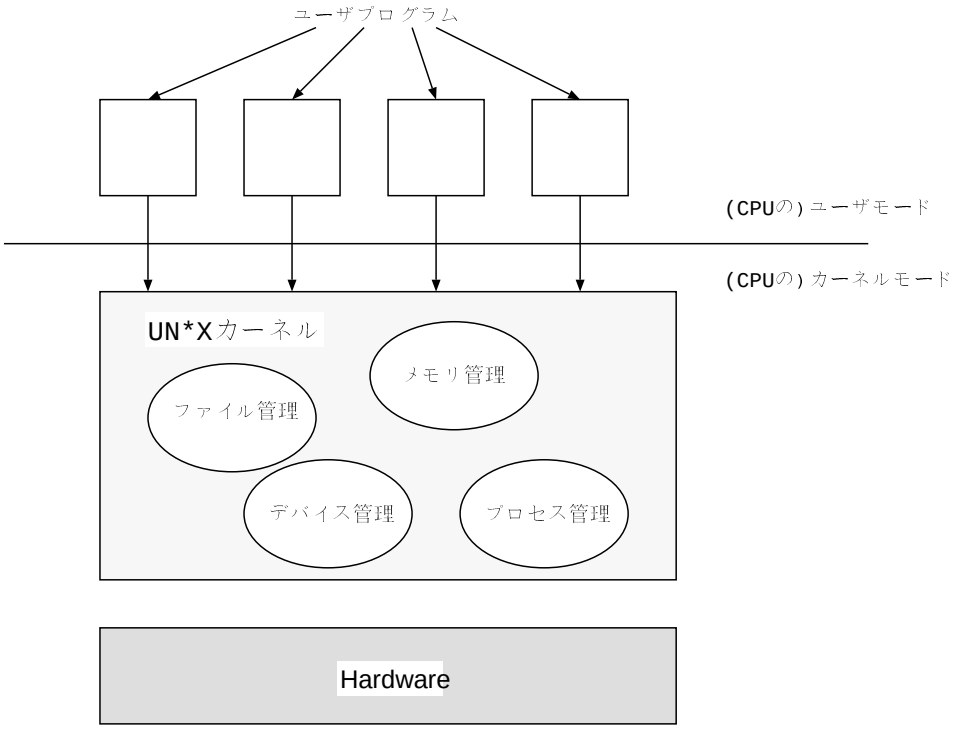
ユーザプログラム
(CPUの)ユーザモード
(CPUの)カーネルモード
UN*Xカーネル
メモリ管理
ファイル管理
デバイス管理
プロセス管理
Hardware

図 3.1: UN*X の構造

してユーザプログラムを動かすためには、Mach の上に載る複数のプログラムが必要となります。

現在のところ、Mach の上にのるプログラム (Mach ではサーバと呼んでいます) は、UN*X のインタフェースをもつ UN*X サーバや MS-D*S のインタフェースをもつサーバなどがあります。また、最近では GNU プロジェクトでも Mach を基にした Hurd という OS の作成を行っています。

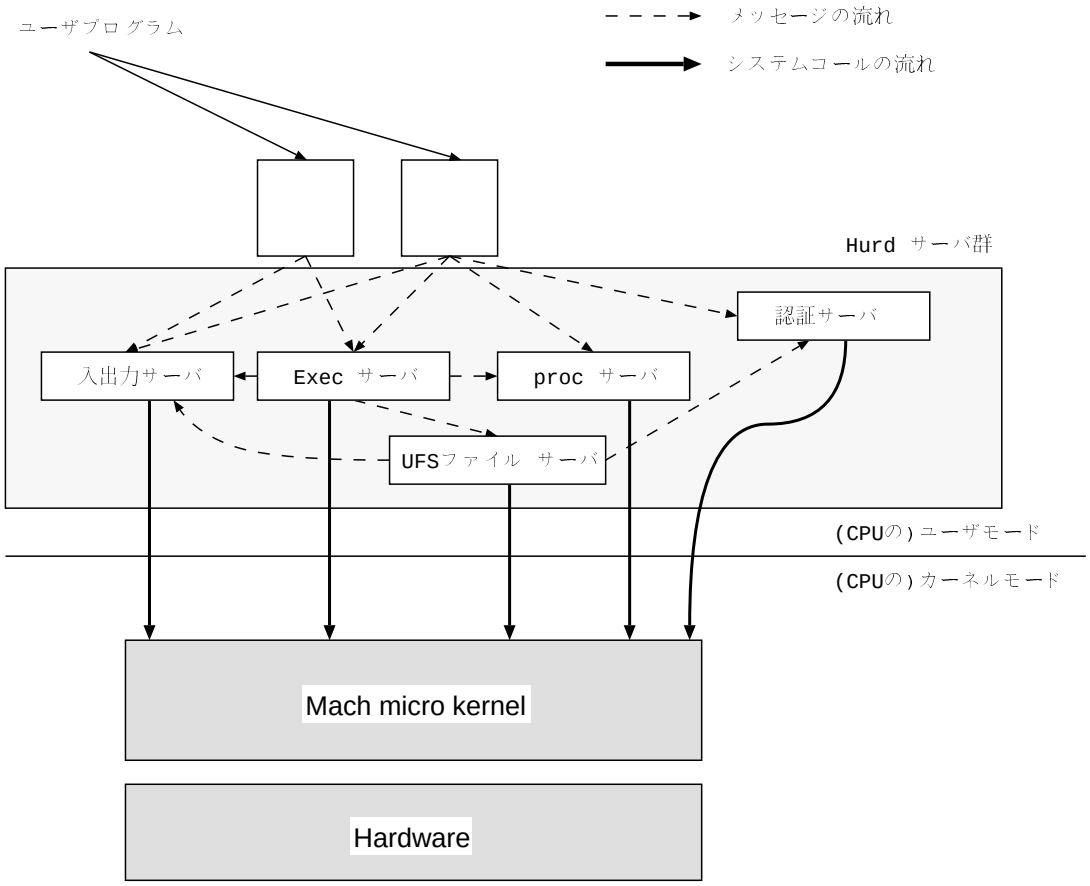

図 3.2: Mach + Hurd の構成

一枚岩的な OS とマイクロカーネルを比較してどちらが良いということは一言では言えません。

さて、問題は B−Free OS でどちらのアーキテクチャを採用するかということです。

B−Free OS を製作する目的 (OS のソースを公開し、簡単に変更などをできるようにする) を考えると、見とおしがよい構造というのが重要になってきます。そう考えると、一枚岩的な OS の利点はほとんど性能面においてであり、

ソースの変更や改良などを簡単に行うのは困難です。逆にマイクロカーネル方式の OS では、よほどうまく作らないと、性能面では一枚岩的な OS よりも劣ります。しかし、中の構造はひとつひとつの要素が分かれており、それぞれの変更が他に与える影響が少ない分だけ見とおしがよいといえそうです。

これらのことから、B−Free OS の目的 (ユーザが自由にソースを見て、OS を変更できる) を考えると、マイクロカーネル方式を採用するのが適当だと思われます。

3.2 全体構成

B−Free での OS の構造は、マイクロカーネル構造をとります。

OS の中心となる核として、ITRON を採用します。この ITRON は、μITRON 3.0 を基にしたものです。

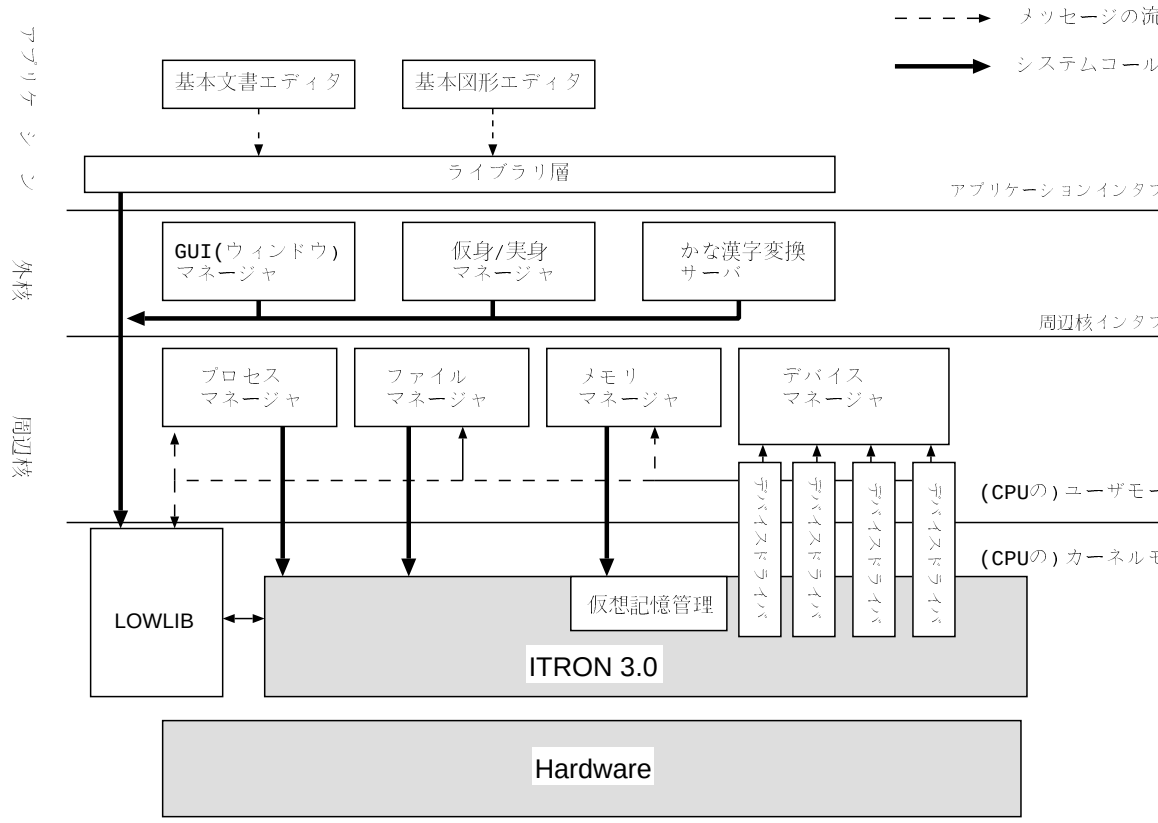

図 3.3: B−Free の構成

3.2.1 B−Free OS の構成要素 (obsoleted)

B−Free OS は、次の構成要素から成り立っています。

- 中心核 (いわゆるマイクロカーネル) μITRON 3.0 に準拠した ITRON OS です。CPU のカーネルモードで動きます。

- LOWLIB アプリケーションプログラムに対してシステムコールインタフェースを提供するための層です。

  厳密にはライブラリではないため、LOWLIB (低レベルライブラリ) という名前になっています。

  LOWLIB は、システムコールインタフェースを提供する他にユーザプロセスの初期化や各周辺核との通信なども行います。

- 周辺核 BTRON OS としての API を提供します。中心核の提供するシステムコールを使用するタスク群です。ファイル管理やプロセス管理などの機能を提供します。また周辺核にあるデバイスマネージャは、デバイスドライバへのアクセスする手段を提供します。

  周辺核に含まれる機能は次のとおりです (カッコの中はその機能を実現するマネージャの名前です)。

  仮想メモリ管理 (メモリマネージャ)
  : 仮想記憶を管理するためのマネージャです。このマネージャでは、仮想メモリの高度な機能を提供します (物理ページの参照管理など)。

    また、マネージャとは別に MMU を操作するような機能は中心核に含まれています。中心核の提供する仮想記憶管理は、ある程度統一化されています。

  プロセス管理 (プロセスマネージャ)
  : 中心核の提供するのは ITRON レベルでのタスク管理機能ですが、プロセスマネージャは、中心核のもつタスク管理のインタフェースを使って BTRON レベルのプロセス管理機能を上位の層に提供します。プロセスの持つ情報を管理するのが主な処理です。

  ファイル管理 (ファイルマネージャ)
  : BTRON レベルでのファイル管理を行います。このレベルでは仮身/実身という単位での管理ではなく、ファイル/レコードという単位で管理が行われます。

  ウィンドウマネージャ
  : ウィンドウの管理を行います。描画自体は、ディスプレイデバイスドライバが行います。

デバイス管理 (デバイスマネジャ)
: デバイスドライバの登録/参照などの管理を行います。

- デバイスドライバ周辺核よりも上位のソフトウェアとハードウェアとの間をとりもつソフトウェアです。基本的に周辺機器ひとつごとにひとつのデバイスドライバが存在します。

  次のようなデバイスドライバは、最低必要となります。

  - ディスプレイ・デバイスドライバ
  - キーボード・デバイスドライバ
  - ポインティグ・デバイスドライバ
  - FD/HD デバイスドライバ

- 外核/殻補助的なサービス — かな漢字変換や仮身・実身操作など、を提供するプロセス群です。

- ライブラリアプリケーションに対して、BTRON API を提供します。実際の処理は、外核や周辺核と協調し、データをやりとりすることによって行います。

- アプリケーションユーザが使用するプログラムです。実身の内容を表示するデータランドエディタ、テキスト実身の内容を編集する基本文書エディタ、そして図形実身の内容を編集する基本図形エディタなどがあります。

第4章　中心核

— 銀河中心核には、とにかく早く入って早くでなければならない。物理法則が存在するかぎり、あと戻りする道はないからな。
ドナルド・モフィット 「第二創世記」

B−Free OS で最もマシン寄りの部分、それが中心核です。中心核は、それ自体 ITRON 仕様の OS となっています。中心核と上位の層とのインタフェースは、ITRON システムコールとして決められたインタフェースを使用しています (一部拡張してあります)。そのため、異なった CPU 上に B−Free OS を移植する場合でも、変更は中心核のみに留め、上位層の変更は最小限にすることができるようになっています。

この章では、中心核の機能と構造について説明します。

4.1　中心核の機能

中心核は、μITRON 3.0 の基本仕様 (一部拡張仕様も含む) 準拠のカーネルです。

中心核は、周辺核、外核そしてアプリケーションなどの上位層に対して次の機能を提供します。

- タスク管理
- 同期・通信機能 (IPC)
- メモリプール管理機能
- 割り込み管理機能
- 例外管理機能
- 時間管理機能
- システム管理機能

この他に μITRON 3.0 では規定されていない次の機能も提供します。

- 仮想メモリ管理機能

中心核は、基本的な OS の機能を上位層に与えます。なお、上位層は中心核に対してシステムコールを発行することによって、中心核の機能を使用します。しかし、中心核より上位の層 (周辺核、外核も含む) は、すべてメッセージの送受信により要求の送受信を行います。システムコール (= CPU でのトラップ) を介して呼び出されるというのは中心核だけです。

中心核では、次のシステムコールをサポートします (* 印は現在未サポート)。

4.1.1 システムコール呼び出しの経路

ユーザアプリケーションが、BTRON の機能を使用する場合、次のような順序で処理を行います (図 4.1)。

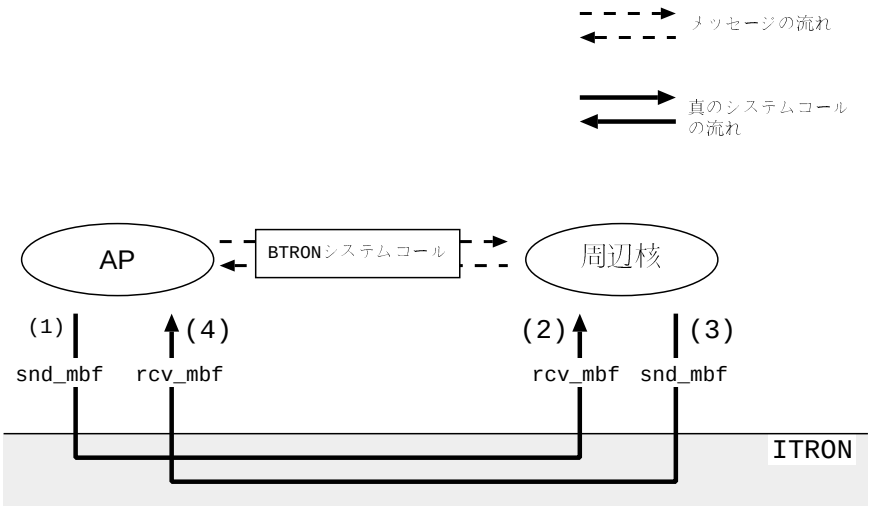

図 4.1: システムコール呼び出しの経路

1. ユーザプログラムは、中心核に対してメッセージを送信するシステムコールを発行して、周辺核へメッセージを送る。
2. 周辺核は、送られてきたメッセージを受けとり、メッセージに書かれた要求を実行する。

