

**L4minix-e 説明書 (仮版)**

Alpha, Beta and Gamam

国立情報学研究所 *H<sub>2</sub>O* グループ

平成 15 年 8 月 15 日

# 第1章 L4minix-e の概要

## 1.1 目的

本 OS 研究は、以下の目的から発足した。L4minix-e は、この目的に向けた中間成果である。

制御システム・組込みシステム等のためのビルディングブロック型 OS 制御システムなどの専用システムの OS には、適用分野毎に異なる機能を容易に開発・追加できること、ハードウェア制御を容易に行えること、障害(プログラムバグも含め)に対し頑強であること、無駄な機能がなくコンパクトであること等が望まれる。このような要求に応えるために、応用プログラムをインストールするかのように OS 機能をビルディングブロックとして追加できる OS を実現したい。

マイクロカーネルとマルチサーバー構成によるビルディングブロック化 ビルディングブロック化を達成するために、マイクロカーネルを採用し、プロセス管理、ファイルサービス、ネットワークサービスなどの箇々の OS サービスを、ユーザモードで走るタスクとして実現する。箇々のサーバーは個別の論理空間を持ちユーザモードで走るため、プログラム暴走などからも強固にガードされる。マイクロカーネル方式は、メッセージ通信のオーバーヘッドから性能が懸念されているが、適切なマイクロカーネルと方式構成により問題をクリア出来ることを実証したい。

我々は当初は理想的なマイクロカーネルをスクラッチから設計開発することを考えたが、検討を進めるうちに L4 マイクロカーネルをベースにした方が効果的と判断した。L4 マイクロカーネルは、ドイツ Karlsruhe 大学の故 Liedtke 教授が研究開発したマイクロカーネルであり、機能・性能ともに非常に優れる。(Mach などの第一世代マイクロカーネルに対し、L4 は第二世代マイクロカーネルと呼ぶに値する。)

ユーザレベルタスクによるドライバー技術の確立 制御システムや組込システムでは、一般にハードウェア制御が要求される。従来は、ハードウェア制御プログラムは、カーネルモードで実行する必要があり、これがドライバプログラムの開発とデバッグを困難化していた。本 OS では、ハードウェア制御もユーザレベルのプロセス内で実現できるようにする。

コンポーネントによるシステムソフトウェアの構築 オブジェクト指向の成功によりコンポーネント化は GUI 部品などでは効果を上げているが、複雑な大規模システムのコンポーネント化は今後の課題である。OS を題材にコンポーネント化を追求する。なお、ここでいうコンポーネントは再利用部品というよりも、機能拡張・変更・差し替えなどが容易なビルディングモジュールという意味で使っている。

OS の学習用素材の提供 Minix という学習用 OS があったから Linux は誕生した (Linus 氏と Tanenbaum 教授の monolithic OS 対 Micro-kernel OS 論争は面白いので一読を)。簡単に扱える マイクロカーネル + マルチサーバー型の学習用 OS を提供したい。また、シンプルで融通性に富む L4 マイクロカーネルの普及を促進させたい。

L4minix-e は、以下のような人に使っていただきたい。

- 制御システムや組み込みシステムのためのしっかりした OS を求めている人。
- 最新のマイクロカーネル技術を知りたい人。また、評判の L4 マイクロカーネルを使ってみたい人。
- モノリシック OS のようなガチガチの OS ではなく、スマートに要求機能を追加して行ける OS を望む人。

- 教育用・学習用として使え、容易に手直しできる OS を求めている人。

## 1.2 L4minix-e のシステム構成

L4minix-e の構成を図 1.1 に示す。

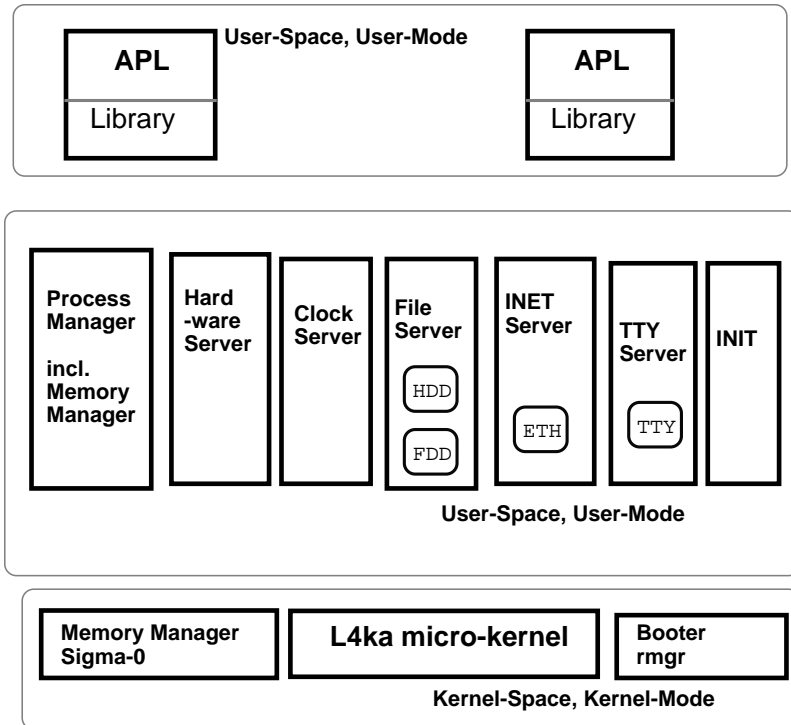


図 1.1: L4minix-e の構成図

**L4ka** マイクロカーネル ドイツ Karlsruhe 大学の故 J. Liedtke 教授の下で研究開発された魅力的な第二世代マイクロカーネルである。L4ka の詳細は、<http://l4ka.org/> を参照されたい。

**sigma0** メモリ管理を階層的に構成できるのが L4 の一つの特徴である。Sigma0 は最下層のメモリマネージャーであり、L4minix の『プロセスマネージャー』は、Sigma0 から利用可能な全メモリページを引き取って、独自にメモリ管理している。

**rmgr** L4ka マイクロカーネルをロードして立ち上げるためのブートプログラム。(名称は、Resource Manager からきているが、本システムではブート機能しか使っていない)

**OS タスク** OS サービスは、プロセスマネージャー、ハードウェアマネージャー、クロックマネージャー、ファイルサーバー、INET サーバーなどのユーザモードで動作するプロセス (= OS タスク) によって提供される。

**APLs** ユーザの応用プログラム

このような OS の作り方をマルチサーバー型 OS と呼び、以下のような特徴がある。

- 本システムでは、プロセス管理、ファイルサービス、INET サービスなどいわゆる OS サービスは、ユーザモードで走る個別のプロセスとして実現している (マルチサーバー方式)

- OS サービスを行うプロセスも、応用プログラムを実行するプロセスも仕組みに差はないが、本資料では前者を (OS) タスク、後者を (ユーザ) プロセスと呼ぶ事もある。
- 各 OS タスクはドライバプログラムも含む。例えばファイルサーバーは IDE-HDD ドライバ、Floppy ドライバ等を、INET サーバーは Ether ドライバなどを含んでいる。
- タスク (= プロセス) は、メッセージのみでやり取りを行う。例えば `read()` システムコールは、ライブラリの中でメッセージに変換され、ファイルサーバーに送られる。
- UNIX 同様、INIT や SHELL も普通のプロセスとして走っている。

### 1.3 本 OS 研究の本来の目的

我々の OS 研究にとって L4minix-e は中間の副産物であって、本来の研究目標は未だ先にある。

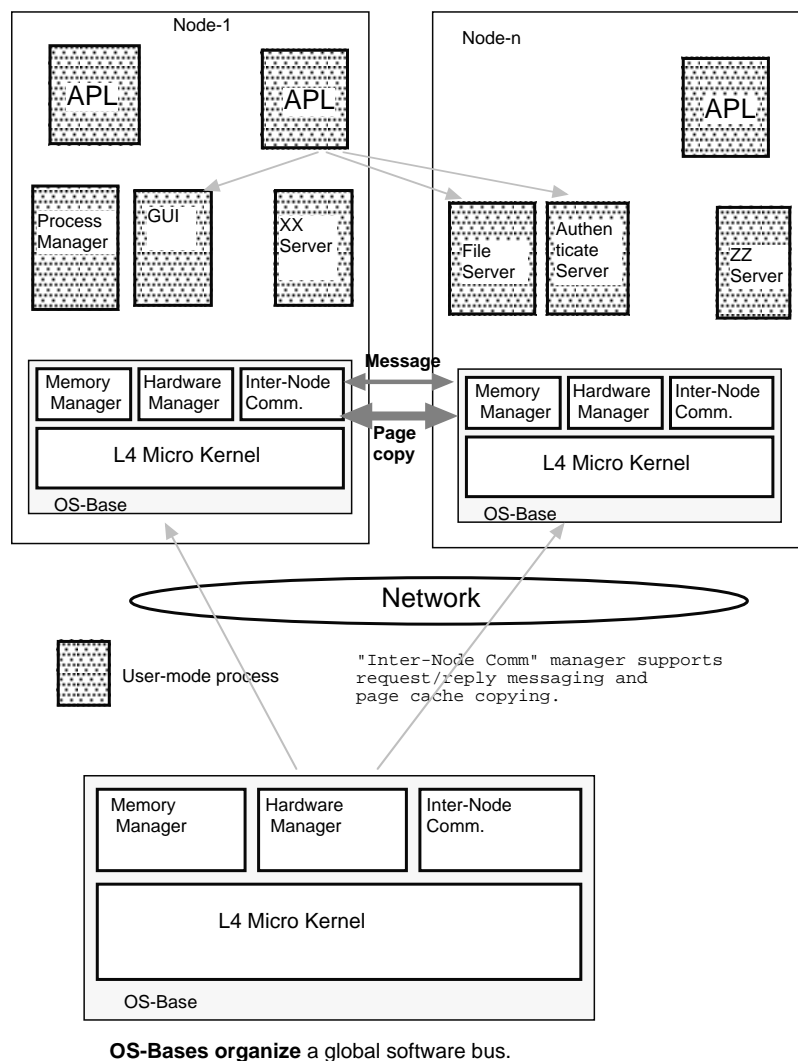


図 1.2: 本 OS 研究の目標

目標 OS のイメージを図 1.2 に示す。この OS では、以下を実現する。

**OS-Base** 各分散ノードのソフトウェア基盤であり、L4 マイクロカーネル、メモリーマネジャー、ハードウェアマネジャー、ノード間通信からなる。メモリーマネジャーは、そのノードのページフレームを上位プロセスに提供する。ハードウェアマネジャーはハードウェアを安全に共用するための機能を提供する。ノード間通信は、要求メッセージ・返答メッセージのノード間転送、ならびにページキャッシュのノード間コピーを行う。OS-Base のソースコードは、50 ~ 80 K 行程度を想定している。

**OS サービスタスク** ファイルサービス、インターネットサービス、認証サービス、GUI サービスなどのいわゆる OS サービスは、個別のユーザーモードのプロセスとして実現する (L4minix-e で実現済み)。これを OS サービスタスクと呼ぶ。各 OS タスクは、個別の論理空間をもち、ユーザモードで実行されるので、強固なプログラム保護が効いている。デバイスドライバは、OS サービスタスクに含まれ、ユーザモードで実行される。OS サービスタスクは、必要に応じて自在にプラグイン可能にする。プロセッサに依存しない IO 空間のアクセス保護も実現したい。

**OS-base の Inter-Node Comm. 機構** 本機構は要求メッセージ・返答メッセージのノード間転送、ならびにページキャッシュのノード間コピーのみを行う。一般のインターネットプロトコル等は OS タスク、もしくはユーザレベルライブラリーで実現する。

このための仕組みとしては、パケットフィルターを用いて Inter-Node Comm. のパケットのみを区分けしてこの階層で処理し、それ以外のパケットは上位階層に転送する。

OS-base の Inter-Node Comm. プロトコルとしては、TCP のような汎用重量プロトコルは不要であり、より軽量なプロトコルを使う。例えば、NII 松本助教授の SSS-core の手法、Plan 9 OS の IL プロトコルなどが適切であろう。

なお、速度が遅くてもよいから Firewall を通したいという場合は、HTTP 等に重畳したトンネリングを考える。

**ノード間でのページキャッシュ(?) コピー** 効率化の機構としてはページキャッシュを広く活用する。たとえば他ノードのファイルサーバーにアクセスすると、そのファイルサーバーのページキャッシュの内容が要求プロセスのライブラリーのページにコピーされる。ファイルセマンティクスを保証すれば良いので、同期の粒度は open, flush, sync, close のレベルを考えている。ページキャッシュの Read-only マッピングの実装は簡単である。ページキャッシュの Read-write マッピングの場合は、実用上十分なセマンティクスとして、簡潔化を図る。ノード間のページコピー機構が、重要な研究課題である。

**分散透過** OS-Base がノード間の連携を図ってくれるので、ユーザの応用プログラムは勿論、OS サービスタスクも、ノード境界を意識せずに連携できる。例えば、ローカルファイルサーバーとか NFS ファイルサーバーといった差は無く、ノード内・別ノードのファイルサーバーに自在にアクセスできる。

**ビルディングブロック** 各ノードは必要な OS サービスタスクのみを積み上げよう。例えばファイルサーバー専用機はファイルサーバーと認証サーバーのみを、ユーザ端末はプロセスマネジャーと GUI サーバーと言った具合に。

**IPC の scatter/gather の拡張** 画像処理などでは、巨大なデータブロックの中からある規則に元づいて部分データの集合を送り合いたいなどということがあがる。これは、L4 マイクロカーネルの scatter/gather をテーブル駆動からプログラム駆動とすることで効率的に実現できる。問題は、このようなアルゴリズムを安全にカーネルに登録する手法である。(Kernel loadable module は、この一例。)

## 1.4 Minix ソースプログラムとの関係 (参考)

OS タスクとライブラリーは、Minix のソースプログラムを移植して実装した。図 1.3 に L4minix-e と Minix の機能配置の関連を示す。

## 1.5 L4minix-s との関係 (参考)

L4minix-s (Straight) は、我々のもう一つの L4minix の実装である。L4minix-S では、Minix のカーネルを L4ka の上で走れるように修正し、その上で Minix の MM サーバー、FS サーバー、INET サーバーを走らせている。L4minix-s では、各タスクは Minix のメモリ管理が適用されている。つまりページングによる論理アドレスではなく、物理アドレスを複数領域に分けて箇々の領域をタスクの論理空間としており、各タスクでは「論理アドレス + 開始物理アドレス = 物理アドレス」となっている。

図 1.4 に L4minix-s の構成を示す。

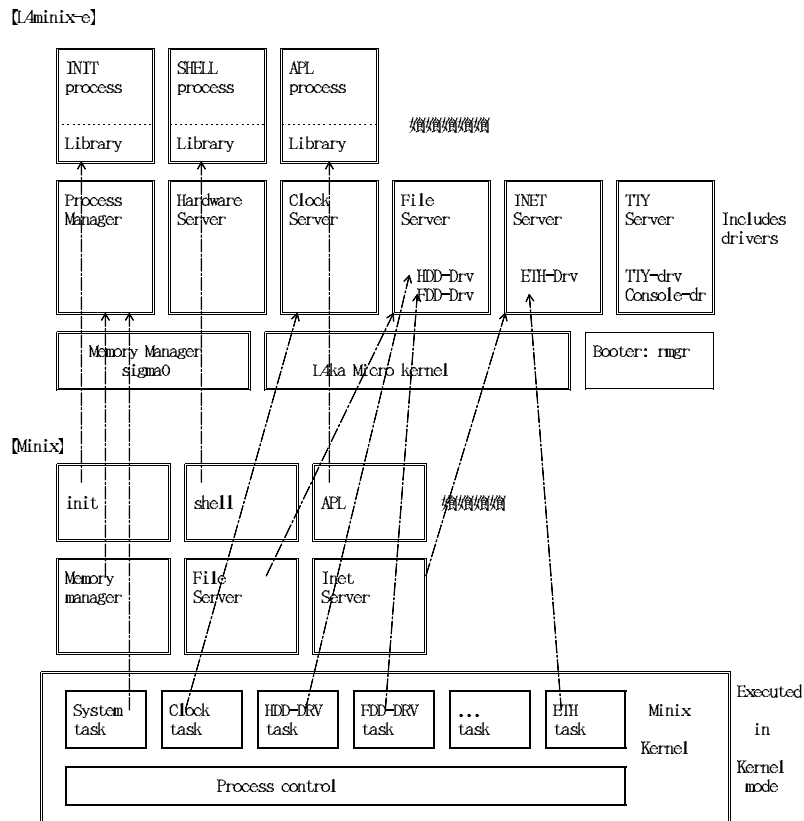


図 1.3: L4minix-e と Minix の関係図

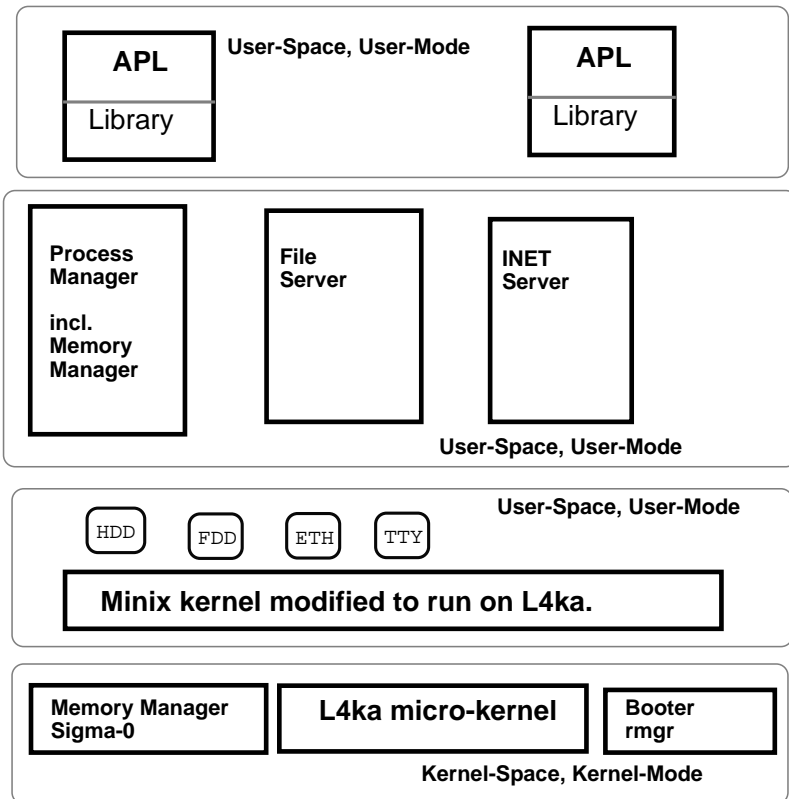


図 1.4: L4minix-s の構成図



## 第2章 L4minix-eのインストールと起動

### 2.1 必要な計算機環境

L4minix を走らせるには、

- ハードディスクに空 primary Partition のある PC-AT 機、あるいは
- VMware (Linux や Windows 上に仮想 PC-AT マシンを作るソフトウェア)

を用意する。デバッグなど行うには、VMware の方が便利である。

プログラム開発には Linux あるいは FreeDSB 環境と、GNU C コンパイラ GCC-V2.95.x が必要である。(GCC-V3.xx では、L4ka のヘッダーファイルがコンパイル出来ない。)

### 2.2 L4minix-e プログラムのダウンロード

L4minix-e のホームページ (URL: <http://research.nii.ac.jp/H2O/L4minix-e/L4minix-e.html> ここで、H2O は H Two Ou = 水) より、必要なプログラムやソースコードをダウンロードできる。なお、以下のファイル名に付いている yy-mm-dd は 2 桁表示の year-month-day を意味する。

`l4minix-yy-mm-dd.tar.gz` L4minix-e のソースコードの圧縮 tar ファイル。

`l4minix-boot-yy-mm-dd.image` L4minix-e のブートフロッピーのイメージ。

`l4mx-bin.tar` L4minix-e の /bin コマンド が入った tar ファイル。

`l4mx-usr.tar` L4minix-e の /usr ディレクトリーが入った tar ファイル。

`l4mx-www.tar` L4minix-e の /www ディレクトリー (html ファイル例) が入った tar ファイル。

`current-yy-mm-dd.tar.gz` “L4ka マイクロカーネル”、“Minix” 及び “L4minix-e” の全ソースファイルを含む圧縮 tar ファイル。このうち、L4minix-e のソースファイルは最新版ではないので、前記の `l4minix-yy-mm-dd.tar.gz` で上書きして使ってください。

### 2.3 L4minix-boot フロッピーの内容

L4minix のブートフロッピー『`l4minix-boot-yy-mm-dd.image`』には、下図に示すように GNU のブートローダー Grub、L4ka マイクロカーネル及び L4minix-e の OS タスク類が乗っている。フロッピーのファイル形式は、Linux のファイルシステム形式である ext2 である (Grub は、ext2 他のファイルシステムを理解できるローダーである)。

```

---- +---boot/ -----grub/ -----+---stage1
      |                                     |--stage2

```

```

|                                     |--menu.lst
|--x86-kernel : L4ka マイクロカーネル
|--rmgr       : L4 の rmgr
|--sigma0    : L4 の Pager
|--simple/    ----+----- この directory に L4minix のプログラムが乗っている。
|

```

boot/grub/menu.list (Menu list) は、OS の立ち上げ法の指示書であり、以下が定義されている。  
ブートローダー Grub は、最初にこのリストを表示し、ユーザが選択したものを立ちあげる。

**L4MINIX-E** L4minix-e を立ちあげる。

**L4MINIX-E INSTALL (MKFS and MKDIR)** L4minix-e をインストールする。ディスクパーティション (VMware の場合は 仮想ディスクのパーティション) 上に、L4minix-e が使うファイルシステム (Minix ファイルシステム第2版) を構成し、必要ないくつかのディレクトリーとファイルを作る。(但し OS 立ち上げは、ブートフロッピーから行う必要がある。)