表 4.1: 中心核でサポートするシステムコール

タスク管理関係	cre_tsk del_tsk sta_tsk ext_tsk exd_tsk ter_tsk dis_dsp ena_dsp chg_pri rot_rdq rel_wai get_tid ref_tsk	
タスク附属同期機能	sus_tsk rsm_tsk frsm_tsk slp_tsk tslp_tsk wup_tsk can_wup	
同期・通信機能	cre_sem del_sem sig_sem wai_sem preq_sem twai_sem ref_sem cre_flg del_flg set_flg clr_flg wai_flg pol_flg twai_flg ref_flg cre_mbf del_mbf snd_mbf psnd_mbf tsnd_mbf rcv_mbf prcv_mbf trcv_mbf	
割込み管理機能	dis_int ena_int	
メモリプール管理機能	cre_mpl	

3. 周辺核は要求を処理すると、結果をメッセージの形にして中心核を介してアプリケーションへ送る。

4. ユーザアプリケーションは、返答メッセージを受けとる。(システムコールの終了)

なお、中心核を呼び出す処理というのは、ライブラリが行うので、アプリケーションが中心核を意識することはありません。ファイルの読み書きなどの処理は、周辺核のファイル管理マネージャが、メッセージを受けとることによって処理します。

4.1.2 接続機能について

また、中心核では、μITRON 3.0 で新たに拡張された接続機能については、サポートしません。異なったホスト間での通信機能については、中心核よりも更に上位の層でサポートします。μ ITRON3.0 の接続機能を使用しない理由は次のとおりです。

- μ ITRON 3.0 での通信機能は組み込み機械で CPU が複数ある場合を想定している。
- 基本的に CPU がひとつだけ入っており、他のマシンとは比較的大域の広いインタフェース (Ethernet など) が使えるパソコンとは相性が悪い。

つまり、μ ITRON 3.0 での接続機能は、パーソナルコンピュータでの通信のような用途には向かないのではないかということです。

4.2 中心核の構成

中心核の構成を図 4.2 に示します。
中心核は、いくつかのモジュールに分かれています。

- タスク管理部分タスク管理では、ITRON の意味でのタスクを管理します。タスクは実行単位としてのプログラムを意味しています。タスク管理部分では、タスクの生成/削除/実行などの操作の他に、タスク同士で同期や通信などを行う機能も含んでいます。
- メモリ管理部分メモリ管理部分では、物理メモリの管理を行います。物理メモリは基本的にページ単位 (80386 で 4K バイト) で管理を行います。しかし、メモリ管理部分が提供するインタフェースでは、バイト単位での物理メモリの取得/解放ができるようになっています。

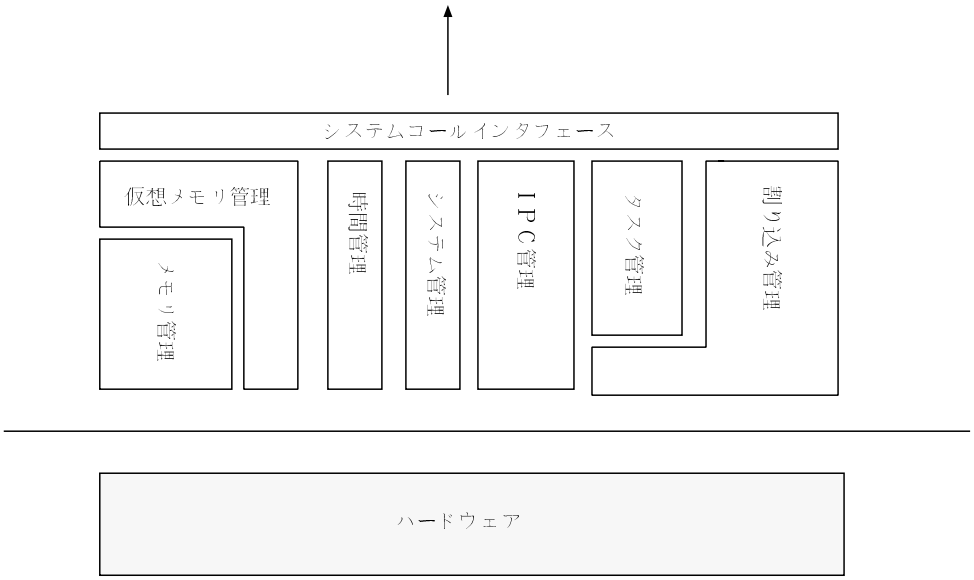

図 4.2: 中心核の構成

- 割込み/例外管理部分割込み管理では、外部割込みおよびトラップ (例外、内部割込み) の割り込みテーブルへの登録を行います。また、実際に割り込みが起った場合の各関数への処理の割り振りも行います。
- 時間管理部分一定時間ごとの指定された関数の実行をする機能を提供します。
- システム管理部分バージョン番号などを管理します。
- 仮想メモリ管理部分 CPU のもつ仮想メモリの管理機構をそのまま提供します。更に抽象的な仮想メモリの管理については、周辺核のメモリマネージャが行います。

  この仮想メモリの管理は、μ ITRON 3.0 では規定していないため、B−Free 独自の仕様を定めます。

4.3 タスク管理部

タスク管理部分は、次のモジュールからできています。

4.3.1 タスク情報

各タスクには、構造体 `t_tcb` の形式のデータがひとつ割り当てられます。

表 4.2: タスク管理部を構成するモジュール

ファイル名	内容
common/task.c	タスク管理システムコール関数の定義
i386/tss.c	TSS (Task State Segment) の管理をする。
i386/startup.s	IDTR/GDTR の設定などを行う。
i386/locore.s	タスクスイッチ等アセンブラで書いた関数。
h/task.h	タスク管理用の定義ファイル。

t_tcb は次の情報を記録します。

- タスクリストのためのリンクリスト
- タスク ID
- タスクの状態
- タスクのプライオリティ(優先順位)
- タスク属性
- タスクが待ち状態のときに使用する情報
- タスク間通信で使用する情報
- スタック情報
- コンテキスト情報 (80386)
- 仮想記憶のために使用する情報

現在動いているタスクは、run_tsk というポインタ変数が指し示しています。

タスク ID

タスク ID は MIN_TASKID から MAX_TASKID までの範囲を占める整数値 (32 ビット) です。MIN_TASKID と MAX_TASKID は、src/kernel/itron-3.0/h/config.h で定義しています。

デフォルトでは、次の値となります。

MIN_TASKID	1
MAX_TASKID	128

中心核のシステムコールでは、対象となるタスクを指定するためにタスク ID を使います。

タスクプライオリティ

タスクのプライオリティ(優先順位) は、32 ビットの整数で表現します。
この値の範囲は、マクロ MIN_PRIORITY(プライオリティの最小値) から
MAX_PRIORITY(プライオリティ値の最大値) となります。

プライオリティは、値の小さい方が優先度が高くなります。そのため、MIN_PRIORITTY
が一番高くなります。

MIN_PRIORITY / MAX_PRIORITY は、src/kernel/itron-3.0/h/config.h
で定義しています。デフォルトの値は次のとおりです。

MIN_PRIORITY	0
MIN_PRIORITY	31

タスクのレディキューの配列は、各エントリがひとつのプライオリティに
対応しています。そのため、プリオリティ値の範囲がレディキューの配列の
エントリ数となります。

タスク属性

記述言語の指定を行います。μ ITRON 3.0 では、TA_ASM (アセンブラ)
と TA_HLNG (高級言語) の 2 つの状態をもちますが、B−Free の中心核では
記述言語による区別はしていないので、TA_HLNG だけが使用できます。

タスク状態

タスクは、表 4.3 示した値のどれかの状態になります。
タスク状態は、図 4.3 に示す状態遷移図のように変化します。

タスクが待ち状態のときに使用する情報

タスクは、以下の原因によって待ち状態となります。

- システムコールによる待ち状態。
- タスク間同期・通信機能による待ち状態
- 資源取得時の待ち状態

タスクの待ち状態は、表 4.4 のマクロによって表現します。

表 4.3: タスクの状態

タスクの状態	タスクの状態
TTS_NON	非存在状態
TTS_RUN	実行状態
TTS_RDY	実行可能状態
TTS_WAI	待ち状態
TTS_SUS	強制待ち状態
TTS_WAS	強制待ち状態 + 待ち状態
TTS_DMT	未生成状態

タスク間通信で使用する情報

タスク間通信を行うための情報を表 4.5 に示すものがあります。

スタック情報

カーネルモードで動作するときのスタック領域を指すポインタ変数です。

スタックの領域は、関数 `make_task_stack()` が作成します。スタックのサイズは、1 ページ [1]です。

コンテキスト情報

コンテキスト情報は、CPU に依存したタスクの情報です。コンテキストスイッチが発生した時点でのレジスタの値が入ります。

コンテキスト情報は、T_I386_CONTEXT [2]という型で定義しています (80386 の場合)。

[1]80386 の場合、4K バイトです

[2]src/kernel/itron-3.0/i386/i386.h で定義

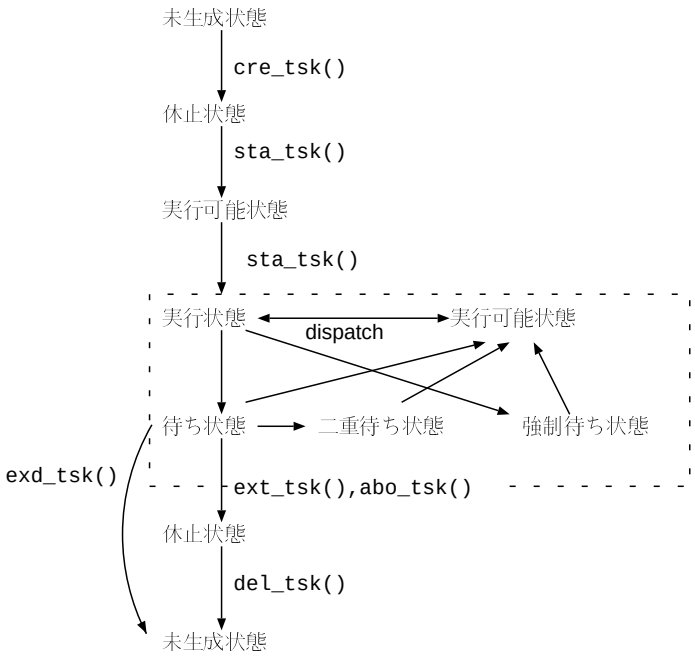

図 4.3: タスクの一生

表 4.4: 待ち状態一覧

状態を示す値 (マクロ)	内容
TTW_SLP	slp_tsk, tslp_tsk による待ち
TTW_DLY	dly_tsk による待ち
TTW_NOD	接続機能応答待ち
TTW_FLG	wai_flg, twai_flg による待ち
TTW_SEM	wai_sem, twai_sem による待ち
TTW_MBX	rcv_msg, trcv_msg による待ち
TTW_SMBF	snd_mbf, tsnd_mbf による待ち
TTW_MBF	rcv_mbf, trcv_mbf による待ち
TTW_CAL	ランデブ呼び出し待ち
TTW_ACP	ランデブ受け付け待ち
TTW_RDY	ランデブ終了待ち
TTW_MPL	get_blk, tget_blk による待ち
TTW_MPF	get_blf, tget_blf による待ち

表 4.5: タスク間通信で使用する情報

タスク間通信の種類	記憶する情報
セマフォ	セマフォ待ちリストのためのポインタ情報
イベントフラグ	待ち条件の値。イベントフラグの ID
メッセージ	メッセージ待ちリストのためのポインタ情報

コンテキスト情報構造体 (T_I386_CONTEXT)

```
typedef struct
{
  UW            backlink;
  UW            esp0;
  UW            ss0;
  UW            esp1;
  UW            ss1;
  UW            esp2;
  UW            ss2;

  UW            cr3;
  UW            eip;
  UW            eflags;
  UW            eax;
  UW            ecx;
  UW            edx;
  UW            ebx;
  UW            esp;
  UW            ebp;
  UW            esi;
  UW            edi;
  UW            es;
  UW            cs;
  UW            ss;
  UW            ds;
  UW            fs;
  UW            gs;
  UW            ldtr;
  UH            t:1;
  UH            zero:15;
  UH            iobitmap;
} T_I386_CONTEXT;
```

4.3.2 コンテキストスイッチ

コンテキストスイッチは、task_switch() と resume() が行います。

task_switch()[3]は、レディタスクリストの中で、一番優先順位の高いタスクをカレントタスクにします。実際のタスク切り換えは、resume() によっておこないます。そのため、この関数の中での処理は、run_tsk 変数とレディタスクリストの更新だけとうことになります。

task_switch() は、引数 save_nowtask をもちます。この save_nowtask が TRUE のとき現タスクをレディタスクリストに保存します。FALSE の時は、ready タスクキューから削除します。

レディタスクリストから削除しない場合、他のタスクのプライオリティが下がると、コンテキストスイッチを行ったタスクは、再び実行されることになります。

逆にレディタスクリストから削除した場合、他のタスクがレディリストに追加しない限り元のタスクが実行されることはありません。レディタスクリストから削除するのは、タスクが待ち状態に入ったときに行います。

resume() [4]は、CPU のコンテキストスイッチ機能を使って、コンテキストスイッチを行います。

具体的には、引数で渡されたセレクタが指しているプロセスの TSS にジャンプします。

resume() のソースは、高々 20 ステップしかないので、実際のリストを次に示します。タスクスイッチは、.byte 0xff, .byte 0x28 と書いた部分で行っています。これは、アセンブラに「TSS にジャンプする」という動作に対応するニモニックが定義されていないためです。

[3]common/task.c の中で宣言しています。
[4]i386/locore.s 内で宣言しています。

```
_resume:
                cli
                pushl   %ebp
                movl    %esp, %ebp
                movl    8(%ebp), %edx
                movw    %dx, (selector)
                lea     offset, %eax
                movl    %cr3, %ebx      /* TLB キャッシュをフラッシュする */
                movl    %ebx, %cr3
_resume0:
                .byte   0xff            /* ここで TSS にジャンプしている */
                .byte   0x28
                leave
                sti
                ret
```

4.3.3 レディタスクリスト

タスク管理部分では、実行可能なタスクのリストをもっています。このリストは、common/task.c の中で宣言しています。

ready_task	実行可能なタスクのリスト。プライオリティ順の配列となっている。

このレディキューを操作するために以下の関数を定義しています。これらの関数は、引数として

- 操作するリスト
- 追加/挿入/削除するエントリ情報

の 2 つをもらい、修正したリストを返り値として返します。

init_task

TCB テーブルの内容を初期化します。そして、カレントタスクをタスク番号 (-1) のタスクとします。

add_tcb_list

引数 list で指定されたリストの一番最後にタスクを追加します。

ins_tcb_list

引数 list で指定されたリストの一番最初にタスクを挿入します

del_tcb_list

引数 list で指定されたリストから、要素 del を削除します。

4.4 メモリ管理部

メモリ管理部分では、主に 2 つの形態の物理メモリを管理します。

- ページ (4K バイト) 単位での管理
- バイト単位での管理

さらに、μ ITRON 3.0 で定義している可変長メモリプールに関するシステムコールのためのモジュールがあります。

4.4.1 ページ単位のメモリ管理

ページ単位でのメモリ管理は、ファイル common/pmemory.c にある関数で管理します。

物理メモリページは、memory_map[MEMORY_MAP_SIZE] という 1 エントリがバイトの大きさをもつ配列で管理します。この配列のエントリが 1 ページの物理メモリに対応します。

各エントリは、次に示す値のどれかになります。

MEM_FREE	メモリページは、使用していない状態 (フリー)
MEM_USE	メモリページは、使用している状態

pmem_init()

配列 memory_map の内容を初期化します。

palloc()

サイズで指定したページ数分の連続した物理メモリ (ページ) をアロケートします。

配列 memory_map のエントリのうちアロケートする物理ページに対応するものの値を MEM_USE に変更します。

pfree()

palloc() でアロケートした物理メモリ (ページ) をフリー状態にします。

配列memory_map の解放する物理ページに対応するエントリの値を MEM_FREE に変更します。

4.4.2 バイト単位でのメモリ管理

ITRON では、バイト単位のメモリのアロケート/フリーを行うためのシステムコールを定義しています。

バイト単位のメモリ管理を行うための関数は、common/itron_memory.c で定義しています。

表 4.6: バイト単位のメモリ管理を行う関数

関数名	動作
init_kalloc()	バイト単位のメモリ管理の初期化
kalloc()	バイト単位のメモリ領域の取得 (アロケート)
kfree()	kalloc で取得したメモリ領域の解放 (フリー)
getcore()	新たに kalloc が管理するメモリ領域を追加する。

バイト単位のメモリ管理を行うために、フリーなメモリ領域をリストで管理しています。各フリーメモリ領域は、構造体 kmem_entry で管理します。この構造体は、freelist という変数につながったリストとなります。

メモリをアロケートする時は、この freelist を辿り、取得したいサイズをもつフリーメモリ領域を割り当てます。フリーリストにアロケートできるようなサイズをもつフリー領域がない場合には、getcore() によって新しいフリーメモリを取り出します。このとき、getcore() は、ページ単位のメモリのアロケートを行う palloc() を呼び出します。

4.4.3 可変長メモリプールシステムコール

μ ITRON 3.0 ではメモリ管理機能として固定長メモリプールと可変長メモリプールの 2 種類を定義しています。中心核でサポートしているのは、このうちの可変長メモリプールに関するシステムコールです。

可変長メモリプールの管理は、common/itron_memory.c にある関数によって行います。

各可変長メモリプールは、構造体 memory_pool によって管理します。構造体 memory_pool には、次の情報を記録します。

表 4.7: 可変長メモリプールを管理する関数

関数名	動作
init_mpl()	メモリ管理機能を初期化します。
cre_mpl()	可変長メモリプールを生成します。
del_mpl()	可変長メモリプールを削除します。
get_blk()	
pget_blk()	可変長メモリブロックを獲得 (アロケート) します。
tget_blk()	
rel_blk()	可変長メモリブロックを返却 (フリー) します。
ref_mpl()	可変長メモリブロックの状態を参照します。

- メモリプールの ID
- サイズ
- メモリブロック取得待ちのタスクのリスト
- このメモリプールが管理しているフリー領域のリスト

メモリプールは、memory_pool_table[MAX_MEMORY_POOL] という配列で管理します。

各メモリプールに属しているフリー領域は、構造体 free_mem_entry をエントリとなるリストで管理しています。

メモリプールを生成するとき、メモリプールの管理するメモリのサイズを指定します。cre_mpl() では、先に説明したバイト単位のメモリ管理を行う関数 (kalloc()) を使ってメモリを取得します。

4.5 タスク間通信機能

中心核では、タスク間の同期・通信機能として次の機能を提供しています。

- セマフォ
- イベントフラグ
- メッセージバッファ

この章では、それぞれの機能の実装内容について説明します。

4.5.1 セマフォの実装

セマフォは、2 つ以上のタスク間で同期を取ったり、同時にアクセスすることができないデータを保護するための機能です。
そのために、次の機能が必要になります。

- すでにセマフォがロックされていた場合、セマフォが解放されるまでタスクを待たせる機能。
- セマフォが解放されたとき、セマフォ待ちの状態にあるタスクを実行させる機能。

セマフォ待ちのタスクを管理するために、待ち状態にあるタスクのリストを管理しています。
セマフォ機能は、次の関数が処理します[5]。

init_semaphore	セマフォ機能の初期化
cre_sem	セマフォの生成
del_sem	セマフォの削除
sig_sem	セマフォの資源返却
wai_sem	セマフォ資源獲得
preq_sem	セマフォ資源獲得 (ポーリング)
twai_sem	セマフォ資源獲得 (タイムアウト有)
ref_sem	セマフォ状態参照
twaisem_timer	twai_sem の時間切れのときに呼びされる関数

ひとつひとつのセマフォは、構造体 semaphore_t で管理しています。

```
typedef struct semaphore_t
{
  T_TCB       *waitlist;      /* セマフォ獲得待ちタスクのリスト */
  ATR         sematr;         /* セマフォ属性 */
  INT         isemcnt;        /* セマフォ獲得待ち数 */
  INT         maxsem;         /* セマフォ獲得待ち数の最大値 */
  VP          exinf;          /* 拡張属性（未使用） */
} T_SEMAPHORE;
```

配列 semaphore_table[NSEMAPHORE] は、すべてのセマフォ情報を収めています。

[5]すべて src/kernel/itron-3.0/common/semaphore.c の中で定義。

4.5.2　イベントフラグの実装

4.5.3　メッセージバッファの実装

4.6　割り込み/トラップ/例外管理部分

ハードウェアからの外部割り込みやシステムコールなどのトラップなどは、
この部分で管理します。
管理モジュールは、以下のとおりです。

i386/interrupt.s
common/fault.c

割り込み/トラップ/例外の違いは次のとおりです：

割り込み

トラップ

例外

割り込み/トラップ/例外処理というのは、i386 ではひとつのテーブル (IDT)
で管理しています。
表 4.8 に i386 で定義している割り込み/トラップ/例外の一覧を示します。

表 4.8: i386 で定義している割り込み/トラップ/例外の種類

番号	種類	内容
1		
2		
3		
4		
5		
6		
7		
8		
9		
10		

4.6.1 割り込みの処理

ハードウェア割り込み (外部割り込み) の処理は、i386/fault.c の中にある interrupt() で行います。

割り込みが発生すると、interrupt() は次の処理を行います。

- 大域変数 on_interrup の値を TRUE に変更。
- 大域変数 delayed_disatch の値を FALSE に変更。
- 各割り込み別に定義してある関数へ分岐。
- delayed_dispatch が TRUE に変更されていたら、task_switch() を実行。このとき、現走行タスクはレディキューから削除しない。

interrupt() が処理する割り込みは、表 4.9 のとおりです。

表 4.9: interrupt() の処理する割り込み一覧

番号	内容
32	タイマー
33	キーボード
42	フロッピーディスク (1M)

4.6.2 トラップの処理

トラップの処理は、trap() で行いますが、この関数は今のところ何もしていません。

トラップの一種であるシステムコールは、int64_handler [6] と syscall() [7]で処理します。

4.6.3 例外の処理

例外は、次の要因で発生する CPU のエラーです。

- 0 割り例外
- プロテクトフォールト

[6] src/kernel/itron-3.0/i386/interrupt.s で定義。
[7] src/kernel/itron-3.0/common/syscall.c で定義。

- ページフォールト
- 不法 TSS 例外

それぞれの例外が発生した時の対応は表 4.10 に示すとおりです。

表 4.10: 例外が発生した場合の対応

例外	対応	
	ユーザモード	カーネルモード
0 割り例外	強制終了 (ユーザ定義可能)	システム停止
プロテクトフォールト	強制終了	システム停止
ページフォールト	ページインまたは強制終了	システム停止
不法 TSS 例外	強制終了	システム停止

4.7 仮想記憶

4.7.1 i386 での仮想記憶管理機能

インテル i386 プロセッサには、ページ単位での仮想記憶を管理する機能があります。

4.7.2 モデル

B-Free での仮想記憶管理をモデル化したものを 図 4.4 に示します。

仮想記憶は、リージョン (Region) という単位で管理します。この場合の管理情報は、仮想領域のアドレス、物理メモリのマップ情報、そして、読み書きの許可を表す permission のことです。

1 つのタスクには 1 つ以上のリージョンを結びつけることができます。たとえば、BTRON レベルでのユーザプロセス (の中のタスク) は、プログラムの実行部分 (コード部分) が入るテキスト・リージョン、読み書きするためのデータが入るデータ・リージョン (実際には、データ・リージョンは、恐らく実行前に値が決まっている変数が入るリージョンと、実行前には領域だけが決まっているリージョンそして、ヒープのために使われるリージョンの 3 つのリージョンに分かれます)、そしてスタック領域を表すスタック・リージョンという複数の Region と結びついています。

タスクが複数のリージョンを所有するのは、次のような利点があります。

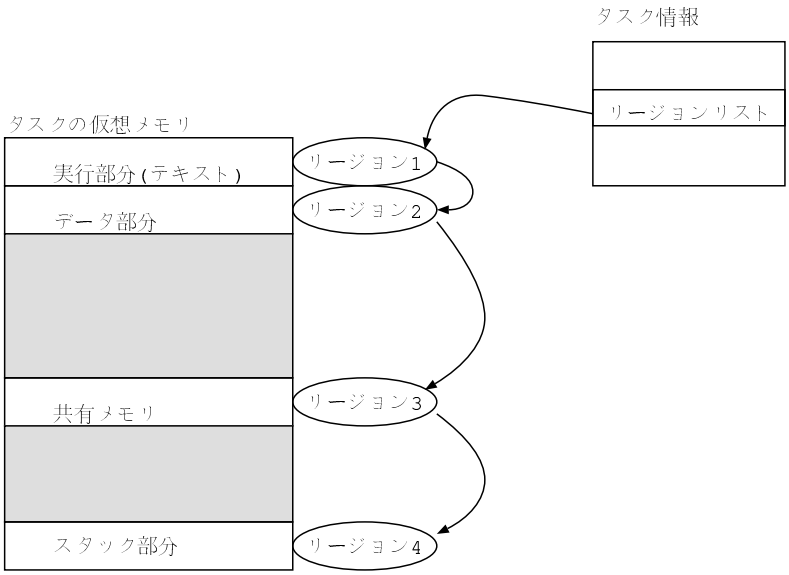

図 4.4: B-Free での仮想記憶のモデル

1. リージョンごとに permission が指定できる。そのことによって、テキストは実行するだけで読み書きできないなどの指定ができる。すべてひとつのリージョンにしてしまうと、permission は最少公倍数的なものになってしまうだろう (つまり、読み/書き/実行のすべてを許可した状態になってしまう)。

2. リージョンを仮想空間の中で離して置くことによって、リージョンの大きさを広げることができる。成長するリージョンにはヒープ、スタックなどがあります。

逆に、複数タスクが 1 つのリージョンを所有することもできます。この場合、複数のタスクから所有されるリージョンは、共有メモリとなります (図 4.5)。

B-Free OS では、デフォルトでデータを共有することはしません。しかし、プログラムの実行部分についてはデフォルトで共有します。これは、プログラムの実行部分は大抵の場合変更しないため、共有しても他のプロセスに影響をおよぼすことがないからです。プログラムの実行部分を変更するような場合、リージョンを共有しないようにシステムに要求する必要があります。もし、共有しているプログラムの実行部分を変更しようとした場合、メモリの保護違反となりプログラムは、強制終了します。

複数のリージョンが、仮想空間の中で重なりあうことはできません。

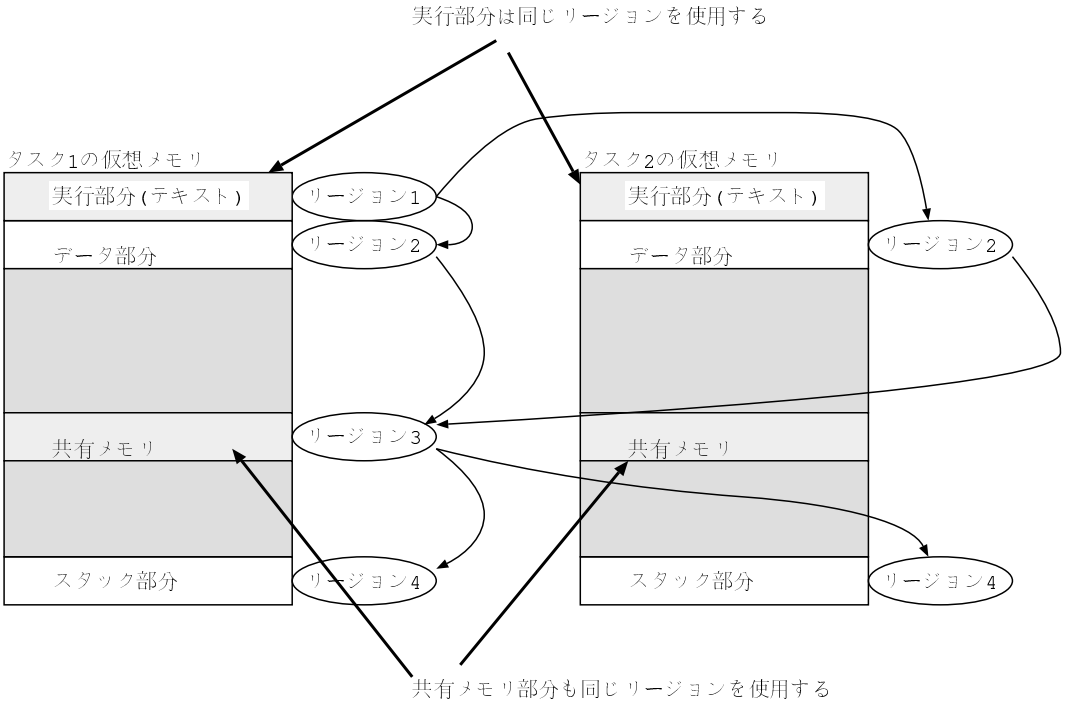

図 4.5: 複数タスクからの共有

4.7.3　リージョン情報

4.7.4　リージョンを管理する情報

リージョンは、一言でいうと任意のタスクの仮想空間の一部の領域です。リージョンを管理するための情報には、次の種類があります。

(1) リージョンで管理する仮想空間領域

リージョンが管理する仮想空間は次の 3 つのパラメータで表します。

- リージョンの開始仮想アドレス
- 最少サイズ
- 最大サイズ
- 大きさが変わらないリージョンの場合には、最少サイズ、最大サイズは同じ値となります。

(2) permission

仮想メモリ中のページの読み書きの許可状態を表ります。

- 実行可/実行不可
- 書き込み可/書き込み不可
- 読み込み可/読み込み不可

(3) 物理メモリのマッピング情報

リージョンの中のページのうちどれが物理メモリとマッピングしているか、マッピングしていた場合には物理メモリ (ページ) 番号を記憶します。

(4) メモリフォールトハンドラ

メモリフォールトが発生した場合、どのように処理するかを指定する情報です。

4.7.5　リージョンの操作

タスクは、リージョンの情報を直接操作することはできません。そのためリージョンの内容を変更する場合、中心核 (ITRON) のシステムコールを実行する必要があります。

中心核のもつリージョン操作関数を表 4.11 に示します。もともと ITRON では、仮想記憶操作については定義していません。そのため、リージョン操作システムールは ITRON で規定している独自システムコールとしてシステムコール名の最初に 'v' がつきます。

表 4.11: リージョン操作関数一覧

システムコール名	機能
vcre_reg	リージョンの生成
vdel_reg	リージョンの削除
vmap_reg	リージョンのマップ
vunm_reg	リージョンのアンマップ
vdup_reg	リージョンの複製を作る
vprt_reg	リージョンのプロテクト情報の設定
vshr_reg	タスク間でのリージョンの共有
vput_reg	リージョンへの書き込み
vget_reg	リージョンからの読み込み
vsts_reg	リージョンの情報