**L4MINIX-E (qsh)** 基本コマンドを組み込んだ簡易な疑似シェル (quasi shell) を使った OS を立ち上げる。

## 2.4 ソースファイルのディレクトリ構成

ソースファイル “l4minix-yy-mm-dd.tar.gz” のディレクトリー構造を以下に示す。

```

----l4minix/----+--- bin/ -----: Binary code (l4-ka kernel, rmgr, sigma0)
|
|-- boot/ ----- menu.lst for GRUB
|-- include/ ----include files
|-- lib/ -----: libraries
|----l4mx/----+--- bin/      主要コマンドのバイナリーが置いて有る
|
|      |--- etc/      etc ファイルの実例
|
|      |--- home/
|
|      |--- mnt/
|
|      |--- usr/      usr-directory の実例
|
|      |--- www/      html ファイルの例
|
|-- src/-----+---commands/ -----ported Minix commands
|
|      |--l4-support/ ----sigma0, rmgr
|      |--lib/ ----- library sources
|      |--task/ -----+--- clock : Clock server
|
|          |-- eth : Ether driver
|
|          |-- fs : File Server
|
|          |-- hw : Interrupt support
|
|          |-- ide : IDE driver
|
|          |-- inet : INET server
|
|          |-- mm : Memory Manager
|
|          |-- qsh : quasi shell

```

```

|
|--test/ -----: Minix test programs
|--work/ -----+-----apl/-- : test programs
|

```

## 2.5 L4minix-boot フロッピーの menu.lst の内容

menu.lst は GRUB ローターに OS の立ちあげ方を教えるものであり、例えば menu.lst の “L4MINIX-E” の内容は以下のようになっている。

【例】 menu.lst の L4MINIX-E の内容

```

title = L4MINIX-E
kernel = (fd0)/rmgr.gz -sigma0 -roottask
modaddr= 0x01000000
module = (fd0)/x86-kernel.gz
module = (fd0)/sigma0.gz
module = (fd0)/simple/mm.gz
module = (fd0)/simple/hw.gz
module = (fd0)/simple/clock.gz      vmware
module = (fd0)/simple/fs.gz partition=1 ## This means the partition 1.
module = (fd0)/simple/inet.gz
module = (fd0)/simple/tty.gz      # ttytype=jpn
module = (fd0)/simple/init.gz

```

GRUB ローターは、module= で指定された各プログラムを計算機のメモリ上に (modaddr 番地以降) コピーし、kernel= として指定された rmgr を起動する。rmgr は、マイクロカーネル (x86-kernel)、メモリマネージャー (sigma0) を起動し、次いでプロセスマネージャー (mm) を起動する。mm は、自分以外の OS タスク (hw, clock, fs, inet, tty, init) を立ち上げる。“gz” は圧縮ファイルを意味する。

module 指定に付けられたパラメータは、各タスクの main(int argc, char \*\*argv) ルーチンの argv[0] に引き継がれる。clock.gz に付けた vmware は、クロック速度を VMware 用に調整することを意味する。使用キーボードが日本語配列の場合には、tty.gz に ttytype=jpn を付ける。

## 2.6 L4minix-e のインストール法

1. Linux-PC の上に “VMware” をインストールする。もしくは、PC-AT 機を用意する。
2. <http://research.nii.ac.jp/H20/L4minix-e/L4minix-e.html> からブートフロッピーイメージ “l4minx-boot-yy-mm-dd” をダウンロードし、Linux の dd コマンドを用いてブートフロッピーを作る。

```
# dd if=l4minx-boot-yy-mm-dd.image of=/dev/fd0 bs=4k
```

3. 同上ホームページ から L4minix コマンド TAR ファイル “l4mx-bin.tar” をダウンロードし、フロッピーに dd する。この TAR ファイルには /bin ディレクトリーの内容が入っている。

```
# dd if=l4mx-bin.tar of=/dev/fd0 bs=4k
```

4. 同上ホームページ から L4minix usr-AR ファイル “l4mx-usr.tar” をダウンロードし、フロッピーに dd する。この TAR ファイルには /usr ディレクトリーの内容が入っている。

```
# dd if=l4mx-bin.tar of=/dev/fd0 bs=4k
```

5. 同上ホームページ から L4minix www-TAR ファイル "l4mx-www.tar" をダウンロードし、フロッピーに dd する。この TAR ファイルには、WWW 用の HTML ファイルの例題 (/www ディレクトリーの内容) が入っている。HTML ファイルを作る手間を省いて、HTTPD サーバーを動かして見たい場合には、これを利用してみてください。

```
# dd if=l4mx-www.tar of=/dev/fd0 bs=4k
```

6. VMware の仮想ディスク、あるいは PC-AT の HDD にパーティションを設定する。パーティション設定は、例えば Minix インストールフロッピーがあればその第一枚目を実行すれば行える。Linux のインストール手順のパーティション設定過程でも作れる。市販ツール Partition Magic 等を使えばより簡単に設定できる。L4minix-e はデフォルトでは Partition 1 を使うが、他の partition を使いたい場合は、l4minix-boot フロッピー (Linux ファイルシステム ext2 なので、Linux で普通に修正できる) の "boot/grub/menu.lst" を修正すること (但し Partition-1 以外は、動作未確認)。

【例】 menu.lst の内容

```
module = (fd0)/simple/fs.gz partition=1
```

7. 先に作った l4minix-boot floppy を用いて、VMware あるいは PC-AT を立ちあげる。

```
# vmware &
```

8. VMware は "menu list" を表示する。

9. 最初のランでは、"L4MINIX-E INSTALL" メニューを選択する。これにより、HDD 上に以下のディレクトリとファイルが作られる。

```
-----+--bin/-----  commands (init, getty, login, ls, cp, pwd, , , ,)
|
|--dev/-----  console, fd0, fd1, hd1, hd2, hd3, hd4, ip, mem, tty
|                (Special nodes are made by "devnode" command)
|
|--etc/-----  fstab, group, motd, passswd, profile, protocols, rc,
|                services, shadow, termcap, ttytab, (copyed from Minix /etc/*)
|                hostname.file, utmp
|
|--mnt/-----  (mount point)
|
|--usr/-----+-----adm/-----  lastlog, wtmp
|                |
|                |-----bin/-----  commands ...
```

10. L4minix には Break Point を設定してあるので、何カ所かで停まる。そしたら 'g' (意味は 'go') と打鍵すると、次に進む。

11. 最初のランでは HDD 上に Minix ファイルシステム (V2) を作るために、次のメッセージが表示されたら 'y' と打鍵する。

```
If you want REMAKE a FILE-SYSTEM, hit 'y', else hit 'g' or RTN.
```

12. Break point に来たら 'g' と打鍵して進む。未完成プログラムなのでエラーメッセージが出るが、無視してすすむ。まだバグを殺してないので、第1回目は必ずエラーメッセージがでる(要修正!!)。そこで、もう一度最初からやり直す。

13. そのうちに、次のメッセージが表示される。

```
--- Install commands ? Then insert 'l4mx-bin.tar' floppy and hit 'g' ---
```

そしたら、"l4minix-boot" floppy を抜いて、"l4mx-bin.tar" floppy を挿入し、'g' と打鍵する。これにより、L4minix-e の基本コマンドが /bin/\* ディレクトリにコピーされる。

14. Break point に来たら 'g' と打鍵して進む。未完成プログラムなのでエラーメッセージが出るが、無視してすすむ。しばらく進むと、以下のプロンプトが表示される。

```
L4MX[/]: #
```

15. これで L4minix-e は準備完了となる。L4minix を停止するには、VMware の "Power off" ボタンをクリック

クする。

- これで L4minix-e はインストールされたので、次回からはブートフロッピーを使用して Power-on し、メニューリストから “L4MINIX-E” を選択する。
- L4minix-e が立ち上がると、シェル (コマンドインタプリター) “sh” (このシェルプログラムのソースは、l4minix/src/commands/sh/\*) がスタートする。

```
L4MX[/]# init
L4-Minix Release 0.1 Version 0
hostname login: root
.....
```

ホスト名はファイル “/etc/hostname.file” で定義する。パスワードファイル “/etc/passwd” には、“root” と “ast” (ast はタネンバウム先生) が用意されている。パスワードはまだ使っていない。

- l4mx-usr.tar ファイルをマウント、TAR 展開, アンマウントする。( /etc/mntab ファイルを作っていないと、このファイルをオープン出来ないと言うメッセージが出るが、無視してよい。)

```
L4MX[/]# cd /
L4MX[/]# mount /dev/fd0 /mnt
L4MX[/]# tar xvf /mnt/l4mx-usr.tar
..... TAR ファイルが展開される .....
L4MX[/]# umount /dev/fd0
.....
```

- 必要なら、同様にして l4mx-www.tar ファイルをマウント、TAR 展開, アンマウントする。なお、この /www/etc ディレクトリーには 実際の /etc ファイルのコピーを置いて有るので、httpd.conf と httpd.mtype を /etc にコピーする。

```
L4MX[/]# cd /www/etc
L4MX[/]# cp httpd.conf /etc
L4MX[/]# cp httpd.mtype /etc
```

## 2.7 コマンド

上記インストール手順で、l4mx-bin.tar ファイルがインストールされていれば、UNIX の代表的なコマンド ( sh, ls, cd, mv, sync, mkdir, rm, , , , , ) は /bin/ ディレクトリーにコピーされ、利用可能になっている。(フロッピーの容量制限から、一部コマンドは含まれていない。)

エディターとしては elle が用意されている。/bin/elle は、Emacs 流のコマンドを持つ簡易エディターである。主要コマンドは、Emacs と同じである (CTL-n, CTL-p, CTL-f, CTL-b, CTL-x CTL-s, CTL-x CTL-c)。

ブートフロッピーメニューの『L4MINIX-E (qsh)』から立ち上げると、基本的なコマンドを組み込んだ 疑似シェル (qsh: quasi shell) が /bin/sh の代わりに立ち上がる。qsh で使えるコマンドは、L4minix-e のホームページ (<http://research.nii.ac.jp/H20/L4minix-e/L4minix-e.html>) を参照のこと。qsh のコマンドは、l4mx-bin.tar をインストールせずに使う事が出来る。

## 2.8 フロッピーを用いて Linux から L4minix-e にファイルをコピーする方法

Linux 上のファイルを L4minix-e にコピーする場合を例に説明する。

## 【Linux 上で】

1. フロッピーに Minix file system (Version 2) を編成する (L4minix のファイル形式は Minix Filesystem V2 があるので)。“-n 14” はファイル名が 14 文字、“-v” は Minix V2 ファイルシステムを意味する。

```
# /sbin/mkfs.minix -n 14 -v /dev/fd0
```

2. フロッピーをマウントする。

```
# mount /dev/fd0 /mnt/floppy
```

3. 目的ファイルをコピーする。

```
# cp myprog /mnt/floppy
```

4. アンマウントする。

```
# umount /mnt/floppy
```

## 【L4minix-e 上で】

1. マウントディレクトリーを作る

```
L4MX[/] cd /
```

```
L4MX[/] mkdir mnt
```

2. フロッピーをマウントする。(VMware 上では、少々時間がかかる)

```
L4MX[/] mount /dev/fd0 /mnt
```

3. ここで、フロッピーが使えるようになる。

```
L4MX[/] cd mnt
```

```
L4MX[/mnt] dir
```

```
L4MX[/mnt] copy myprog /usr
```

4. フロッピーを unmount する。

```
L4MX[/mnt] cd
```

```
L4MX[/] umount /dev/fd0
```

## 2.9 ネットワーク機能の使い方

L4minix-e のネットワーク機能の使い方の例として、VMware 上で走る L4minix-e による WEB サービスを説明する。VMware の中では L4minix-e の上で HTTPD サーバーを走らせ、Linux 上の WEB ブラウザからアクセスする (Cf. 図 2.1)。