これらのシステムコールは、リージョンの情報をアクセスするだけで CPU のメモリ管理機能には影響を与えないものもあります。

4.7.6 物理メモリの割り付け

4.7.7 ページフォールト処理

ユーザプログラムが物理メモリにマップしていない仮想ページにアクセスした場合、ページフォールトが発生します。

ページフォールトが発生した場合、中心核は次の処理を行います。

- ユーザのページフォルトハンドラを呼び出す。

第5章　LOWLIB

LOWLIB (低レベルライブラリ) は、ユーザプログラムに対してシステムコールインタフェースを提供するための層です。

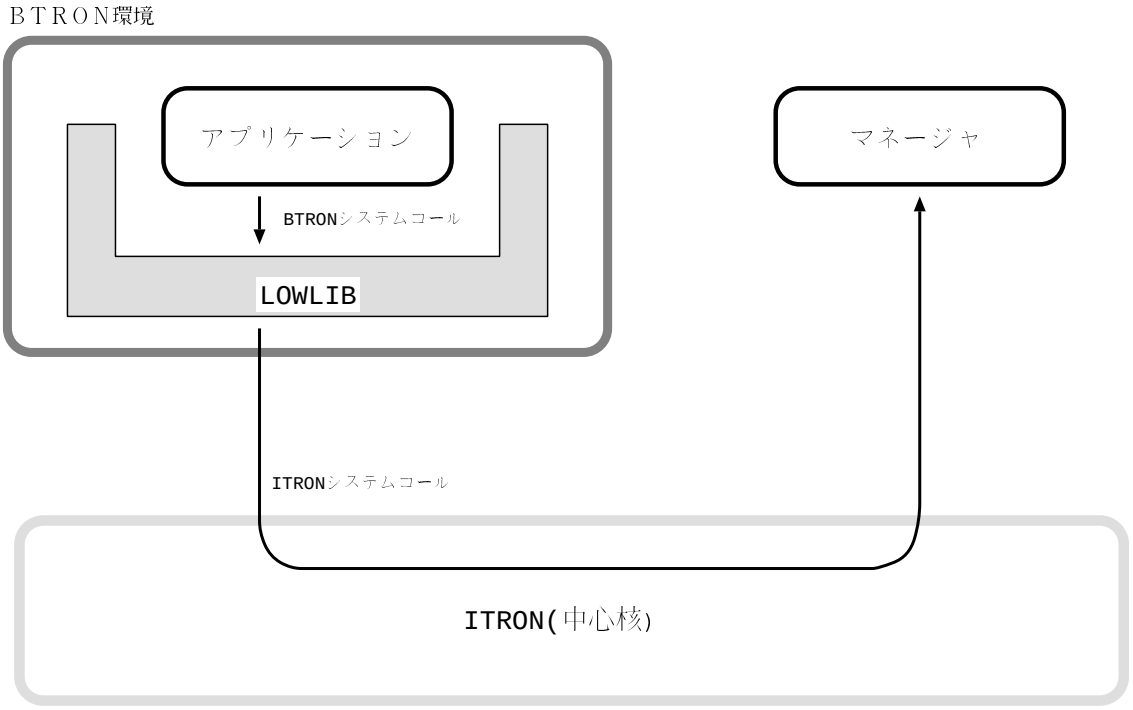

図 5.1: LOWLIB の役割

LOWLIB は、次の示す働きをします。

- アプリケーションの動作環境の初期化。
- システムコールをアプリケーションに提供。
- マネージャ群との通信を行う。

LOWLIB は、BTRON/POSIX という環境毎に、別々にもつことになります。そのために、LOWLIB をマッピングするための中心核のシステムコールがあります。

表 5.1: LOWLIB 用システムコール (中心核)

システムコール名	機能
lod_low	指定した LOWLIB をロード/マッピングする。
uld_low	指定した LOWLIB をアンロードする。
sts_low	LOWLIB の情報を取り出す。

5.1 動作環境の初期化

中心核がサポートするタスク生成の機能だけでは、アプリケーションプログラムを動作させることはできません。

LOWLIB は、中心核のタスク生成の機能ではサポートしていないアプリケーションプログラムが動作するための環境を初期化します。

具体的には、次の処理を行います。

- ユーザモードで動くためのスタック領域を確保します。
- システムコール用のトラップハンドラを登録します。
- コードおよびデータ用の領域を仮想空間上に確保します。

これらの処理を行った後に、ユーザプログラムのエントリアドレスへジャンプします。このとき、カーネルモードからユーザモードへの遷移を行います。

5.2 システムコールの実行

結局のところアプリケーションにとっての環境というのは、システムコールの機能によって決定されます。

LOWLIB は、アプリケーションプログラムにシステムコールを提供することによって、アプリケーションの動作環境を提供します。

システムコールの処理を行うために、LOWLIB 層にシステムコールのエントリテーブルをもっています。

また、システムコールの実行に必要な情報についても LOWLIB は各プロセス毎にもっています。

システムコールの実行は次のように行います。

1. ユーザプログラムがシステムコールを呼び出す (CPU のトラップ命令を実行)。
2. LOWLIB のトラップハンドラを実行。
3. LOWLIB は、システムコール番号からシステムコール関数を選択、実行する。
4. システムコールの実行後、ユーザプログラムへ戻る。

5.3 BTRON 環境での LOWLIB(obsoleted)

BTRON 環境での LOWLIB は、次のようなソース構成になっています。

表 5.2: LOWLIB のソース構成

ソース名	内容
`lowlib.c`	LOWLIB の main 関数部分。
`entry.c`	システムコールのエントリ部分
`syscalls/`	システムコール毎に定義している関数の入っているファイル群
`misc.c`	その他のファイル

これらの他にライブラリとして、`src/kernlib/libkernel.a` をリンクします。

5.3.1 LOWLIB/BTRON の初期化処理

BTRON 環境用の LOWLIB は、`lowlib_start()` という関数から実行をはじめます。

`lowlib_start()` は、BTRON プロセスを動作させるために必要な、次の初期化処理を行います。

- ユーザプロセス用の Region を生成。

  ユーザプロセスはデフォルトで、次の Region をもちます。

  - コード用 region (読み込みと実行のみ可能)
  - データ用 region (読み込みと書き込みのみ可能)
  - ヒープ用 region (読み込みと書き込みのみ可能：この region は大きさが変化する)

- スタック用 region (読み込みと書き込みのみ可能)

- スタック領域の確保 (物理メモリ)。
- 非同期動作用のタスクを生成 (メインに動作するタスクは、親プロセスが生成します)。実行します。
- プロセス情報をプロセスマネージャに登録します。
- システムコール用のトラップハンドラをプロセス毎に存在するトラップベクタに登録します。
- ユーザプログラムのエントリルーチンへジャンプします。

5.3.2 LOWLIB/BTRON のシステムコールの処理

システムコールのエントリルーチンは、次の 3 つの引数をもちます。

sysno システムコール番号。

uargp ユーザスタックの先頭アドレス。

errnop システムコールを実行した結果のエラー番号が入る領域のポインタ。

エントリルーチンの動作は、次のようになります。

```
entry(int sysno, void *uargp, int *errnop)
{
  <システムコール番号のチェック>

  <システムコール番号で指定しているシステムコール関数を呼び出す>
  <システムコール関数は、配列 syscalls[] に登録している>

  <システムコールの実行結果を *errnop に入れる>

  <return>
}
```

第6章　周辺核

6.1　周辺核の探索 (obsoleted)

周辺核は、中心核 (ITRON) を使って、B−Free OS の中でも重要な機能 — BTRON API を提供します。

周辺核で提供する機能には、次の種類があります。

プロセス管理
: 中心核の提供するタスク機能を使って、ファイルの管理情報などを追加した実行単位 — プロセスを管理します。

メモリ管理
: BTRON プロセスが扱うメモリを管理します。周辺核では、2 種類 — ローカル、共有メモリ — のメモリを提供します。

ファイル管理
: 可変長レコードによるファイル機能を提供します。ファイル自体には、BTRON の特徴である仮身・実身の機能はありません[1]。

  B−Free OS では、次のような複数のファイル形式を使用することができます。

  - BTRON FD ファイルシステム
  - *B-Free Standard File System*
  - `MS-DOS` ファイルシステム

イベント管理
: ポインティングデバイスからのイベントを管理します。

デバイス管理
: 周辺機器を操作するデバイスドライバを管理する機能です。デバイスドライバをロード・アンロードすることができます。(デバイスドライバ自体は、ITRON タスクとして動作します)

時間管理
: 時間に関係する機能を管理します。

[1] 実身や仮身は、仮身・実身マネージャが提供する機能です。

システム管理

上記の管理機能に属さない、「その他の」機能です。

6.2 周辺核の構造 (obsoleted)

周辺核、すなわち B−Free OS での BTRON API を処理する部分は、決して巨大なプログラムではありません。

周辺核は、図 6.1 に見るように単機能なプログラムの集まりです。

周辺核と上位層 (外核とアプリケーション) は、中心核の IPC 機能によって通信します。

周辺核を構成する要素を次のリストに示します。

プロセスマネージャ

BTRON プロセスを管理する。プロセスは、ITRON タスクと結びついています。プロセス管理サーバでは、プロセスに付随する情報を管理します。また、プロ セスのユーザ情報も管理し、特権レベルによって資源にアクセスできるかどうかの判断も行います。

メモリマネージャ

仮想メモリ情報を管理します。仮想メモリ機能の中でハードウェアに依存する機能については、ITRON で管理します。メモリ管理サーバでは、メモリが足りなくなった場合の物理メモリの解放機能 (ページアウト) や、物理メモリに結びついていない仮想メモリ領域をどのように物理メモリに結びつける (マッピング) かの方針を決定します。

ファイルマネージャ

BTRON のファイルシステムに関係する機能を提供します。

B−Free では、ファイル形式を複数取り扱うことができるので、ファイル管理サーバでは、各ファイル形式ごとにあるファイル管理プログラムを統合します。

デバイスマネージャ

周辺機器を制御するプログラム、すなわちデバイスドライバの管理を行います。

デバイスには各々名前が付いています。デバイス管理サーバでは、デバイス名とデバイスドライバの持つ通信用のポートを結びつけます。

6.3 プロセスマネージャ(obsoleted)

6.4 ファイルマネージャ(obsoleted)

6.5 メモリマネージャ(obsoleted)

メモリマネージャは、仮想メモリを操作するためのマネージャです。

B−Free /OS では、仮想メモリベースの OS です。つまり、B−Free /OS は、ページ単位での仮想メモリを扱うことができます。

6.5.1 仮想記憶の概念 (obsoleted)

— B−Free /OS での仮想記憶とはどういうものなのか。

仮想記憶機能とは、物理メモリに依存しない仮想的なメモリを扱う機能のことをいいます。

物理メモリのみを使用する実記憶ベースの OS の場合、実メモリのサイズを越えてメモリを使用することはできません。

例えば、実メモリが 4M バイトの大きさをもつシステムの場合、5M のメモリを消費するアプリケーションを動かすことはできません。

仮想記憶の機能をもつ OS の場合、仮想記憶機能を使うことによって物理メモリのサイズを越えた記憶容量をもつことができます。仮想記憶では、物理メモリのサイズを越えた分の記憶領域を 2 次記憶装置にもつことにより、アプリケーションに対して物理メモリのサイズを越えたメモリをもっているように見せます (図 6.2)。

仮想記憶の機能を実現するために、メモリマネージャは次の処理を行います。

- ページイン処理 二次記憶に追い出した情報を物理メモリに戻します。このとき、物理メモリが空いていない時には、物理メモリに空き領域を作ります。ページイン処理は、アプリケーションが物理メモリにないページのアドレスをアクセスしたときに発生するページフォールトを契機にして実行します。
- ページアウト処理 物理メモリにマッピングしているが、使用していないページの内容を二次記憶装置に追い出します。
- 仮想記憶ページ情報の管理 仮想記憶ページの情報を管理します。管理する情報は次のとおりです。
  - 仮想記憶ページが物理メモリにマッピングされている場合、物理メモリのアドレスを記憶します。

– 仮想記憶ページが二次記憶装置にページアウトされているときには、二次記憶装置のどこにページアウトしたかという情報を管理します。

6.5.2 ページイン処理 (obsoleted)

6.5.3 ページアウト処理 (obsoleted)

6.5.4 仮想メモリマネージャのメッセージ (obsoleted)

メモリマネージャが受け付けることのできるメッセージは、次のものがあります。

表 6.1: メモリマネージャの受けつけるメッセージ

メッセージ	処理の内容
VM_FAULT	ページフォールトの通知を行います。
VM_CREATE	仮想メモリ領域の生成を行います。
VM_SHARE	仮想メモリ領域をプロセスの間で共有します。
VM_REMOVE	仮想メモリ領域を削除します。
VM_READ	仮想メモリ領域の内容を読み取ります。
VM_WRITE	仮想メモリ領域にデータを書き込みます。
VM_UNSHARE	仮想メモリ領域の共有を解除します。
VM_MAP	仮想メモリ領域に物理メモリをマップします。
VM_UNMAP	仮想メモリ領域に物理メモリをマップします。
VM_LOCK	仮想メモリ領域をロックします。ロックしたメモリ領域は、ページアウトされなくなります。

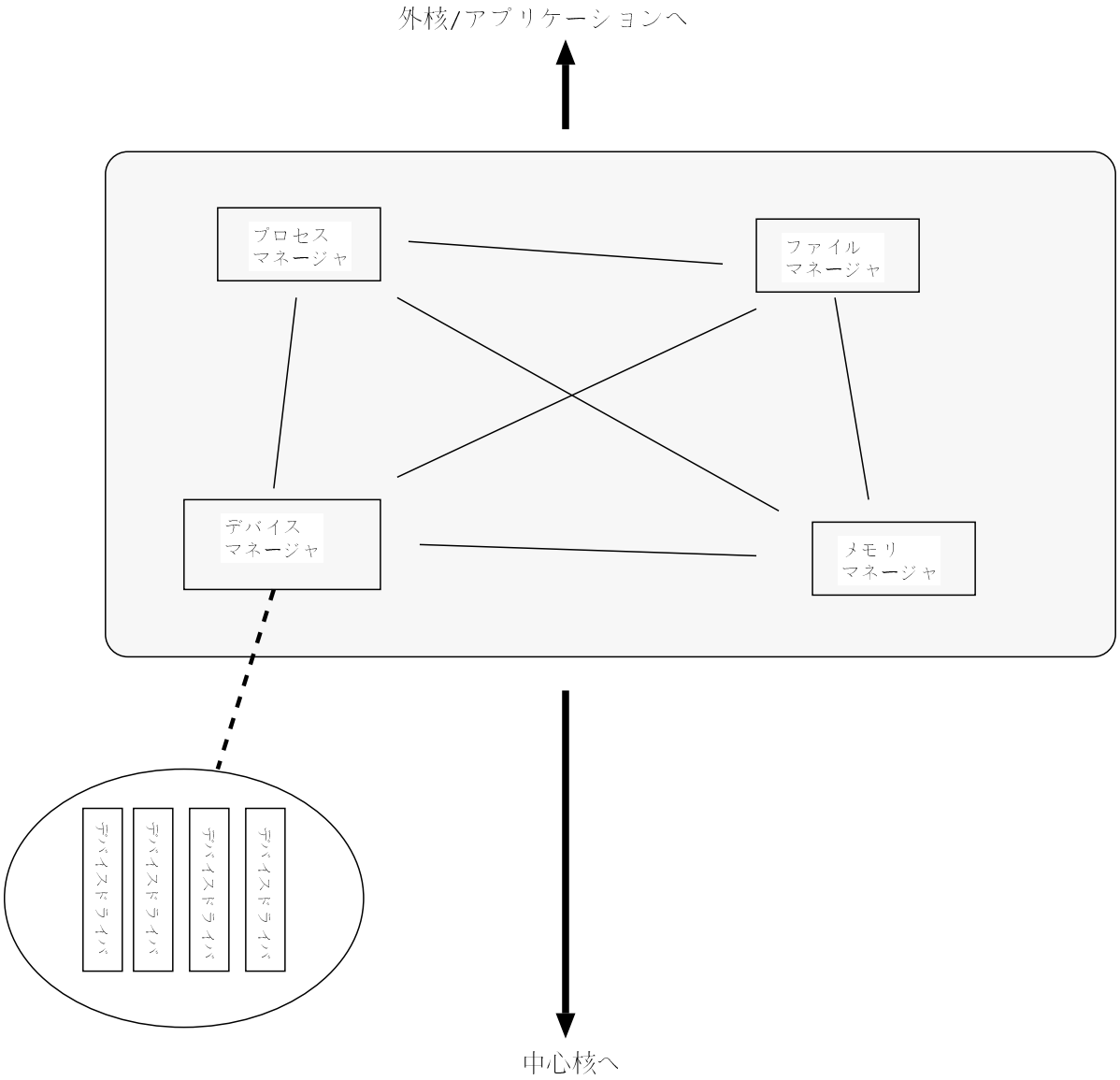
外核/アプリケーションへ
プロセス
マネージャ
ファイル
マネージャ
デバイス
マネージャ
メモリ
マネージャ
デバイスドライバ
デバイスドライバ
デバイスドライバ
デバイスドライバ
中心核へ

図 6.1: 周辺核の構造

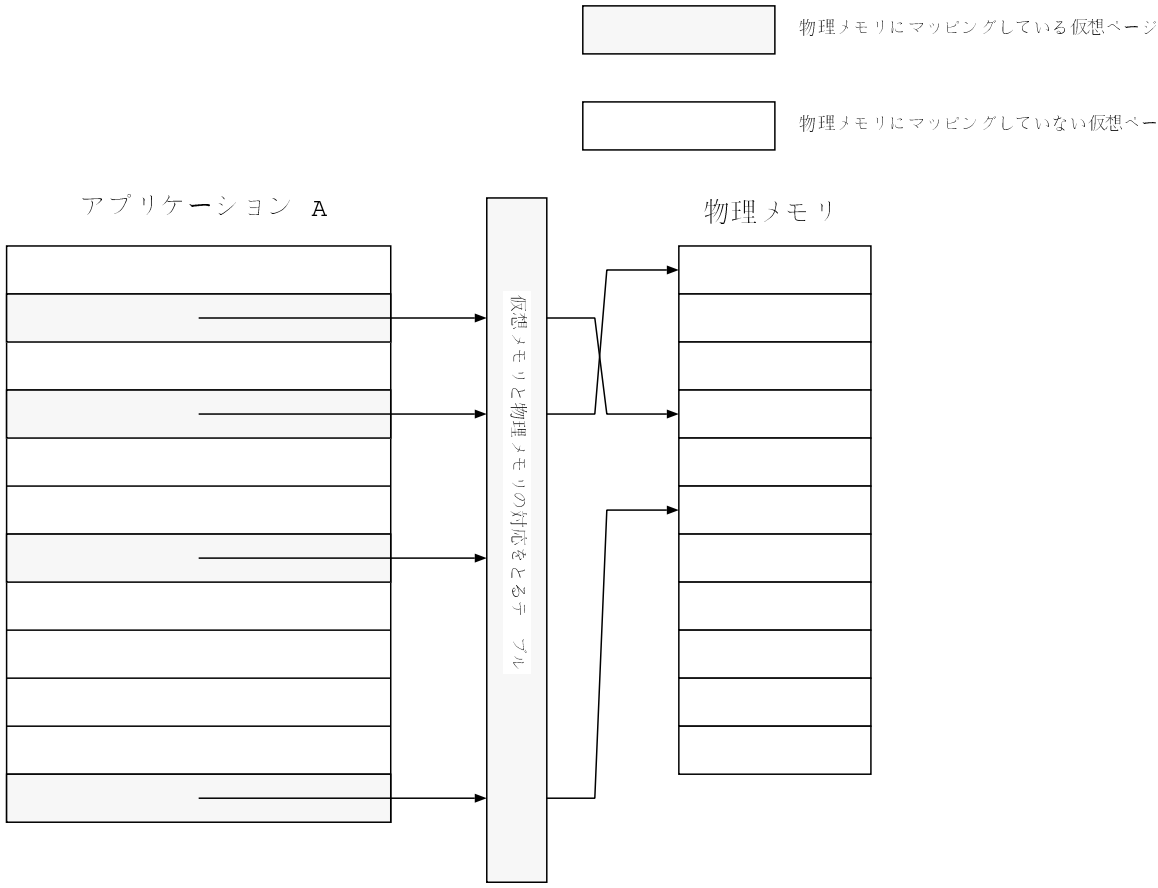
物理メモリにマッピングしている仮想ページ
物理メモリにマッピングしていない仮想ペー
アプリケーション A
物理メモリ
仮想メモリと物理メモリの対応をとるテーブル

図 6.2: 仮想記憶の概念

第7章　デバイス管理

ハードウェアデバイスを使用するためには、デバイスドライバが必要です。
この章では、B−Free /OS で使用するデバイスドライバについての説明とデバイスドライバ自体を管理するためのマネージャであるデバイスドライバマネーッジャについての説明を行います。
デバイスドライバマネージャは、BTRON 環境だけではなく、POSIX 環境でも使用する環境に依存しないマネージャです。

7.1　B−Free にとってのデバイスドライバとは何か (obsoleted)

7.1.1　デバイスマネージャとデバイスドライバ (obsoleted)

デバイスドライバというのは、ハードウェアデバイス (FD や HD などのストレージデバイスやキーボード、ディスプレイなどの入出力装置など) を管理するためのソフトウェアです。

B−Free では、デバイスドライバは、個々のデバイスを管理するためのデバイスドライバと、それらのデバイスドライバを管理するデバイスマネージャという 2 つの構成要素から成り立っています。

図 7.1: デバイスドライバとデバイスマネージャの関係

7.1.2　論理デバイス名

B−Free の個々のデバイスドライバは、固有の論理デバイス名をもっています。論理デバイス名は、アプリケーションがデバイスを指定するときに使用します。

論理デバイス名は、3 つの要素からなりたっています。

デバイスの種類

デバイスの種類を表す名前です。

ユニット

ひとつのデバイスドライバが、複数の物理デバイスを扱う場合にどのデバイスかを指定するための名前です。英文字 1 文字が使用されます。

サブユニット

ひとつのユニットが複数に分割できる場合の分割した個々の要素を表すための名前です。最大 3 桁の数字で表現します。また、サブユニット全体 (例えば HD 全体など) をあらわすためには、'$' の文字を使用します。

HD のパーティションなどがサブユニットにあたります。

論理デバイス名の例を 図 7.1.2 に示します。

ｆｄａ	1 番目のフロッピィディスクデバイス
ｈｄｂ＄	2 番目の HD デバイスのサブユニット全体
ｈｄｂ１	2 番目の HD デバイスの 1 番目のサブユニット (パーティション)

図 7.2: 論理デバイス名の例

7.2 デバイスマネージャ(obsoleted)

デバイスマネージャは、デバイスドライバの管理を行います。デバイスマネージャは、ITRON レベルでのタスクとして動作します。

表 7.1 に示すリクエストを受け付けます。

表 7.1: デバイスマネージャのリクエスト一覧

リクエスト名	処理内容
dev_define	デバイスドライバの登録
dev_remove	デバイスドライバの削除
dev_find	デバイスドライバの検索
dev_load	デバイスドライバのローディグ
dev_unload	デバイスドライバのアンローディグ

7.2.1 dev_define — デバイスドライバの登録

dev_define () は、デバイスドライバの情報をデバイスドライバマネージャに登録します。

登録する情報は次のとおりです：

- (ユニット名/サブユニット名を除いた) デバイスドライバの名前
- ユニット名の最大値
- サブユニット名の最大値
- 要求受けつけ用のメッセージバッファ ID
- 次に示すデバイスドライバ属性
  - ドライバのタイプ (ブロック or キャラクタ)
  - ドライバに使用するバッファサイズ
  - 排他的使用となるか非排他的使用 (複数のプロセス間で共有可) となるか

これらの情報は、デバイスドライバマネージャが内部に持っているデバイスドライバの管理テーブルに記録します。

7.2.2 dev_remove — デバイスドライバの削除

dev_remove () は、デバイスドライバマネージャの中に記録してあるデバイスドライバの情報のうち、指定したものを削除します。

削除するデバイスドライバの指定は、名前で行います。

7.2.3 dev_find — デバイスドライバの検索

dev_find () は、デバイスドライバの検索を行います。

具体的には、デバイスドライバを使用するために要求を送るメッセージバッファID を調べるために使用します。

検索に使用するキーは、デバイスドライバの名前です。

デバイス名の中にユニット名/サブユニット名が含まれていた場合、デバイスの種類のみを取り出して、デバイスドライバ登録テーブルを検索します。

たとえば、デバイスドライバ名として次の名称:

`hda1 ..... HD デバイスのユニット 0/サブユニット 1 を指定。`

を指定した場合、ユニット名/サブユニット名を除いた次の名称に変換して検索します。

hd ..... HD デバイスを指定 (ユニット名/サブユニット名を除いた名前)

デバイスマネージャ自身は、デバイスドライバの登録しか行いませんが、BTRON OS 環境 あるいは POSIX OS 環境のサーバによって、動的にデバイスドライバをロード/登録することも可能です。その場合、各 OS 環境の API によってファイルシステムからデバイスドライバをロードしタスクとして動作できるようにしてから、デバイスドライバマネージャに登録するということになります。

7.2.4 デバイスドライバのロード (obsoleted)

指定したファイルからデバイスドライバを読み取り、メモリ上にロードします。

7.2.5 デバイスドライバのアンロード

dev_load によってファイルから読み込んだデバイスドライバをメモリ上から削除します。
当然ですが、この処理を実行したあとはデバイスドライバは使用できません。

7.3 デバイスドライバの機能 (obsoleted)

デバイスドライバが受信し、処理するパケットの種類は表 7.2 のとおりです。
B−Free では、デバイスドライバもひとつのタスクとして動作しています。
デバイスドライバの概略をリストにすると次のようになります。

表 7.2: ドライバの処理するパケットの種類

種類	処理
DeviceInit	デバイスドライバの初期化
DeviceExit	デバイスドライバを終了させる
DeviceOpen	デバイスをオープンする
DeviceClose	デバイスをアンロックする
DeviceRead	デバイスから情報を読み出す
DeviceWrite	デバイスに情報を書き込む
DeviceControl	デバイス固有の制御を行う
DeviceProbe	デバイスが実際にあるか探る

```
driver_main ()
{
  < ドライバの初期化 >
    ・割り込みハンドラの登録。
    ・インタフェース LSI の初期化
    ・ドライバで使用するテーブル類の初期化
    ・要求受信用のメッセージバッファをドライバマネージャに登録

  /* 要求受けつけループの実行 */
  for (;;)
    {
      <要求パケットの受信>
      switch (パケットタイプ)
        {
      case DeviceInit:   <デバイスドライバの初期化>
      case DeviceExit:   <デバイスドライバの終了>
      case DeviceOpen:   <デバイスのオープン処理>
      case DeviceClose:  <デバイスのクローズ処理>
      case DeviceRead:   <デバイスの read 処理>
      case DeviceWrite:  <デバイスの write 処理>
      case DeviceControl:<デバイスの Control 処理>
      case DeviceProbe:  <デバイスのプローヴ処理>
    }
}
```

7.4 デバイスドライバが便利に使える関数群 (obsoleted)

デバイスドライバが共有する資源としては、DMA [1]や割り込みなどがあります。

この章では、これらの資源をアクセスするための関数について説明します。

なお、これらの関数は、libkernel.a に入っています。

7.4.1 DMA の制御 (obsoleted)

PC9801 では、DMA の制御用として μ PD8237A (DMA コントローラ) を使用しています。

この LSI では、4 つの DMA 用のポートがありますが、ひとつは PC9801 のアーキテクチャ上の問題 (メモリリフレッシュ用に使用) でデバイスドライバ用としては使用できないため、使えるのは 3 つということになります。

PC9801 では、3 つの DMA をそれぞれ次のデバイスに割り当ててあります。

表 7.3: DMA ポート

DMA チャネル番号	使用機器
0	5 インチハードディスク
1	メモリリフレシュ (デバイスドライバは使用できない)
2	1MB FDD
3	640K FDD

これら以外にも、デバイス自体に DMA コントローラを搭載しているものがあります (SCSI ボードなど)。

B−Free では、DMA の制御を行うための次の関数を用意してあります。

dma_setup(obsoleted)

dma_setup は、DMA を使用するための前準備を行います。具体的には、DMA コントローラに以下のパラメータを設定します。

- モード設定
- チャネルマスク値設定

[1]Direct Memory Access

- 転送アドレス設定
- バンク番号指定
- 転送カウント設定

dma_setup を実行すると、DMA コントローラは DMA 要求を受けつける状態になります。その状態で、周辺デバイス (FD ドライブならば FDD コントローラ) が転送要求を DMA コントローラに送ると DMA 転送が発生します。

dma_setup は、次のようにして呼び出します。

dma_setup (void *addr, W mode, W length, W mask)

addr 転送アドレス

mode 転送モード

length 転送長

mask チャネルマスク値

`dma_setup()` を実行すると DMA コントローラによる DMA 転送の準備が行われます。この後で、DMA コントローラに転送開始のイベントが上がることによって DMA 転送が行われます。このイベントは、通常各制御 LSI (例えば FDD の場合には μ PD765) が、送ります。

7.4.2 割り込み制御 (obsoleted)

PC9801 の場合、周辺機器のために表 7.4 に示すような割り込みエントリが用意してあります。

表 7.4: 周辺機器のための割り込みエントリ

割り込みエントリ番号	使用するデバイス
0x08	タイマ (8053)
0x09	キーボード (8251A)
0x0A	CRTV (μ PD7220 (マスタ))
0x0C	RS-232C (8251A)
0x10	セントロニクスプリンタ (9255A)
0x11	ハードディスク
0x12	640KB FD
0x13	1MB FD
0x15	マウス

割り込みを使用するためには、中心核のシステムコールを使って割り込みハンドラを登録する必要があります。割り込みハンドラを登録するための中心核のシステムコールは、**def_int ()** です。

ER def_int (UINT intno, T_DINT pk_dint);

intno 割り込みのエントリ番号 (表 7.4 に示した番号) を指定します。

pk_dint 割り込みハンドラのアドレスを指定します。

7.5 HD ドライバ (obsoleted)

HD ドライバは、PC9801 版の B−Free では、SASI 版と SCSI 版の 3 種類があります。

7.6 FD ドライバ (obsoleted)

PC9801 の FDD は、μ PD765A というコントロール LSI を使って制御しています。

7.7 RS232C ドライバ

7.8 コンソールドライバ (obsoleted)

第8章　外核

第9章　ユーザインタフェース

ユーザインタフェースについて説明する章。(ウィンドウや仮身実身など)

第10章　POSIXインタフェース

— UNIX は単なるオペレーティングシステムではなく、プログラミングの思想なのだ。
Don Libes & Sandy Ressler 「Life with UNIX」

B−Free /OS の基本構造がマイクロカーネルアプローチを取っていることから、ユーザからみたシステム環境は、複数個もたせることが可能です。

ユーザからみたシステムインタフェースのうち、メインとなっているのは、これまで説明してきた BTRON/OS です。

BTRON とは別のシステム環境が B−Free にはひとつあります。それが、この章で説明する POSIX インタフェースです。

10.1　Posixインタフェース (obsoleted)

POSIX [1] は、IEEE [2] が規定したオペレーティングシステムのインタフェースです。

基本的には、これまでの UN*X と呼ばれてきた OS の最大公約数といえます。

B−Free の POSIX 環境には、以下の API があります。

access	execv	getgid	mkdir
chdir	execve	getgrgid	mkfifo
chmod	execvp	getgrnam	open
chown	_exit	getgroups	opendir
close	fcntl	getlogin	pause
closedir	fork	getpgrp	pipe
creat	fseek	getpid	read
dup	fstat	getppid	readdir
dup2	getcwd	getuid	remove
execl	getegid	kill	rename
execle	getenv	link	rewind
execlp	geteuid	lseek	rewinddir