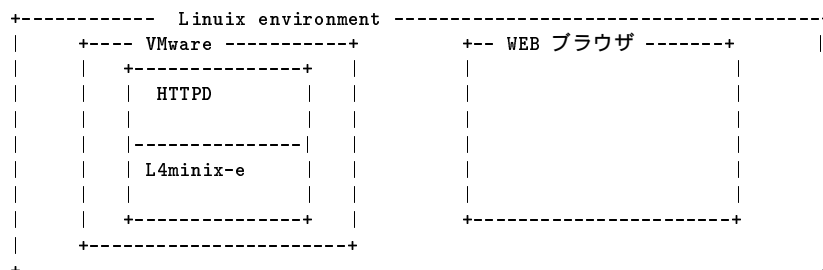


図 2.1: Linux 開発・実験環境

1. L4minix-e 上に HTTPD 機能を用意する。これは、TAR ファイル `l4mx-usr.tar`、`l4mx-www.tar` をインストールすることで行えるが、以下の手順でも行える。
  - (a) `/usr/local/bin/` に `httpd` コマンドをコピーする。
  - (b) `/bin/` に `tcpd`, `ifconfig`, `add_route`, `ping` などをインストールされていることを確認する。なければインストールする。
  - (c) ソースファイル `/etc/httpd.conf` ファイルを用意する。このファイルの書き方は `l4minix/src/commands/httpd` を参照
  - (d) `/etc/httpd.mtype` を用意する。ソースファイル `l4minix/src/commands/httpd` を参照
  - (e) `html` ファイルを用意する。
2. VMware のネットワーク環境を Host-only に設定して起動する。
3. Linux 上で `ifconfig` コマンドにより、VMware の IP アドレスを調べる。

```
# /sbin/ifconfig
eth0      .....
          .....
          vlnet1 .....
          Inet addr:192.168.211.1  Mask:255.255.255.0
          .....
```

これは、VMware の IP アドレスが 192.168.211.1、ネットマスクが 255.255.255.0であることを示している。この場合、L4minix-e の IP アドレスは 192.168.211.2、ネットマスクは 255.255.255.0 になる。

4. L4minix-e に以下の内容の `/etc/netw3` ファイルを作る。

```
ifconfig -h 192.168.211.2 -n 255.255.255.0
add_route -g 192.168.211.1
ping 192.168.211.1
tcpd http /usr/local/bin/httpd &
/usr/local/bin/httpd &
```

5. Shell で `/etc/netw3` ファイルを実行する。

```
# sh /etc/netw3
```

6. Linux 環境の WEB ブラウザで、192.168.211.2 (の適当な html ファイルの場所: 例えば `http://192.168.211.2/www/index` ) を指定すると、内容が表示される。

## 2.10 L4minix-e のコンパイル法

1. 解凍する。

```
# gunzip l4minix-yy-mm-dd.tar.gz
```

2. TAR ファイルを展開する。

```
# tar xvf l4minix-yy-mm-dd.tar
```

3. `l4minix/src` ディレクトリーに移る。

```
# cd src
```

4. Make する。

```
# make      l4minix/src/ ディレクトリーで make する。
```

バイナリファイル (`task/hw/hw`, `task/clock/clock`, `task/hw/hw`, `task/mm/mm`, `task/fs/fs`, `task/inet/inet`, `task/sh/sh`, `test/apl/...` ) が生成される。

5. バイナリファイルをブートフロップの "simple" directory にコピーする。

- ```
# make floppy
(これは src/task/*/*.gz ファイルを floppy /mnt/floppy/simple にコピーする.)
```
6. ....
7. あるプログラム (例えばファイルサーバー fs のプログラム) を修正した場合には、その directory で make し、ブートフロッピーの simple/ directory にコピーすればよい。
- ```
# cd task/fs
... プログラム修正...
# make
# cp fs.gz /mnt/floppy/simple
```
8. 14minx/src/ ディレクトリーで make clean を行えば、全てがリカーシブに make clean される。

【注意】Makefile の定義は不完全なので、ヘッダーファイルを修正した場合は make clean してから make して下さい。最近の PC は十分に速いので、たいして待たされない。

## 2.11 L4minix 応用プログラムの開発

- 応用プログラムのリンクの仕方 (Makefile の書き方) については、“14minix/src/test/ap1/Makefile” を参照されたい。スタートアップルーチン (crt0-x96.o)、ライブラリー (-ll4mx, -ll4mxc 等) に注目のこと。
  - 実行ファイルを作ったら、それを VMware の仮想ディスクにコピーする。 次項参照
  - ライブラリー：ライブラリーのソースは、14minix/src/lib/.... に、ライブラリーの バイナリーは、14minix/lib/... にある。以下で ( ) 内はリンカーに与えるパラメータを示す。
- libl4mx.a ( -ll4mx ) : L4minix に必要な基本ライブラリー。
- libl4mxc.a ( -ll4mxc ) : UNIX の libc.a 相当のライブラリー。
- libiox.a ( -liox ) : printfz(), putcz(), getcz() 等を提供。(OS タスクは、TTY タスクが立ち上がる前にメッセージを出力するので、TTY タスクが提供する printf() は使えない。そこで、printfz() を使う必要がある。)

## 2.12 Linux から VMware の仮想ディスクにファイルをコピーする手法

### (1) VMware version-2 の場合

VMware version-2 は vmware-mount というコマンドが用意されており、以下の手順で簡単にホスト OS (Linux) 上で VMware の仮想ディスクにファイルをコピーすることができる。

#### 【環境】

- VMware の仮想ディスクは、~/vmware/L4/L4.dsk であるとする。
- Linux 上では、マウントポイントとして /mmt/vm が用意されているとする。

#### 【手順】

1. Linux 上のウィンドーにて
2. # cd ~/vmware/L4 ( /vmware/L4/L4.dsk の directory)
3. # su root

```
4. # vmware-mount.pl L4.dsk 1 -t minix /mnt/vm
```

ここに、各パラメータの意味は、以下の通りである。

L4.dsk : Linux 上での仮想ディスクのファイル名

1 : L4minix は (仮想) ディスクの partition-1 を使う

-t minix: L4minix のファイルシステムは minix

/mnt/vm : マウントポイント

5. Linux 上の別のウィンドーにて

```
# ls /mnt/vm
```

```
# cp myprog /mnt/vm
```

の如く、/mnt/vm に L4minix のファイルシステムがマウントされている。

6. コピーなどの作業が終わったら、4) のウィンドーにて CTL-c を打鍵すると、マウントがはずされる。

7. VMware 上で L4minix を起動して、該当プログラムを走らせる。

## (2) VMware version-3 以降の場合

VMware Version-3 以降では vmware-mount は使えないので、以下を手法で行う。

1. VMware の RAW disk を使う。(VMware のマニュアルを参照)

2. L4minix 上で FTP を使う。





**Main thread** (ローカルスレッド番号 = 1) Minix プロセスは main thread に反映される。

**IPCX thread** (ローカルスレッド番号 = 63) タスク間でバッファコピーを行う時に、このスレッドが使われる。

**Signal thread** (ローカルスレッド番号 = 61) 非同期事象が生じてシグナルハンドラーを起動する場合には、(主として mm タスクが) Signal Thread に メッセージを送る。Signal thread は、指定された関数を実行する。

**Driver threads** デバイスのドライバーは、Minix ではカーネルモードで走る Minix カーネルに含まれているが、L4minix ではユーザモードで走るサービスタスクに含まれている。つまり、ドライバープログラムも APL プログラム同様に保護された環境で走るので、プログラム開発が容易化される。

ドライバーのスレッドとサービスタスクのスレッドとはメッセージ結合なので、別プロセスに配置することも可能であるが、L4minix-e では両者を同居させてみた。

### 3.3 OS タスクの主要スレッド (参考)

OS サービスを実行している主要スレッドの ID (l4\_threadid\_t 値) を、以下に示す。

○ SIGMA0	[4, 2, 0, 1]	0x04020001
○ MM	[4, 4, 0, 1]	0x04040001
○ HW-server	[4, 5, 0, 1]	0x04050001
○ CLOCK-server	[4, 6, 0, 1]	0x04060001
• SYN_ALARM_TASK	[4, 6, 3, 1]	0x04060c01
○ FILE_SERVER	[4, 7, 0, 1]	0x04070001
• IDE_DRIVER	[4, 7, 1, 1]	0x04070401
• FLOPPY_DRIVER	[4, 7, 2, 1]	0x04070801
• RAMDISK_DRIVER	[4, 7, 3, 1]	0x04070c01
○ INET_SERVER	[4, 8, 4, 1]	0x04081001
• ETH_DRIVER	[4, 8, 3, 1]	0x04080c01
○ TTY_SERVER	[4, 9, 0, 1]	0x04090001

### 3.4 OS タスクへのパラメータ引き継ぎ

menu.lst の module 行は、該当 OS タスクの main(int argc, char \*\*argv) の第一引数として引き継がれている。

Ex. module = (fd0)/simple/fs.gz partition=1 は fs タスクの main( ) では以下のように参照できる。  
 argv[0]: "(fd0)/simple/fs.gz partition=1"

## 第4章 L4-Minix のタスク間通信

### 4.1 Minix メッセージを L4 メッセージで運ぶ

Minix の message は、L4 の message に乗せて転送される。Minix メッセージの L4 メッセージへの乗せ方を 図 4.1 に示す。Minix メッセージの `m_source` は送り主のアドレス (タスク番号) であるが、L4 ではシステムコールが送り主アドレスを返すので、メッセージ上に明示的に置く必要はない。

また、メッセージの宛て先・送り主の指定は、Minix ではタスク (プロセス) 番号であるが、L4minix では L4 のスレッド識別子を 32 ビットの `unsigned int` に読み換えた `l4_threadid_t.raw` を使う

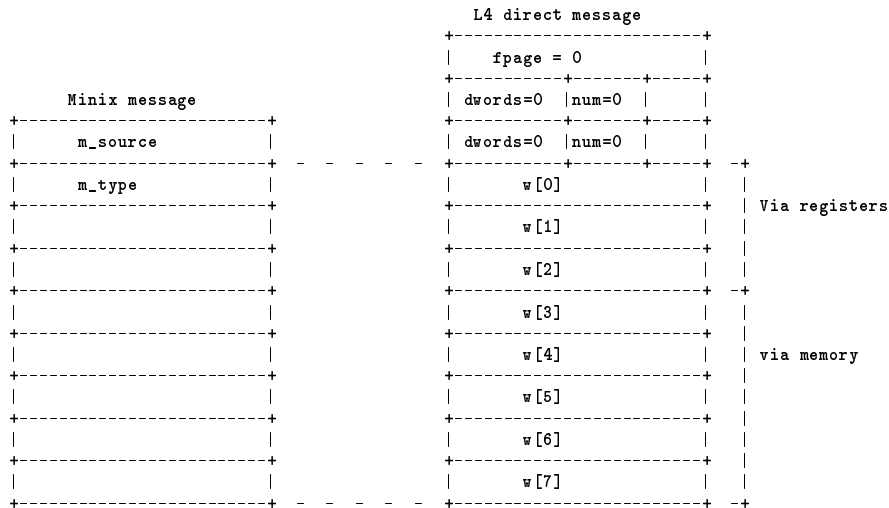


図 4.1: L4 メッセージへの Minix メッセージの乗せ方

Minix のメッセージは、ライブラリルーチン `_send()`, `_receive()`, `_ewecvrec()` により、送受される。`l4minix/include/Minix/minix/syslib.h` は以下の定義を含むので、プログラム中では `send()`, `receive()`, `sendrec()` として記述できる。(Minix の流儀を引き継いだ)。

```
#define send      _send
#define receive  _receive
#define sendrec  _sendrec
```

(1) メッセージ送信

```
int _send(unsigned to,          //メッセージの宛て先 (L4_threadid_t)
          int  msg[]);        // Minix メッセージのアドレス
```

(2) メッセージ受信

```
int _receive(unsigned from,     //メッセージの送り主 (L4_threadid_t)
```

```
int msg[]); // Minix メッセージのアドレス
from = 0 を指定すると、Open wait、つまり誰からもメッセージを受け取る。
```

## (2) メッセージ送受信

```
int _sendrec(unsigned to_from, //メッセージの宛て先 (L4_threadid_t)
             int msg[]); // Minix メッセージのアドレス
to_from で指定したスレッドにメッセージ msg[] を送り、かつ返答を msg[] に受け取る。
```

## 4.2 バッファ転送

### 4.2.1 単純なバッファ転送

異なるタスク(論理空間)の間でも、簡単・効率的にバッファをコピーすることが出来る。図 4.2 にバッファコピーの仕組みを示す。

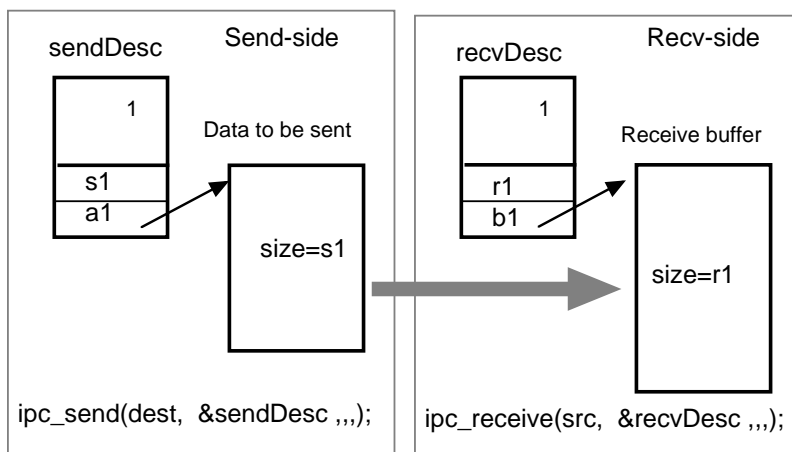


図 4.2: バッファ間転送

### 4.2.2 Scatter/gather バッファ転送

また、一つのバッファの内容をそれよりサイズの小さい複数のバッファにコピーする (Scatter copy) ことも、複数のバッファ内容を集めて一つのバッファにコピーする (Gather copy) ことも出来る。図 4.3 に Scatter/gather 型バッファコピーの仕組みを示す。

### 4.2.3 Scatter/gather 型バッファ転送のためのライブラリ関数

Scatter/gather 型バッファ転送を簡単に記述出来るように、ライブラリルーチン `set_send_dope()`, `_send_indirect()`, `set_receive_dope()`, `_receive_indirect()` 等を用意している。

ライブラリルーチンの記述例を次に、また 図 4.4 にこの時使われるデータ構造を示す。

送信側では、`set_send_dope()` で送信記述子を作り、`_send_indirect()` でデータ転送を行う。

受信側では、`set_receive_dope()` で受信記述子を作り、`_receive_indirect()` でデータ受信を行う。

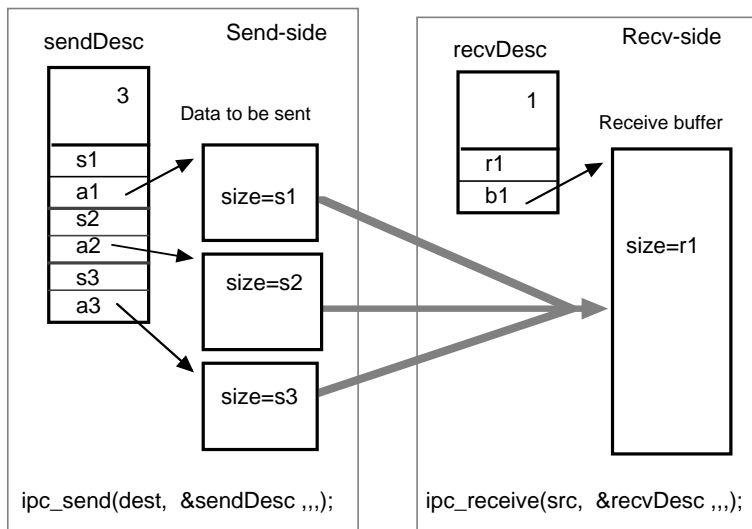


図 4.3: Scatter-gather メッセージ転送

前章“プロセスの概形”で述べた IPCX スレッドが、このメッセージ転送を行う。

ライブラリルーチンの詳しい使い方は、別冊の“プログラミングマニュアル”を参照されたい。プログラム例を以下に示す。

#### 【送信側】

```
int recmsg[10];
for(int i=0; i<10; i++) recmsg[i]=0;
set_send_dope(recmsg, 0, バッファアドレス, バッファサイズ, 0);
_send_indirect(0, w0, w1, w2, sndmsg, 1);
```

#### 【受信側】

```
int recmsg[14];
for(int i=0; i<14; i++) recmsg[i]=0;
set_rcv_dope(recmsg, 0, バッファ1 アドレス, バッファ1 サイズ, 0);
set_rcv_dope(recmsg, 0, バッファ2 アドレス, バッファ2 サイズ, 1);
from = _receive_indirect(0, &x, &y, &z, recmsg, 2);
```

## 4.3 タスク間のインタフェース

タスク間は、メッセージでのみやりとりをする。メッセージは同期式であり、送信側は受信側がレスポンスを返すまでブロックされる。

ファイルサーバー、INET サーバー、TTY サイバー等は Minix プログラムを移植したので、シングルスレッド動作である。従って、複数の要求を同時に処理することは出来ない。しかしながら、ある要求が I/O 動作待ちになっている時、他の要求が受け付けられないと非常に効率が悪くなるので、その対策として SUSPEND/REVIVE の仕組みが用意されている。

いずれの場合も Client 側は以下の形式である。

1. 要求メッセージを送信する。
2. 返答メッセージを待つ(中断)
3. 返答メッセージを受信したら、処理を再開する。

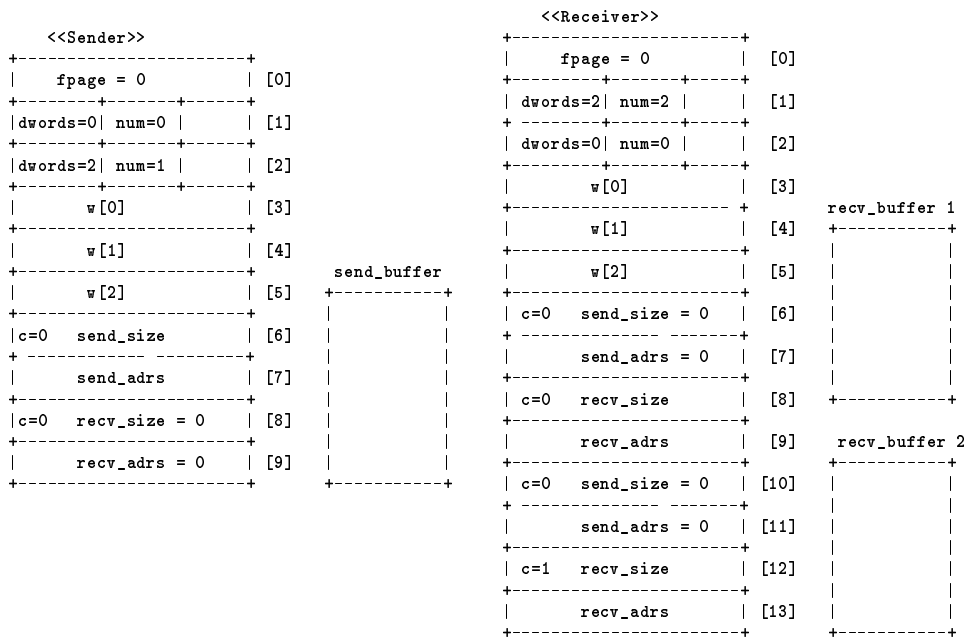


図 4.4: Buffer copy メッセージ転送

### 4.3.1 サーバーが直ちに返答する場合

システムコールの大部分はこのケースに該当する。サーバー側処理も非常に簡単で、以下のように行われる、【Server 側】

1. 要求メッセージが送られてくるのを待つ。
2. 要求メッセージを受信したら、それを処理する。
3. 返答メッセージを返し、次の要求メッセージを待つ。

### 4.3.2 サーバーの中で中断が生ずる場合

例えば ファイルサーバーにデータ読み出しを要求した所、該当データがキャッシュに載っていないくて HDD からの読み出しが必要になった場合が、このケースに該当する。この間に、別の Client から要求メッセージを送ってきた場合、サーバーはその要求メッセージを受けつけて、処理を試みるべきである。

【Server 側】

```
while(1) {
    要求メッセージあるいは動作完了のメッセージを受信する。
    switch (メッセージ種別){
    case 要求メッセージ :
        {
            要求を処理する。
            if (処理完了) {
                返答メッセージを Client に返送する。
                continue ; // 次のメッセージ受信
            }
        }
    }
```

```
    else { //処理の中断
        たとえば HDD ドライバーに読み出し要求を送る。
        処理の再開に必要なデータをプロセステーブルにセーブする。
            (Client は返答待ちで、中断のまま)
        continue ; // 次のメッセージ受信
    }
}

case 動作完了メッセージ :
{ //例えば HDD ドライバーが動作完了時に送ってくる
  セーブして置いた再開データをリカバーする。
  この要求を出したクライアントに返答メッセージを送る。
  これにより、中断していた client は中断を解かれる。 //REVIVE
  continue ; // 次のメッセージ受信
}

default: ....
}

} // while
```



## 第5章 プロセスマネジャー (mm タスク)

プロセスマネジャー (mm タスク) は、論理メモリ域の管理と、プロセス (OS タスクも含む) の生成・消去とを司っている。論理メモリ域は、Paging 方式で実現している。(実メモリが不足した場合のページの swapping はまだ実装していない)。