[1] Portable Operating System Interface for Computer Environments
[2] 電気電子技術者協会

rmdir	sigemptyset	sleep	unlink
setgid	sigfillset	stat	utime
setpgid	sigismember	time	watipid
setsid	siglongjmp	times	write
setuid	sigpending	ttyname	mount
sigaction	sigprocmask	tzset	umount
sigaddset	sigsetjmp	umask	
sigdelset	sitsuspend	uname	

POSIX 環境は、BTRON 環境と同様に POSIX マネージャ と LOWLIB そしてユーザプログラムからできています。

この章の残りは次の構成になっています。

- POSIX マネージャについての説明
- POSIX システムコールをサポートした POSIX 用 LOWLIB の説明。
- POSIX プログラム (ユーザプログラム) の構成およびライブラリについての説明。

10.2 POSIX マネージャ(obsoleted)

POSIX の OS 環境では、次の 4 つの POSIX マネージャが動きます[3]。

ファイルマネージャ (FM)	POSIX のセマンティクスに沿ったファイル管理機能を提供
プロセスマネージャ (PM)	POSIX プログラムのプロセス管理を行う
メモリマネージャ (MM)	仮想メモリ管理を行う
デバイスマネージャ (DM)	デバイスドライバの管理を行う

10.2.1 ファイルマネージャ (FM)(obsoleted)

POSIX 環境でのファイル (obsoleted)

ファイルマネージャは、POSIX が規定しているセマンティクスに従ったファイルシステムの管理機能を提供します。

POSIX では、ファイルとして次のものを定義しています。

- いわゆる通常のファイル。中には、構造のないバイトの列。
- ディレクトリ。ファイルへのポインタが入る。
- スペシャル (デバイス) ファイル。入出力機器とのインタフェース。

[3] ここでいうマネージャとは、BTRON 環境での周辺核のことです。

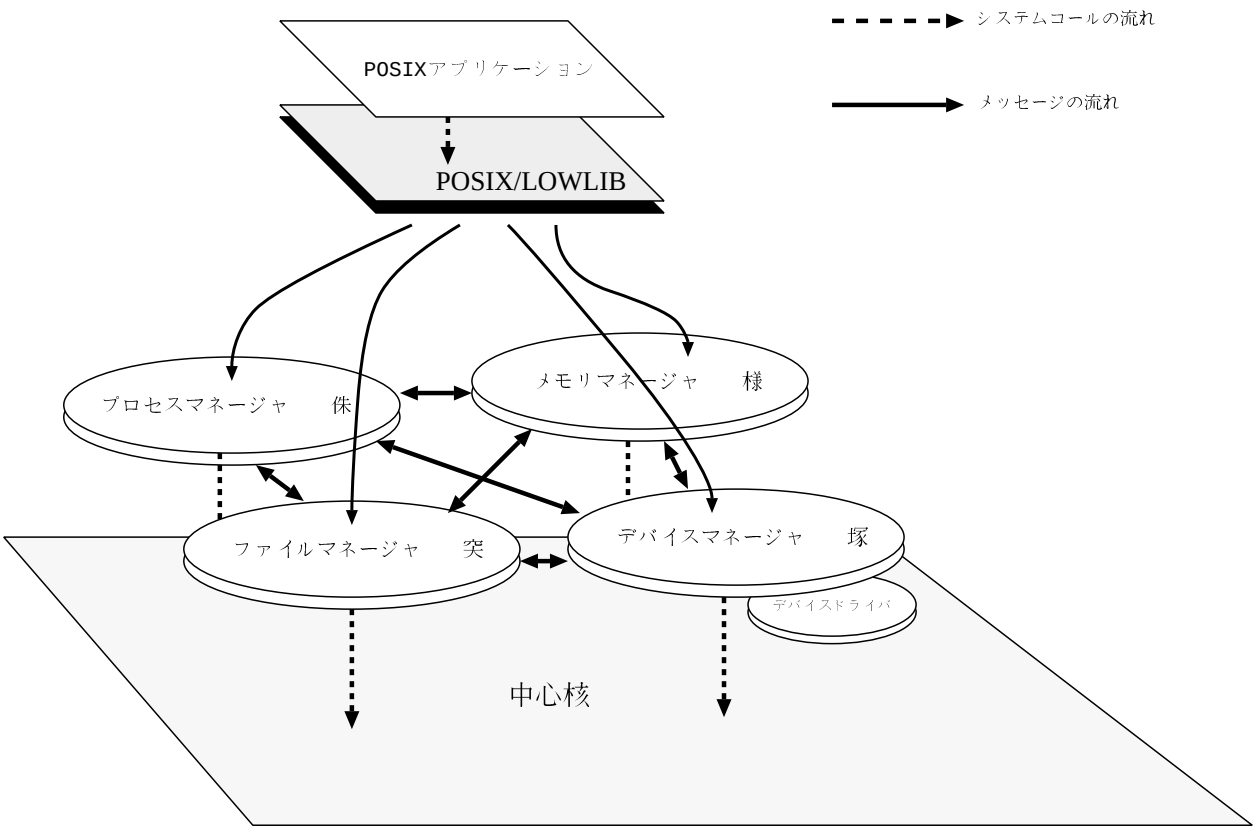

図 10.1: POSIX 環境の構成

パス名　ファイルの位置はパス (名) によって表現します。

パスは、ファイルの名前をならべたものです。個々のファイル名は、'/' によって区切ります。

'/' からはじまるパス (名) を絶対パス (名)。

'/' からはじまらない (ファイル名からはじまる) パス (名) を相対パス (名) と呼びます。

'/' の 1 つだけのパス名は、ルートディレクトリ (root directory) という特別のディレクトリを示しています。ルートディレクトリは、ファイルシステムの始点です。

絶対パスの場合ルートディレクトリからパス名をたどっていくことになります。

相対パスの場合、パスの基点はカレントディレクトリからとなります。カレントディレクトリというのは、パス名を指定したプロセスが今いる位置を示しています。カレントディレクトリは、専用の API によって変更することもできます。

```
/foo/bar/baz ..... 絶対パスの例
bar/baz ........... 相対パスの例
./../foo/bar/baz ..... これも相対パスの例
```

図 10.2: パスの例

ファイル情報　POSIX 環境内でのファイルは、すべての (POSIX 環境内での) ファイルシステムで共通な属性情報をもっています。

属性は、`/usr/include/sys/stat.h` の中で定義している、stat 構造体で定義しています。

この構造体には、以下の情報が入ります。

- ファイルの種類
- ファイルの識別番号 (I ノード番号[4])
- ファイルのリンク数
- 所有ユーザ/所有グループ ID
- ファイルのサイズ
- 作成/更新/アクセス日付

実際の stat 構造体を、図 10.3 に示します。

[4] B−Free の場合、サブファイルシステムがファイルシステム内でユニークな番号を割り振ります。

```
struct stat
{
  mode_t        st_mode;
  ino_t         st_ino;
  dev_t         st_dev;
  nlink_t       st_nlink;
  uid_t         st_uid;
  gid_t         st_gid;
  off_t         st_size;
  time_t        st_atime;
  time_t        st_mtime;
  time_t        st_ctime;
};
```

図 10.3: ファイル情報

BTRON では、ファイルは可変長レコードという構造をもっていましたが、POSIX 環境ではファイルは単なるバイトの列として扱います。

ファイルマネージャが関係するシステムコール (obsoleted)

ファイルマネージャが関係するシステムコールを 表 10.2.1 に示します。

access	ファイルにアクセスできるかをチェックします
open	ファイルを読み書きする準備を行います具体的には、ファイルマネージャは、指定したファイルを読み書きするタスクを (必要ならば) 生成し、読み出し元との通信を行うメッセージバッファを生成します
close	ファイルをクローズしますもし、ファイルをみているプロセスがひとつもないならば、読み書き用のタスクを除去します
execXX	ファイルをメモリ中に読み込み、実行します
read	ファイルの内容を読みます
write	ファイルにデータを書き込みます
lseek	ファイルのカレントポジション (読み書きの開始位置) を、指定された値に変更します
select	複数のファイルの入出力を同時に監視するためのシステムコールです
fcntl	ファイルを制御するためのシステムコールです。主にスペシャルファイルに対して使用します
rename	ファイル名を変更します
remove	ファイルを削除します
mkdir	ディレクトリを作成します
rmdir	ディレクトリを消去します
chown	ファイルの所有者を変更します
chgrp	ファイルの所有グループを変更します
access	ファイルのアクセス権をチェックします
stat	ファイルの管理情報を取得します
mount	ファイルシステムを接続する
unmount	ファイルシステムの接続を解除する

これらのシステムコールについては、他のマネージャと協調して処理する場合もあります。

マネージャの構造 (obsoleted)

POSIX/ファイルマネージャは、中心核の上で直接動く ITRON タスク (の集合) です。ITRON 上で動くことからわかるように、POSIX プロセスとして動くわけではありません。そのため、POSIX/ファイルマネージャは中心核のシステムコールを使って、メモリ取得などを行います[5]。POSIX 環境上では、POSIX/プロセスとして動作するのはユーザプログラムだけとなります。

[5]そのためのライブラリが libkernel.a です (Chapter B を参照してください)。

POSIX 環境のマネージャ群は、CPU のユーザモードで動作します。中心核は、安全のために、ユーザモードで動作するプロセスが (中心核の) システムコールを発行するのを禁止しています。そのため、ユーザプロセスは、LOWLIB を介してのみ中心核のシステムコールを使うようにしています。

ただし、LOWLIB を使う方法は、中心核の使用頻度の高いマネージャでは動作のためのコストが高くなるため、使用しません。POSIX/マネージャ群は、中心核の ITRON からの拡張機能のひとつである、タスク生成時の属性指定により直接、中心核のシステムコールを発行することができます。

POSIX/ファイルマネージャは、階層構造をもつ複数のタスクからできています。これらのタスクは、すべて一つの仮想記憶空間に収納します。そして、オープン中のファイルの情報などは、すべてのタスクで共有します。

まず、トップレベルにあるのが、すべてのファイルシステムを管理するファイルマネージャのタスクです。

ファイルマネージャ主タスク　ファイルマネージャ全体の管理をするタスクです。

オープンファイルタスク　オープンしているファイルを管理するタスクです。オープン中のファイルひとつにつき、ひとつのタスクが対応します。

さらにサブレベルとして、個別のファイルシステムに依存したモジュールがあります。POSIX 環境では、複数のファイルシステム形式が混在して使用できるようになっています。

ファイルマネージャの動きを簡略化すると次のリストのようになります[6]。

[6] ファイルマネージャ以外のマネージャについても大枠は同じです。

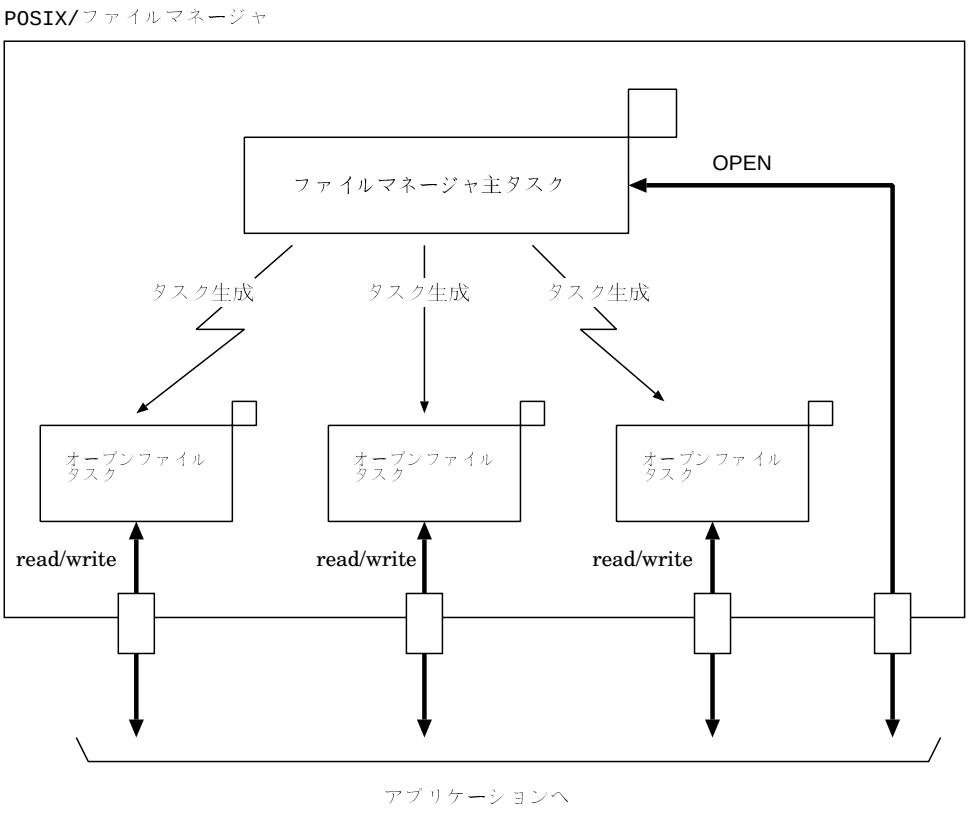

図 10.4: ファイルシステムマネージャの構造

```
main ()
{
  <ファイルマネージャ全体の初期化>

  for (;;)
    {
      <メッセージの受信>
      if (メッセージは正しい形式ではない)
        {
          <エラーを返す>
        }

      switch (受信したメッセージのタイプ)
        {
        case  FILE_NULL:
          <何もしない>
          break;
        case  FILE_TRAVERSE:
          file_traverse  (メッセージ);
          break;
          :
          :
        default:
          <認識できないメッセージタイプ -> エラーを返す>
          break;
        }

      <処理の結果を返答>
    }
}
```

POSIX ファイルマネージャの初期化 (obsoleted)

ファイルマネージャが起動すると、まず最初にファイルマネージャ内のテーブルを初期化します。

ファイルマネージャで管理しているテーブルとしては、次のものがあります。

open_file_table オープン中のファイルテーブルです。この中には、ファイルの識別子、管理用タスクの ID そしてメッセージ通信用のメッセージバッファ ID などが入っています。

buffer_table 読み書きを高速化するためにファイルマネージャはデータのバッファリングを行います。buffer_table はバッファの内容を管理します。

初期化が終了すると、ファイルマネージャはそのまま要求受け付け - 処理ループに入ります。

表 10.1: ファイルマネージャが処理するパケットの種類

FILE_NULL	何もしない
FILE_TRAVERSE	パス名を辿る
FILE_CLOSE	ファイルをクローズする
FILE_SELECT	デバイスの select
FILE_READ	ファイルの読み取り
FILE_WRITE	ファイルの書き込み
FILE_TRANC	ファイルのサイズを変更
FILE_GETATTR	ファイル属性を得る
FILE_SETATTR	ファイル属性を設定
FILE_ACCESS	ファイルのアクセス権をチェック
FILE_LINK	ファイルのハードリンク
FILE_MKDIR	ディレクトリを作成
FILE_RMDIR	ディレクトリを削除
FILE_MKSPEC	スペシャルファイルの作成
FILE_RMSPEC	スペシャルファイルの削除
FILE_CONTROL	ファイルの制御
FILE_MOVE	ファイルの移動 (名前の変更)
FILE_MOUNT	ファイルシステムをマウントする
FILE_UNMOUNT	ファイルシステムをアンマウントする

要求の受けつけ (obsoleted)

ファイルマネージャが受け付けるパケットの種類を表 10.1 に示します。

要求の受けつけは、次の `get_request()`[7] によって行います。

要求の処理 (obsoleted)