プロセスマネジャーの概形を図 5.1 に示す。MM-Server がプロセスの生成・消去などを管理し、MX-pager が論理メモリ域の管理を行っている。

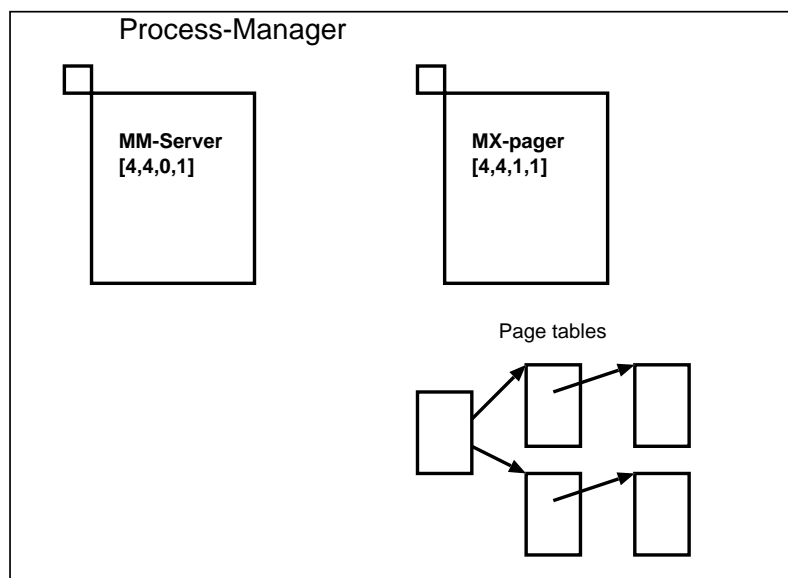


図 5.1: プロセスマネジャーの構造

プロセスマネジャーのソースプログラムは `14minix/src/task/mm` ディレクトリにある。Minix の名前を引き継いだため、ソースプログラムではプロセスマネジャーを pm タスク (process manager) とは呼ばずに mm タスク (memory manager) と呼んでいる。

プロセスマネジャーは、以下のように立ち上がる。

- ブートプログラム `rmgr` により、本プロセスマネジャーが立ち上げられる。
- `sigma0` から利用可能なメモリ (ページフレーム) を引き継ぐ。
- メモリページャー (`src/task/mx-pager.c` で定義されているスレッド) を立ちあげる。
- mm タスク以外の OS タスクにメモリを割り付け、プログラムをロードして、起動する。

### 【参考: Minix】

Minix はページングによる仮想ページは提供していない。Minix のメモリ管理は非常に単純で、各プロセスには物理的な連続番地を割り当てている。例えば、あるプロセスを物理アドレス から割り付けたとすると、この



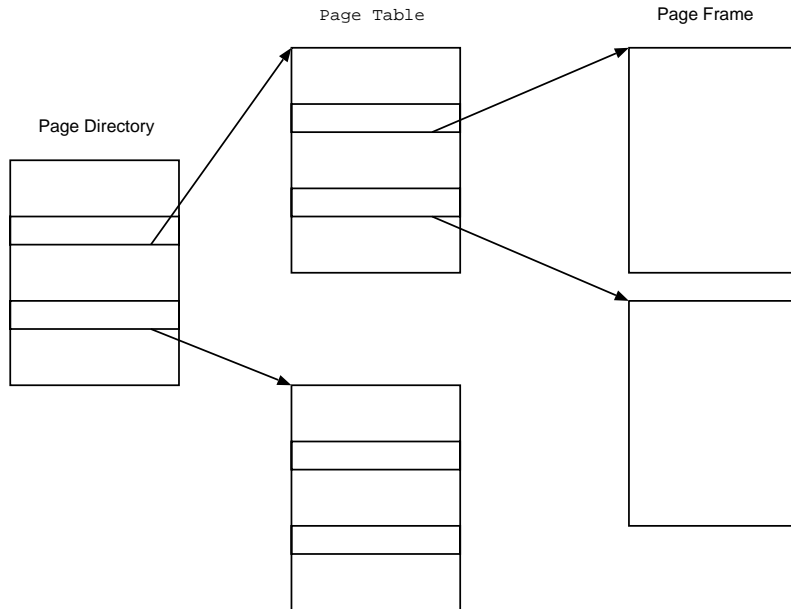


図 5.2: ページテーブル

```

void mxpager_thread( )
{
  for(;;)
  {
    (1) 要求メッセージ (Page fault 等により生成される) を受信する
        l4_ipc_receive (&client, (void *) 0x01, & xx, & yy, & zz,
                       L4_IPC_NEVER, &result);
        /* パラメータ xx には、一般に Page fault の生じた番地が乗っている */
    (2) yy == 0xDA00 ~ 0xDA10 ならば、DMA 域の要求なので
        DMA 域 (物理アドレス 1 M ~ 16M) のページをマッピングする。
        0xDA00 の場合は 1 ページ、0xDA01 の場合は 2 ページ、0xDA02 の場合は 4 ページ、...

    (3) yy = 0xAD00 ならば、そのページの物理アドレスを求める。

    (4) 4996 <= xx < 1M ならば、
        物理アドレス param1 番地を含むページをマッピングする。

    (5) 0 <= param0 < 4096 の場合:
        Trampoline code が入っているページを、0 ページとして Map する。
        Trampoline code は position independent code で、ページ先頭が Entry point である。

    (6) それ以外の場合:
        空いている任意のページをマッピングする。

    (7) 返答を返す。
        l4_ipc_send (client, (void *) L4_IPC_SHORT_FPAGE, dw0, dw1, dw2,
                   L4_IPC_NEVER, &result);
        // IPC は、ページ をクライアントの page fault を生じたページにマッピングする。
  } //for
} //pager_thread

```

### 5.3 プロセス=タスクのメモリエイアウト

L4minix-e のプロセス生成としては、UNIX 同様にコピープロセスを生成する `fork()` と、別個の新しいプロ

セスを生成する `spawn()` とを用意している。

各プロセスは、個別の論理空間で走るので、任意の論理番地を与える事が可能である。但し、OS タスクのデバッグの都合から、今の所各 OS タスクには異なる論理アドレスを与えている。APL プロセスには Linux と同様の論理アドレス ( $0x08048000=128M + 288K$ ) を与えている。プロセスのメモリレイアウトを、図 5.3 に示す。

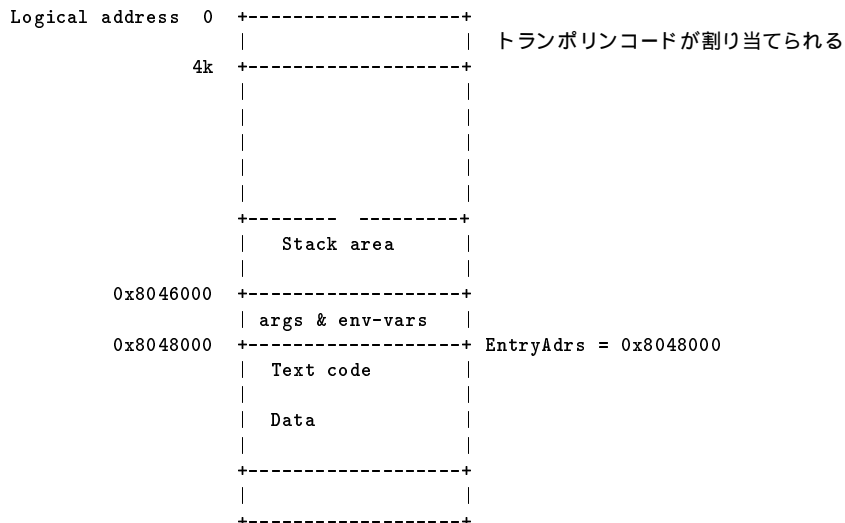


図 5.3: プロセスのメモリレイアウト

1. トランポリンコードは、プロセスの立上時に実行されるコードである (後述)。
2. プログラムの default の開始アドレスは、Linux の ELF ファイルに合わせて  $0x8048000$  番地とした。但し、OS タスクの場合はデバッグのために (論理アドレスをみれば、どのタスクかが分かるように) それぞれ異なるアドレスを与えている。
3. 引数および環境変数を引き継ぐために、1 ページをリザーブしている。(  $0x8046000$  番地のページ)
4. スタックは main thread が使うスタック域であり、引数・環境変数ページの前から若番側に伸びる。(Linux ではユーザ論理空間の最高番地から下方に伸びるが、これは本来の ELF ファイル形式の意図には反している。何故かを考えると面白い。)

## 5.4 Spawn の仕組み

プロセス生成のシステムコールとしては、UNIX 標準である `fork()` の他に、新プロセスを生成して指定したプログラムを起動する、つまり `fork()` と `exec()` を一緒にした `spawn()` も用意している。まず `spawn()` から説明する。システムコール `spawn()` を呼ぶと、以下のように処理が進む。

1. `spawn()` ライブラリルーチンを実行 (プロセスマネジャーに SPAWN メッセージを送る)。
2. `do_spawn()` を実行
3. `spawn_exec()` を実行
4. `start_prog()` を実行
5. `l4_task_new()` を実行して新プロセスを生成
6. 新プロセスのトランポリンコードを実行
7. 新プロセスのスタートアップルーチンを実行
8. 新プロセスの `main()` ルーチンを起動

### 5.4.1 spawn() ライブラリ関数

spawn() ライブラリー関数は、/src/lib/l4mxc/posix/\_spawn.c にある。  
spawn() は、exec() に対応した複数のパラメータ引き継ぎ法を持っている。

```
int spawnl(char *filename, char *arg, ...);
int spawnle(char *filename, char *arg, ...);
int spawnve(char *filename, char *argv[], char *envp[]);
int spawnvp(char *name, char *argv[]);
```

spawn() ライブラリーでは、引数と環境変数を図 5.4 のように編集して、プロセスマネジャー (MM タスク) に SPAWN メッセージを送る。SPAWN メッセージのパラメータは、( path のバイト長、path のアドレス、arg\_env ブロックのサイズ、 arg\_env ブロックのアドレス) である。

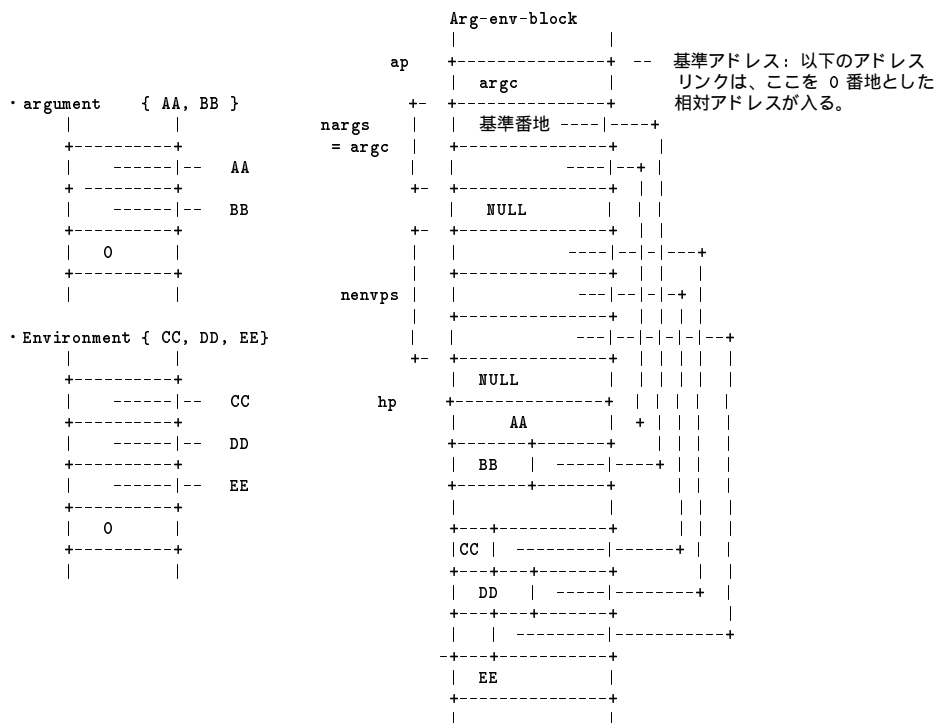


図 5.4: 引数・環境変数の引継

### 5.4.2 do\_spawn()

これからがプロセスマネジャーの仕事である。

do\_spawn() のソースプログラムは、src/mm/exec.c: do\_spawn() にある。プロセスマネジャーは、SPAWN メッセージを受信すると do\_spawn() を呼び、以下の処理を実行する。

1. child プロセスに プロセス管理テーブル (mproc テーブル) を割り当て、そこに親プロセスの内容をコピーする。
2. FS サーバーに FORK メッセージを送り、fproc テーブルを準備させる。

3. `spawn_exec()` を呼ぶ

### 5.4.3 `spawn_exec()`

ソースプログラムは、`src/mm/exec.c`: `spawn_exec()` である。

1. `copyin()` を使って、プログラム名 (ファイルの path 名) を引き継ぐ (L4 の IPC を効果的に使えば、FORK メッセージの中にプログラム名を含めて 1 メッセージ化できるが、ここでは Minix の手法を踏襲している)
2. FS サーバーに CHDIR メッセージを送る。
3. `copyin()` を使って、`arg-env` ブロック (引数環境変数ブロック) をローカルバッファ域の `arg_buf` 域に読み込む。  
引数と環境変数の引き継ぎ方については、後述。
4. `arg-env` ブロック (引数環境変数ブロック) のアドレスリンクが、新プロセスの `ARGBLOCK_ADRS` 番地 (現在は、`0x8046000`) で有効になるように、各アドレスリンクに `ARGBLOCK_ADRS` を加算する。
5. `start_prog(newtask, fd, arg_buf, ARGBLOCK_ADRS, arg-env サイズ, 1)` を呼び出す。

### 5.4.4 `start_prog()`

ソースプログラムは、`src/task/mm/elfexec.c` にある。

1. `l4_task_new()` (タスク ID, 優先度, ESP, EIP, ページャー) を呼んで、新タスクを生成する。ここで設定されるパラメータは以下の通りである。  
タスク ID 新規生成するタスクの ID を与える。  
優先度 優先度を与える。  
ESP main スレッドのスタックポインター: プログラム先頭で設定されるので 0 を与える。  
EIP main スレッドのプログラムカウンタ: 0 を与えておく。これがどのように動作するかは、後述の `trampoline` コードでのお楽しみ。  
ページャー このタスクの main スレッドが `Pagefault` を起こした場合に、ここで指定されたページャーに処理が依頼される。勿論 `mx=pager` を与える。
2. 実行プログラムファイルから順次コード・データを読み出し、`load_segment()` を使って新タスクに設定する。
  - `load_segment()` が新タスクに書き込もうとすると、まだページが割りつけてないので `pagefault` が発生
  - `Pagefault` が生ずるたびに、`mx-pager` の `pager_thread` がページを `map` する。
  - `Page` が `map` されたら、先程のコード・データが書き込まれる
3. 同様に BSS 域をクリアする。
4. `load_segment()` を用いて、`arg_buf` 域に入っている `arg-env` ブロック (引数環境変数ブロック) を新プロセスの `ARGBLOCK_ADRS` 番地 (現在は、`0x8046000`) に書き込む。
5. `trampoline` コード (`src/task/mm/trampoline.s`) に、`TRAMPOLINE_LAUNCH` メッセージを送る。

```
w[0] = TRAMPOLINE_LAUNCH
w[1] = 新プロセスでの arg-env ブロック (引数環境変数ブロック)
      のアドレス = ARGBLOCK_ADRS 番地
w[2] = プログラムの Entry ドレス
```

【参考】 start\_prog() の疑似コード

```
void start_prog( ..... )
{
  (1) ELF ヘッダーを解析する。
  (2) タスク生成システムコール
      14_task_new(newtask, 255, 0, 0, mx_pager);
          // 第 5 パラメータ mx_pager は、新タスクの pager として
          // 前述の mxpager_thread を指定。
  (3) ELF ファイルの各セグメントに対して繰り返す。
      for (... 各セグメントに対して...)
      {
        ELF ファイルの該当セグメントに lseek() する。
        ELF ファイルからコードを読み出し、これを新タスクにトランポリンコード (後述)
        を介して送り込む。トランポリンコードにより、新タスクではメモリが割りつけられ、
        プログラムが書き込まれる。
        ...
      }
  (4) 新タスクにトランポリンコードマジックを使って環境変数や引数を引き継ぐ。
  (5) トランポリンコードマジックを使って新タスクを起動する。
}
```

## 5.5 Trampoline コード

L4minix-e では、プロセスマネジャーはカーネルプログラムではなく、普通のプロセスである。プロセスマネジャーが新プロセスを立ち上げて指定プログラムを走らせるには、異なる論理空間を相手にしなければならない。同一論理空間にプログラムコードを書き込でプログラムを起動する事は簡単であるが、相手が別論理空間となるとそうはいかない。

つまり、プロセスマネジャーが新しいプロセスを立ち上げることは、国交の無い国と交渉を開始するように特殊な処理が必要となる。そこでトランポリンコードを使ったマジックが必要となる。本コードは Karlsruhe 大学の学習用 OS ChacmOS (<http://14ka.org>) の trampoline コードを参考に開発した。

基本的なアイデアは、以下の通りである。

1. プロセスマネジャーは 14\_task\_new() を実行して、マイクロカーネルに新プロセスを用意させる。パラメータのプログラムカウンターの初期値は 0 番地としておく。これにより、新プロセスは 0 番地から実行を開始しようとする。
2. 新タスクにはまだメモリが割りつけられていないので、0 ページでページフォールトが生ずる。
3. mxpager は空きページを探して、そこにトランポリンコードをコピーし、このページを新プロセスの 0 ページとしてマップする。これにより、新タスクの 0 番地にトランポリンコードが埋め込まれる。  
(現在は、trampoline コードは論理空間の 0 番地ページに割り当てているが、これは将来修正する予定である。)
4. プロセスマネジャートランポリンコードが会話して、プログラムコードの書き込みやプログラム起動を行う。

trampoline コードは、プロセスマネジャーから以下のメッセージを受信して、以下仕事を行う。メッセージ種別は、第 0 パラメータ、つまり EAX レジスタで引き継がれる (w[0] = EAX)。

**TRAMPOLINE\_RECEIVE** プログラムコードを受け取って、論理空間の指定アドレスに書き込む。

**TRAMPOLINE\_ZERO\_ZONE** 論理空間の指定領域をゼロクリアする。

**TRAMPOLINE\_LAUNCH** プロセスマネジャーから制御を受け取って、新プロセスの実行をスタートする。

トランポリンコード (アセンブラ記述) の処理内容を、以下に疑似コードで示す。

```

# How the trampoline program works ---
_trampoline: // zero address of each logical space.
    goto _loop;
    asm(" .space 64" ); // Stack area
-loop:
while(1) {
    int msg[10];
    ESP = 64; // Stack top
    l4_ipc_receive(0x4050001, 0, &w0, &w1, &w2, L4_IPC_NEVER, &rc);
    // EDI: w0, EBX: w1, EDI:w 2
    switch(w0){
    case TRAMPOLINE_RECEIVE: //0, w1=adrs, w2=size
        msg[0] = 0;
        (sizedope)msg[1] = {2,1};
        msg[2] = 0;
        msg[8] = w2; //size
        msg[9] = w1; //address
        l4_ipc_receive(0x4050001, msg, &w0, &w1, &w2, L4_IPC_NEVER, &rc);
        continue;
    case TRAMPOLINE_ZERO_ZONE: // 1, w1=adrs, w2=size
        ECX = (w2 + 3)/4;
        EDI = w1;
        EAX = 0;
        asm( "rep ; stosl ");
        continue;
    case TRAMPOLINE_NEW_PAGER: //2, w1 = pagerid
        l4_thread_ex_regs(0, -1, -1, .....);
    case TRAMPOLINE_LAUNCH: // w0 = 3, EBX = w1= argc, EDI = w2 = entry
        EAX = 1024; // EAX: base adrs of argc-arg-env-block
        ECX = 1756 // Magic number
        // EBX = argc,
        JUMP *EDI; // EDI = w2 = entry
    }
} //while

```

例えば、TRAMPOLINE\_LAUNCH の処理では、プロセスマネジャーからは以下のパラメータを受け取り：

1. w[0] = EAX = TRAMPOLINE\_LAUNCH を指示する。
2. w[1] = EBX = 新プロセスでの arg-env ブロック (引数環境変数ブロック) のアドレス
3. w[2] = EDI = 新プログラムの Entry ドレス

以下のように新プロセスのスタートアップルーチンを起動する。

1. EAX = EBX = 新プロセスでの arg-env ブロック (引数環境変数ブロック) のアドレス
2. ECX = 1756 (マジック番号)
3. 新プログラムの Entry アドレス (EDIレジスタの値 = これは次に述べるスタートアップルーチンを指している)へジャンプする。これにより、スタートアップルーチンに制御が渡される。

## 5.6 スタートアップルーチン crt0-x86

スタートアップルーチン (ソースコードは、l4minix/src/lib/crt0/crt0-x86.S) は、プロセスの起動時に実行されるプログラムであり、プロセスの main() 関数はここから呼ばれる。

L4minix のプロセスの内、プロセスマネジャー (mm タスク) は rmgr ローダーにより立ち上げられ、それ以外はプロセスマネジャーにより立ち上げられる。

### (1) rmgr 起動の場合

EAX レジスタの値が 0x2BADB002 (ELF ファイルのマジック番号) ならば、これは rmgr ローダーにより立ち上げられたプロセス (つまりプロセスマネジャー) である。

この場合は、IPCX スレッドや signal スレッドを立ち上げてから、main() 関数を呼び出す。

### (2) プロセスマネジャー起動の場合

プロセスマネジャーから起動された場合は、trampoline コードを經由して本スタートアップルーチンが起動される。Trampoline コードでは、EAX レジスタに値 1756 を設定している。

従って、EAX レジスタの値が 1756 ならば、これはプロセスマネジャーにより立ち上げられたプロセスである。

この場合は、先ずスタック域に環境変数リストのアドレス、引数リストのアドレス、および引数の個数を積む。つまり、main(argc, argv, envp) を設定する。

```

|-----|
|-----|
|  argc  | Cf. main(argc, argv, envp)
|-----|
|  argv[0] のアドレス |
|-----|
|  envp[0] のアドレス |
|-----|
老番

```

ついで、IPCX スレッドや signal スレッドを立ち上げてから、main() 関数を呼び出す。

## 5.7 Fork と exec の仕組み

### 5.7.1 fork() ライブラリー関数

fork() ライブラリー関数 (ソースコードは src/lib/14mxc/posix/fork.c) は、プロセッサの EFLAG 及び汎用レジスタをセーブし、再開番地、スタックポインタを求め、プロセスマネジャーに “再開番地、スタックポインタ” を引数として、FORK メッセージを送る。

### 5.7.2 do\_fork()

プロセスマネジャーは FORK メッセージを受けると、do\_fork() (ソースコードは、src/task/mm/forkexit.c) を呼んで以下の処理を実行する。

1. 子プロセスの pid を割り当てる。
2. FS サーバーに FORK メッセージを送り、fproc テーブルを準備させる。
3. clone\_space() を呼んで、子プロセスの論理空間を割り付け、親プロセスのメモリー内容を子プロセスの論理空間にコピーする。
4. l4\_task\_new(タスク ID, 優先度, ESP, EIP, ページャー) を呼んで、子プロセスを生成する。ここで設定されるパラメータは以下の通りである。