受信した要求メッセージは、まず `doit()` 関数に渡されます。
`doit()` 関数は、次の処理を行います。

(1) 要求メッセージの分解

ファイルマネージャに送られた要求メッセージを要求を処理する関数に渡すために、メッセージを分解します。

[7] libkernel.a で定義している関数です。

(2) 各要求に対応した処理を行う関数に分岐

要求メッセージの先頭に入っている要求の種類を見て、対応する関数を呼び出します。

そのとき、分解したメッセージの内容を引数として関数に渡します。

パスの辿り (traverse)(obsoleted)

ファイルを使用する時には、open システムコールによってパス名を指定し、該当するファイルをオープンします。

パスを辿る処理は次のように行います。

サブファイルシステム (obsoleted)

ファイルマネージャ自体は、物理媒体 (HD/FD など) に対するアクセスは行いません。物理媒体をアクセスするのは、サブファイルシステムというモジュールによって行います。

サブファイルシステムは、ファイルシステムのタイプごとに存在しています。POSIX 環境では、次のサブファイルシステムがあります。

DOS ファイルシステム MS-DOS ファイルシステム。FAT による管理を行う。

各サブファイルシステムは、ファイルマネージャと同じ実行モジュールにリンクしています。そのため、サブファイルシステムの処理を行うタスクは、ファイルシステムマネージャと同じタスクとなります (ファイルシステムマネージャから見ると、サブファイルシステムを処理するのは単なる関数となります)。

ただし、サブファイルシステムとファイルマネージャとは、表 10.2 示すインタフェースでのみやりとりを行います。表 10.2 に示したのはマクロ[8]ですが、サブファイルシステムは、このマクロに対応する関数インタフェースをもちます。

サブファイルシステムインタフェースは、ファイルマネージャのインタフェース (表 10.1) と同等のインタフェースをもちます。

[8] src/posix/usr/include/server/file.h に定義。

```
/*
 * パス名の辿り ..... 引数 path で指定したパスを順々に辿っていく。
 *                    最後の要素まで辿ったら、その要素に対応する
 *                    ポートを返す。
 */
ID
traverse(char *path)
{
  ID   tmp;
  char 残りのパス名;
  char 現在の要素;

  if (パス名が絶対パス)
     tmp = rootdir;
  else
     tmp = カレントプロセスのカレントディレクトリ;

  残りのパス名 = path;

  while (<のこりのパス名が NULL ではない>)
    {
      現在の要素 = <残りのパス名から先頭要素をひとつ取り出す>;
      <残りの要素は残りのパス名に入れる>;

      tmp = FS_LOOKUP (tmp, 現在の要素); /* パス名をひとつだけ辿る */

      < 必要ならばエラー処理を行う >

    }
  return (tmp);    /* 最後まで辿った */
}
```

表 10.2: サブファイルシステムインタフェース

インタフェース名	機能
FS_INIT	サブファイルシステムの初期化
FS_LOOKUP	パス名を要素ひとつだけ辿る。
FS_GETATTR	ファイルの属性を読み取る
FS_PUTATTR	ファイルの属性を書き込む (変更する)
FS_READ	ファイルの内容を読み取る
FS_WRITE	ファイルに情報を書き込む
FS_TRUNC	ファイルサイズを変更する
FS_MKDIR	ディレクトリを作成する
FS_RMDIR	ディレクトリを削除する
FS_MKNOD	スペシャルファイルを作成
FS_RMNOD	スペシャルファイルを削除
FS_LINK	(ハード) リンクを作成する
FS_REMOVE	ファイルを削除する
FS_MOUNT	ファイルシステムをマウントする
FS_UNMOUNT	ファイルシステムをアンマウントする

MS-DOS ファイルシステム (obsoleted)

MS-DOS ファイルシステムは、POSIX 環境でのサブファイルシステムのひとつです。

基本的には、MS-DOS の FAT ファイルシステムと同一ですが、次の点が拡張されています。

- 長いファイル名 (最大 256 文字) のサポート
- シンボリックリンクのサポート
- スペシャルファイルのシミュレート機能のサポート

これらの拡張機能は、MS-DOS ファイルシステム自体は変更しない形で拡張しています。そのため、従来の MS-DOS システムから読み書き可能となっています。

拡張方法　FAT ファイルシステムの拡張方法は、DOS との共有を行えるようにすることから、もとの FAT ファイルシステムの管理情報はそのまま使用する方式をとります。

拡張部分については、変換テーブルの入ったファイルを使用することにします (このファイルも通常の FAT ファイルです)。

変換テーブルのファイルは、各ディレクトリにひとつだけ存在することとします。ファイル名は、POSIX.TBL となります。

変換テーブルは次の情報が入ります。

- MS-DOS のファイル名 (POSIX ファイルマネージャが自動的に生成します)
- POSIX 環境から見えるファイル名
- 属性情報 (ファイルのパーミッション/ファイルの種類など)

具体的には変換テーブルの内容は次のようになります。

DOS001	長いファイル名の例	`<0777>regular`
DOS002	シンボリックファイルの例	`<0755>symbolic=../foo`
DOS003	長いファイルの例	`<0700>chrdev=0,0`

10.2.2 プロセスマネージャ(obsoleted)

POSIX プロセスマネージャは、POSIX 環境上で動いているプロセス (以下 POSIX プロセスと略記) を管理します。

各 POSIX プロセスは、プロセスマネージャ内では 構造体 process で表現されます。この構造体は POSIX プロセスマネージャ内でのみ使用するものです [9]。

```
struct process
{
  struct process      *prev;
  struct process      *next;

  enum proc_status    status; /* プロセスの状態を示す */
  pid_t                       pid;    /* プロセス ID */

  uid_t       uid;            /* プロセスが属する所有者 */
  gid_t       gid;            /* プロセスが属するグループ */

  ID  main_task;      /* ユーザプログラムのコードを実行するタスク */
  ID  signal_task;    /* シグナルの受信処理を行うタスク */
  ID  fifo_task;      /* パイプを使うときに使用するタスク */
  ID  alarm_task;     /* alarm システムコール用のタスク */

  ID  efile;          /* 実行ファイルを指しているメッセージポート
                       * コード部のページインのときに使用する。
                       */
};
```

つまるところ、POSIX プロセスマネージャはこの構造体情報を管理するのが仕事となります。

POSIX/プロセスマネージャの機能 (obsoleted)

POSIX 環境でのプロセスマネージャは、アプリケーションに対して次の機能を提供します。

[9] src/posix/usr/src/sys/server/PM/pm.h で定義

- 新しいプロセスの生成
- プロセスの終了処理
- プロセススケジューリング
- プログラムの実行 (exec)
- シグナルの処理
- インターバルタイマ

新しいプロセスの生成 (obsoleted)

POSIX 環境上で新しいプロセスを生成するのは、`fork` システムコールです。

アプリケーションが `fork` システムコールを実行すると、次の処理を行います。

(1) アプリケーションが `fork` システムコールを実行

(2) ライブラリが LOWLIB へ `fork` システムコールを発行する

(3) fork システムコールを受けとった LOWLIB は次の処理を行います

1. 中心核に対して、新しいタスクを生成するシステムコールを発行 (`cre_tsk`)。
   デフォルトでは、1 つのプロセスごとに以下のタスクを生成します (表 10.4 も参照のこと)。
   - 主タスク
   - シグナル
   - alarm 用タスク

   これら以外のタスクについては、必要なときに LOWLIB が生成する。
2. 新しく生成したタスクの仮想空間に LOWLIB (自分自身) を複製します。以後の処理は、複製した LOWLIB が行います。
3. 新しいタスクに対して `fork` システムコールを発行した Region の内容を複製する (`vdup_reg`)。
   複製は、以下の Region に対して行う。
   - コード領域

- データ領域
- スタック領域

4. 新しいタスクの実行ポイントを fork の子プロセス実行エントリポイントへセット。
   子プロセスは次に CPU の使用権が渡ってくるときに、この処理で設定したエントリから実行する。
5. POSIX/プロセスマネージャに、新しく作成したプロセスの情報を追加。

(4) POSIX/プロセスマネージャは、プロセス情報テーブルに新しく作成したプロセスを追加する

(5) 呼び出し元へ戻る

プロセスの終了処理 (obsoleted)

プロセススケジューリング (obsoleted)

プログラムの実行 (obsoleted)

シグナルハンドリング (obsoleted)

POSIX/プロセスマネージャでは、シグナルの送信/受信の処理のみを扱います。つまり、シグナルを受信した結果 core ダンプを作成するなどの処理は、各プロセスが自分自身で処理を行います。

POSIX/プロセスマネージャが受信するメッセージのうち、シグナルに関係するのは次のものです。

PROC_KILL	シグナルを送信する。
PROC_SETUP	シグナルを受信するメッセージバッファ ID を登録する。

これらのうち、PROC_SETUP についてはプロセスの初期化の時に使用するものです。

PROC_KILL シグナルを送信する

PROC_KILL は、次の構造をもったメッセージです。

```
struct proc_kill
{
  proc_t        dest_proc;    /* シグナルの送信先のプロセス ID */
  unsigned int  signo;        /* シグナル番号 */
};
```

このメッセージで明らかのように送信元のプロセスは、送り先のプロセスについては、プロセス ID のみ知っていることを前提にしています。

このメッセージを受けとると、POSIX/プロセスマネージャはプロセス情報として登録しているシグナル情報の送信用メッセージバッファへシグナル情報を送信します。

シグナル情報は、シグナル番号を表現する単なる 32 ビットの整数値です。

シグナル情報を受信するのは LOWLIB 層で動いているシグナル処理用タスクです。

インターバルタイマ (obsoleted)

POSIX には、一定時間後にシグナル (SIGALRM) を送信する alarm システムコールがあります。

alarm システムコールの使用方法は簡単です。

```
alarm（待ち時間）
```

と指定することによって、引数 **待ち時間** で指定した時間が経過するとカーネルが SIGALRM シグナルを alarm システムコールを実行したプロセスへ送ります。

alarm システムコールを実行したあとも、ユーザプロセスは他の処理を続けることができるため、一種のマルチタスク的な処理を行うことができます。

POSIX/プロセスマネージャは、alarm システムコールを実現するため、一定時間ごとに、起動するタスクをもっています。

10.2.3 メモリマネージャ(obsoleted)

POSIX 環境で使用するメモリマネージャは、次の仕事をします。

- POSIX プロセスのページフォールト発生時の処理
- 仮想空間の割り当て管理
- POSIX プロセスのもつ物理メモリのページアウト処理

POSIX プロセスの仮想空間レイアウト (obsoleted)

POSIX プロセスの仮想空間上でのレイアウトを図 10.5 に示します。

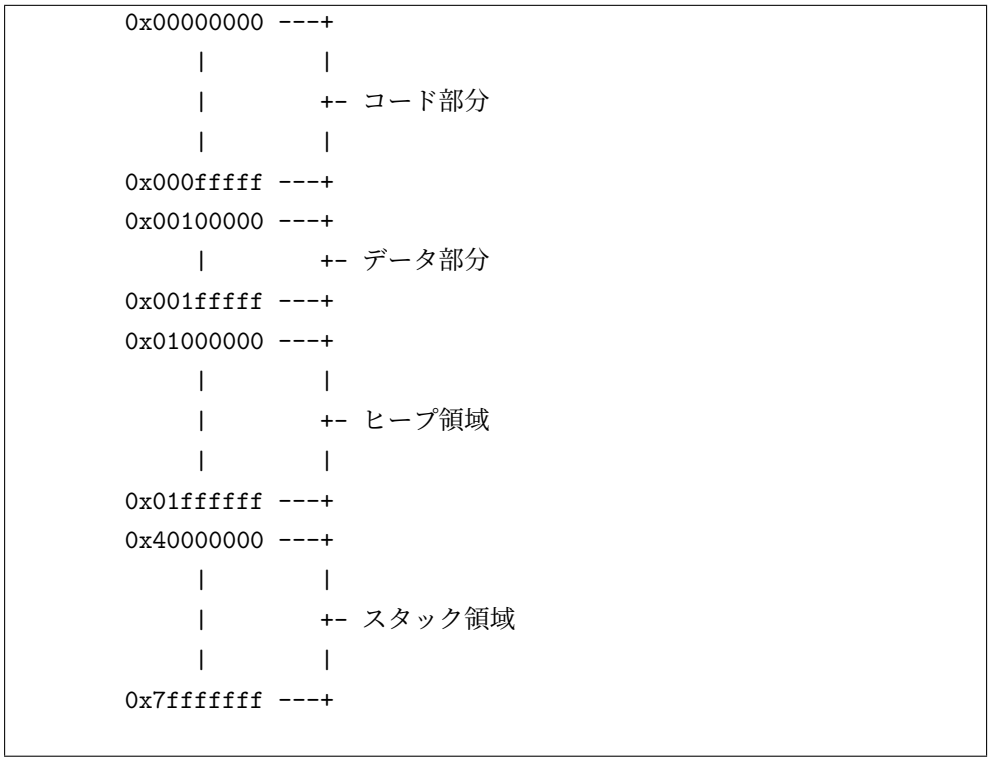

図 10.5: POSIX プロセスの仮想空間

ページフォールト (obsoleted)

ページフォールトの処理は、オンデマンドに行われます。

ユーザプロセスの処理中にページフォールトが発生した場合、その原因によって対処方法が異ります。

ページアウト処理 (obsoleted)

ページアウト処理は、すでに使っていない物理メモリをマッピングしている領域を他の役割に使用するためのものです。

ページアウト処理は、メモリマネージャの中の一タスクが行います。

表 10.3: ページフォールト処理

ページフォールトの原因	対処方法
ページアウトしている仮想ページをアクセスした	スワップ領域からページインを実行
読み込んでいないコード領域をアクセスした	実行ファイルを読み込み
スタック領域をアクセス	スタックを成長する
管理していない領域をアクセス	ページフォールトシグナルを発行

10.2.4 デバイスドライバマネージャ(obsoleted)

POSIX 環境でのデバイスマネージャは、BTRON 環境のデバイスドライバマネージャができるまでのつなぎとして動作します。

インタフェースなどは、BTRON 環境のデバイスドライバマネージャと同じですので、そちらを参照してください (Chapter 7)。

10.3 POSIX 環境での LOWLIB(obsoleted)

POSIX 環境を実現するために、POSIX 環境では独自の LOWLIB をもっています (以下、POSIX/LOWLIB と略記)。

POSIX/LOWLIB は、次の処理を行います。

ユーザプロセスの初期化を行う部分

プロセスではじめに実行する処理です。

ユーザのプログラムコードを実行するための準備を行います。ユーザプロセスが起動された状態では、スタックの内容などは設定されていません。LOWLIB のこの部分によってコマンドライン引数 (argc, argv) などの設定を行います。

システムコールの処理

ユーザプログラムが POSIX システムコールを発行したときに、システムコールの処理を実行する部分です。

システムコールの機能は、POSIX に定められた規格に準拠しています。(一部拡張/変更あり)

シグナルの処理

シグナルの受信処理を行います。

10.3.1 ユーザプロセスの初期化 (obsoleted)

ユーザプロセスの初期化は、POSIX/LOWLIB とユーザプロセスがリンクする初期化コード (pstart.o) の共同作業によって行います。

- POSIX/LOWLIB

  ユーザプロセスが動作するための環境を整えます。

  具体的には、以下の処理を実行します。

  - ユーザプロセス用の仮想空間 (コード/データ/ヒープ/スタック) を生成。
  - 生成した仮想空間のうちコード部分を読み込む (正確にはデマンドページングにより、アクセスした瞬間に実行ファイルを読み込むように設定する。
  - プロセスマネージャにプロセスを登録する。
  - ユーザスタックのボトム部に必要な引数 (argc/argv) をロードします。
  - LOWLIB 層で動くタスクを生成します。
  - ユーザプログラムのエントリ部分 (pstart.o の先頭) へジャンプします。

- `pstart.o`

  `pstart.o` は、ユーザスタックの内容を整え `main()` をコールします。

10.3.2 システムコールの処理 (obsoleted)

POSIX 環境でのシステムコール番号は、65 番を使用します。

10.3.3 シグナルの処理 (obsoleted)

POSIX/プロセスマネージャから送られたシグナルメッセージは、まず、POSIX/LOWLIB 層で動いているシグナル (受信) タスクが受けとります。

シグナルタスクは、受け取ったシグナルの種類によって次のどれかの処理を行います。

- ユーザプロセスを強制終了させる (必要ならば core ダンプを出力する)。
- シグナルを無視する。
- ユーザプログラムのシグナルハンドラを実行する。

- プロセスを一時停止させる。
- プロセスを再開させる (すでにプロセスが一時停止していた場合)。

10.4 ユーザプロセス (obsoleted)

POSIX 環境では、ユーザプロセスは複数のタスクが協調して動作しています。

ユーザプログラムでもプロセスに従属しているタスクを生成することもできます。ユーザプログラム中でタスクを生成していない状態では、次のタスクが動いています (表 10.4)。

表 10.4: 各ユーザプロセスで動いているタスク

主タスク	ユーザプログラムのコードを実行しているタスク。
シグナル受信タスク	シグナルを受信するためのタスク。
IPC 用タスク	パイプなどによって他のプロセスと通信を行うときに動くタスク。
alarm タスク	Alarm システムコール用に使用するタスク。
select 用タスク	select システムコール用のタスクです。複数のストリームからの入力を監視します。通常は動きません。

同じプロセスに属しているタスクは、仮想記憶空間を共有します。つまり、同じプロセス内のタスクの間では、メモリ内容が互いに読み書きできるようになっています。

各タスクの関係を図によって示すと 図 10.6 のようになります。

10.4.1 主タスク (obsoleted)

主タスクでは、ユーザプログラムのコード部分を実行します。

プロセスに属しているタスクの中でもメインの処理を行うタスクです。

ユーザプロセスは、主タスクを複数生成することができます [10]。

10.4.2 シグナルタスク (obsoleted)

シグナルの受信処理を行うタスクがこのシグナルタスクです。

POSIX ではシグナルとして表 10.5 に示す種類を定義しています。

[10]現状では、未サポート

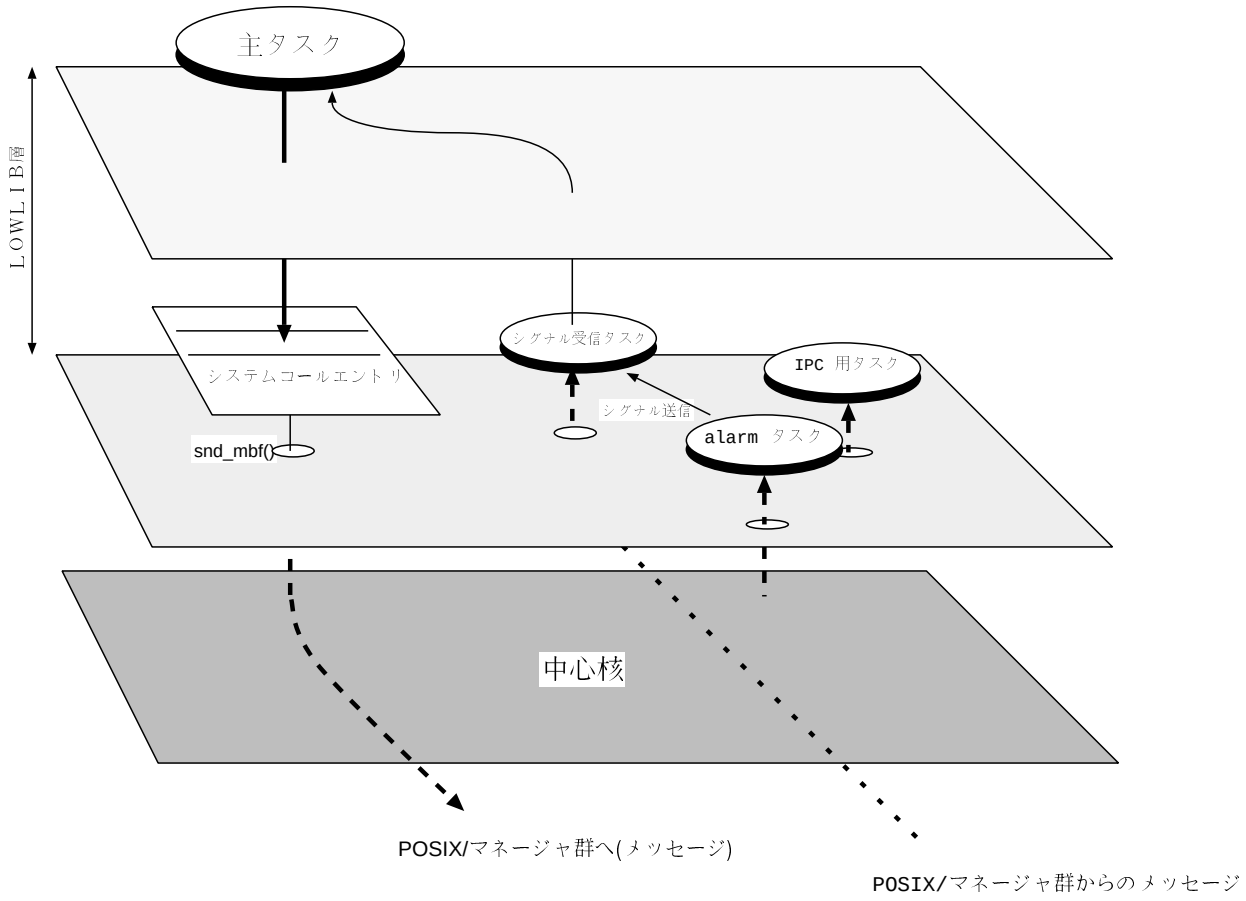

図 10.6: POSIX プロセス内で動く (ITRON) タスク

表 10.5: シグナルの種類

SIGHUP (1)	ハングアップシグナル。端末がハングアップしたときに発行するシグナル
SIGINT (2)	キーボードの割り込みキーを入力したときにプロセスに送るシグナル
SIGQUIT (3)	キーボードの割り込みキーを入力したときにプロセスに送るシグナル
SIGILL (4)	illegal instruction
SIGABRT (5)	abort() 関数によって引き起こされるシグナル
SIGFPE (6)	
SIGTTIN (7)	
SIGTTOU (8)	
SIGKILL (9)	強制終了シグナル。このシグナルはマスクすることができない。
SIGSEGV (10)	セグメンテーションフォールトが発生したときに送られるシグナル。
SIGALRM (11)	アラームシグナル。alarm システムコールによって設定した時刻に送るシグナル。
SIGSTOP (12)	
SIGUSR1 (13)	ユーザ定義シグナル
SIGUSR2 (14)	ユーザ定義シグナル
SIGTERM (15)	
SIGCHLD (16)	
SIGTSTP (17)	プロセス中断シグナル
SIGCONT (18)	プロセス再開シグナル
SIGPIPE (19)	

これらのシグナルを受けとるとプロセス (シグナルタスク) は、次のいずれかの処理を行います。

- シグナルを無視する。
- プロセスを終了する。
- プロセスを中断する。
- プロセスを再開する。
- signal システムコールによって設定してあったシグナルハンドラを実行する。

これらのうち、シグナルハンドラを実行するケース以外のケースについては、シグナルタスク内だけで処理します。

付 録A　B−Free のブート方式

A.1　ブートの概要

ブートは、いくつかの段階に分けて実行されます。

0) IPL

IPL は、マシンの ROM に元々収められているプログラムです。IPL では、立ち上げデバイスの先頭ブロックにあるコードを読み取り、処理を受けわたします。

PC9801 での IPL が終了した時点での各種レジスタの値は、次の仕様になっています。

AX,BX,CX,DX		不定
ロードアドレス	セグメント	0x1FC0
	オフセット	0x0000
ロードサイズ		1024 バイト

(1) first boot

まず、立ち上げデバイスの先頭ブロック (ブートブロック) に入っているブートがマシンの IPL によって読み込まれ/実行されます。

first boot は、コンソールに立ち上げメッセージを出力し、指定されたパーティションから second boot を読み込み実行します。このとき second boot のサイズは無条件に 64K bytes と見なします。

first boot は、8086 モードで動作し、FD などの操作は、BIOS を介して行います。

なお、first boot は、as86 (Linux にある 8086/8088 用アセンブラ) でかかれています。

(2) second boot second boot は、マシン (CPU) のモードを 32 ビットに変更し、カーネルをロードする。カーネルは、カーネル自身と最低限必

要なマネージャがパックになった形式で second boot の直後に入っています。

second boot は、ファイルシステムの扱いかたを知っていますが、特に指定しない限りファイルシステムは見に行きません。

カーネルに処理を渡すときに、カーネルの動作環境も整える。具体的には、仮想メモリ環境のセットアップ、CPU モードの変更、GDT/IDT の初期化などを行います (つまりカーネルに制御が渡った時には、32 ビットモードかつ仮想ページ機能つきで CPU が動いています)。

second boot は、C によって記述されています (一部はアセンブラが入っている)。

second boot が実行されたときのコンソールのメッセージを次に示します。(';' の後は註釈です)

```
        Loading second boot...done      ; first boot が出力
します。
        Second BOOT for BTRON/386
        Version 1.0.0                   ; バージョン番号は異
なっている
                                        ; 場合があります。
        Waiting 10 second.              ; 10秒間待ちます。こ
のときに何
                                        ; かキーを押すと対話
モードに入
        Loading kernel...done           ; ります。
                                        ; 後はカーネルに制御
が渡ります
```

second boot の対話モードでは、次のコマンドが使用できます。

ls [parttion] [path]	path で指定した実身のリンクを表示します
boot [partition] [path]	path で指定したカーネルをロードします
reset	CPU をリセットします

partition は、次のように指定します。

FD:0	ドライブ A の FD
SCSI0:0	SCSI ID 0 のパーティション 0

path は、次のように指定します。

/foo/bar	実身 foo の中の 実身 bar

A.2 ブートブロックの構造

パーティションの先頭にあるブートブロックは、次の構造をしています。

1) firstboot (1024 bytes)	0 ブロック - 1 ブロック
2) second boot (64 K bytes)	2 ブロック - 129 ブロック
3) kernel (MAX 1M bytes)	130 - 2177 ブロック

!! **重要** !!
現在の BTRON/386 のブートは 1 セクタ 512 バイトにしか対応していません。
もし、ブートを作り直した時にうまくローディングできなかった場合には、フロッピィのフォーマットが 1 セクタ 512 バイトとなっているかを確認してみてください。

A.3 ファーストブート終了時のメモリマップ

ファーストブートがセカンドブートをロードした後のメモリマップは次のようになっています。

```
  Addr                内容
0x00000000      BIOS が使用する領域              4K
0x00001000 --+  セカンドブートロード領域        60K
    |        |  0x00001000 16bit code.
    |        |  0x00002000 32bit code.  start address: 0x9000
    |        |
0x0000FFFF --+
0x00010000      GDT テーブル                     2K
    |
0x00011FFF
0x00012000      IDT テーブル                     2K
    |
0x00013FFF
```

A.4　セカンドブート終了時のメモリマップ

　セカンドブートがカーネルをロードした後のメモリマップは、次のようになっています。

(1) 仮想アドレス

```
        0x00000000
             |
             |
        0x7FFFFFFF

        0x80000000 --+  セカンドブートロード領域の残骸
             |       |  ページテーブルに使用する。
             |       |  (0x80000000 - 0x800003FFF は、セカンドブートが
             |       |  初期化する。
        0x8000FFFF --+
        0x80010000      GDT テーブル                              2K
             |
        0x80011FFF
        0x80012000      IDT テーブル                          2K
             |
        0x80013FFF
        0x80020000 --+  カーネルが使用する領域            1G (Max)
             |       |  ここにカーネルのイメージが
             |       |  ロードされる。
        0xBFFFFFFF --+
```

(2) 物理アドレス

```
        0x00000000 --+  セカンドブートロード領域の残骸
             |       |  ページテーブルに使用する。
             |       |  (0x80000000 - 0x800003FFF は、セカンドブートが
             |       |  初期化する。
        0x0000FFFF --+
        0x00010000      GDT テーブル                          2K
             |
        0x00011FFF
        0x00012000      IDT テーブル                          2K
             |
        0x00013FFF
        0x00020000 --+  カーネルが使用する領域            1G (Max)
             |       |  ここにカーネルのイメージが
             |       |  ロードされる。
        0x3FFFFFFF --+
```

付 録B　libkernel.a

— つまり、シュヴァルツシルト・ホールははだかの炎のようなもの。穴居人の道具ですな。コントロール可能なカーネルは、もっと洗練されたものです。
Charles Sheffield 「マッカンドルー航宙記」

B.1　libkernel.a の役割 (obsoleted)

libkernel.a は、周辺核とデバイスドライバ そして LOWLIB で使用することができるライブラリです。

libkernel.a は、以下の機能を提供します。

- 中心核の機能を使用するためのシステムコール関数
- デバイスに対して入出力を行うための関数
- 割り込み操作関数
- DMA 操作関数
- 文字列操作などの各種ライブラリ関数

このライブラリを使用することによって、周辺核などの作成を助けることができます。

B.2　使用方法 (obsoleted)

libkernel.a は、ライブラリとしてリンクするだけで使用できます。

libkernel.a を使用する方法は、直接 libkernel.a をリンクする方法とライブラリパスを指定する方法の 2 通りがあります。

B.2.1　直接 libkernel.a をリンクする (obsoleted)

直接 libkernel.a を指定してリンクする方法です。

リンクコマンド ld を使う場合にはこの方法を使用します。

表 B.1: libkernel.a の関数一覧

中心核システムコール	ITRON 構築マニュアルを参照のこと	
I/O 関係	inb	指定したアドレスから 1 バイト幅で読み込む
	inh	指定したアドレスから 2 バイト幅で読み込む
	inw	指定したアドレスから 4 バイト幅で読み込む
	outb	指定したアドレスに 1 バイト幅で書き込む
	outh	指定したアドレスから 2 バイト幅で書き込む
	outw	指定したアドレスから 4 バイト幅で書き込む
DMA 関係	dma_setup	DMA の転送設定を行う
文字列操作	strlen	文字列の長さを測定 (ASCII 文字を前提)
	strcmp	2 つの文字列が一致しているかを調べる
	strcat	2 つの文字列を連結 (concat) する

```
ld -o foo bar <libkernel.a>
```

B.2.2 ライブラリパスを指定する方法 (obsoleted)

cc のコマンドラインオプションのひとつ -L を指定する方法です。
具体的には、次のように指定します。

```
cc -o foo bar -L<libkernel.a の入っているディレクトリ> -lkernel
```

B.3 libkernel.a の関数 (obsoleted)

表 B.1 に libkernel.a のもつライブラリ関数の一覧を示します。

付 録C　B-Free OS のインストール方法

付 録D　B−Free ソースディレクトリ一覧

1995 年 3 月 現在の B−Free OS のディレクトリは次のとおりです。

bin(obsoleted)

make に必要なツールが入ったディレクトリです。今のところ入っているのは、文字コード変換用の kp コマンドだけです。

doc(obsoleted)

ドキュメント類の入ったディレクトリです。更に次のようなサブディレクトリに分かれています。

introduction この文書が入ったディレクトリです。

manual マニュアル類が入ったディレクトリです。

meeting B−Free プロジェクトチームのミーティングの記録です。

note 細かなメモ文書が入っているディレクトリです。

src(obsoleted)

ソースの入っているディレクトリです。

付 録E　API 一覧

E.1　ITRON (中心核)

E.1.1　リージョン操作システムコール

付 録F　参考文献

TRON プロジェクト全般について

- TRON を創る　共立出版
- TRON 概論　　共立出版

ITRON について

- μ ITRON3.0 標準ハンドブック パーソナルメディア社

BTRON について

- BTRON1 プログラミング標準ハンドブック パーソナルメディア社
- TRONWARE Vol.28 BTRON(1B/V1) の体験フロッピー付 パーソナルメディア社

POSIX について

- POSIX — Part 1: System Program Interface (API) [C Language] *IEEE*