タスク ID 新規プロセスの ID を与える。

優先度 優先度を与える。

ESP main スレッドのスタックポインタ初期値: fork() ライブラリから引き継がれた値を設定する (spawn() との違いに注意)

EIP プログラムカウンターの値: fork() ライブラリから引き継がれた値 (spawn() との違いに注意)

ページャー このタスクの main スレッドが Pagefault を起こした場合に、ここで指定されたページャーに処理が依頼される。勿論 `mx=pager` を与える。

5. 以上により、新プロセスでは `fork()` ライブラリから引き継がれた再開番地から処理が開始される。

### 5.7.3 `clone_space()`

`fork()` では、子プロセス用に親プロセスと同じ内容をもった論理空間を生成する必要がある。このために、図 5.5 に示すように、`mx-pager` が管理しているページテーブルを使って、親と同一内容をもった論理空間を生成する。

現在はテキスト域も新たにメモリーが割り当てられ内容がコピーされるが、今後 “Copy on Write (COW)” を実装してこの無駄を省く予定である。

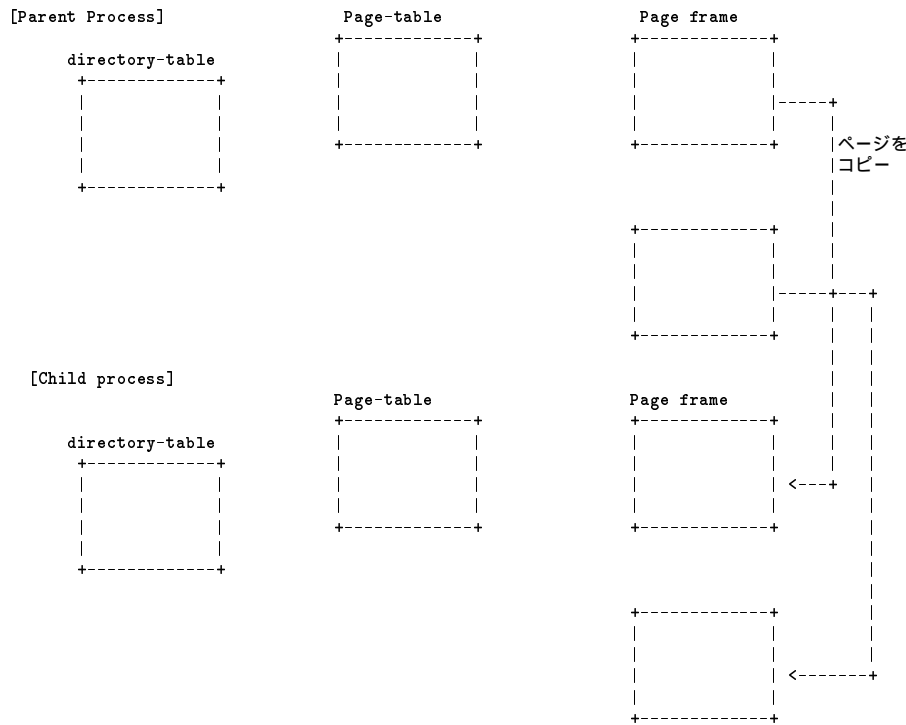


図 5.5: メモリー空間の複製

### 5.7.4 `exec()` システムコール

`exec()` システムコールの処理は、`apawn()` システムコールの後半の処理から容易に類推がつくので、説明は省略する。

## 第6章 HWサーバーとCLOCKサーバー

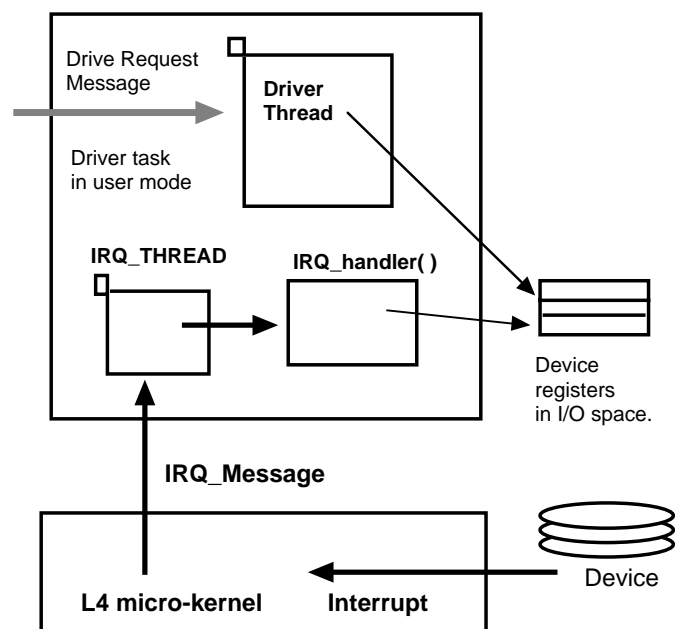


図 6.1: 割り込み処理



## 第7章 ライブラリ

### 7.1 L4mx ライブラリ

ソースプログラム `l4minix/src/lib/l4mx`

メッセージの送受信 (`send()`, `receive()`, `sendrec()` など)、I/O ポート制御などの primitive な機能を提供している。

L4mx ライブラリーは、メッセージの送受信 (`send()`, `receive()`, `sendrec()` 他)、I/O ポート制御等の基本的な機能を提供している。ソースプログラムは、`l4minix/src/lib/l4mx` ディレクトリーにある。

### 7.2 L4mxc ライブラリ

L4mxc ライブラリーは、UNIX の `libc` ライブラリー相当の機能を提供している。ソースプログラムは、`l4minix/src/lib/l4mxc` ディレクトリーにある。

UNIX 構文のシステムコールは、ライブラリ関数により OS サーバーへのメッセージ通信に変換される。例えば、`read()` システムコールは、概略以下の内容である。

```
int read(int fd, void *buf, int nbytes)
{ message msg;      int r;
  msg.FD = fd;
  msg.SIZE = nbytes;
  msg.ADRS = buf;
  msg.m_type = READ;
  r = sendrec(FILE_SERVER, & msg);
  .....
  return リターン値;
}
```



## 第8章 ファイルサーバー (FS タスク)

### 8.1 ファイルサーバーの位置づけ

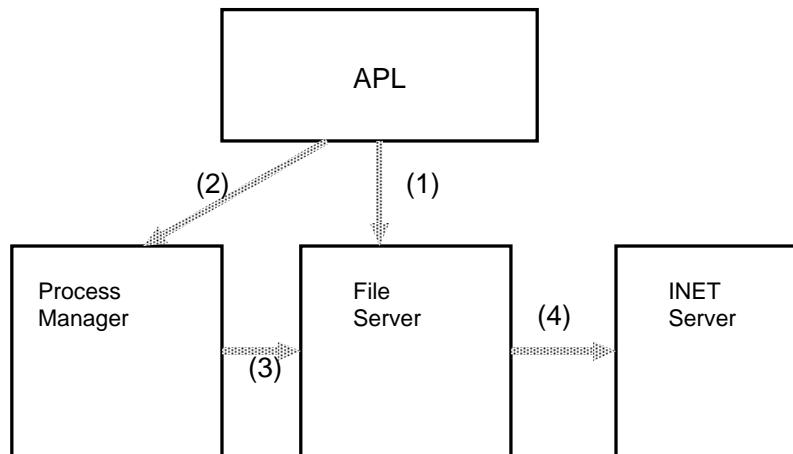


図 8.1: ファイルサーバーの位置付け

システムコールは、ライブラリルーチンにより以下のようなメッセージに変換する。

- (1) 応用プログラム ⇒ ファイルサーバー read, write, open, close, creat, link, unlink, chdir, mknod, chmod, chown, stat, lseek, mount, umount, sync, mkdir, unlink, ioctl, fcntl, ,,,  
ファイルやネットワーク関連のシステムコールは、ファイルサーバーへのメッセージになる。
- (2) 応用プログラム ⇒ プロセスマネジャー exit, fork, wait, waitpid, getpid, alarm, pause, kill, sigaction, ,,,  
プロセスの生成・消滅などに関するシステムコールは、メモリ・プロセスサーバーへのメッセージとなる。
- (3) プロセスマネジャー ⇒ ファイルサーバー chdir, exec, exit, fork, setgid, setsid, setuid, sync, unpause  
プロセスの生成・消滅に必要なファイル処理は、ファイルサーバーにメッセージを送って行ってもらう。
- (4) ファイルサーバー ⇒ INET サーバー NW\_OPEN, NW\_CLOSE, NW\_IOCTL, NW\_READ, NW\_WRITE, NW\_CANCEL  
ファイルサーバーは、ネットワーク処理を INET サーバーに要求する。

### 8.2 ファイルサーバーの内部構造

図 8.2 にファイルサーバーの構造を示す。

L4 マイクロカーネルのスレッドを活用してマルチスレッドサーバーとすることにより、効率とプログラム構造の両方を改善できるが (当然そちらが理想的だが)、Minix プログラムを流用したのでシングルスレッド処理になっている。

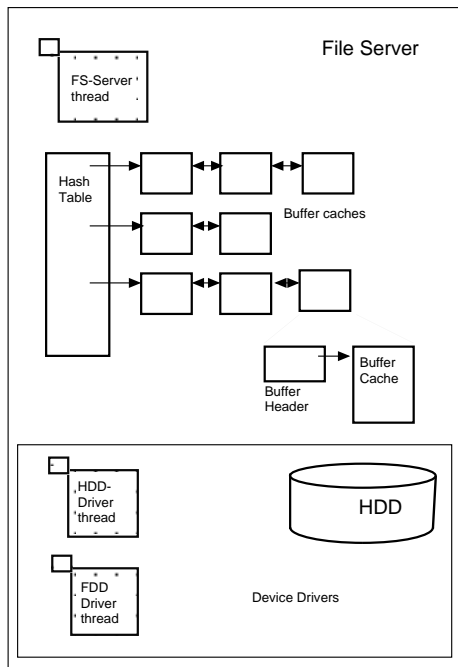


図 8.2: ファイルサーバーの構造

ファイルのキャッシュは、UNIX の伝統的な Buffer cache 方式を用いている。箇々の Buffer cache のサイズは 1 KB である。

なお、今後仮想記憶を利用したページキャッシュを実装する予定である。

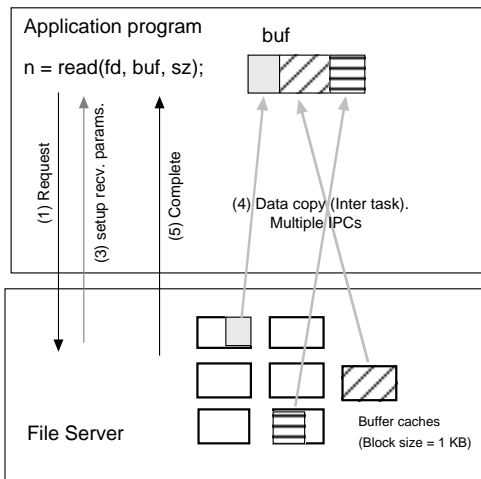
### 8.3 Scatter/gather による高速化

システムコール `read()`, `write()` 等は、バッファークッシュ単位でファイルの読み書きを行う。たとえば 16 KB のデータを読み出す場合、裏では 16 ~ 18 回のデータ転送が行われる。この手法によるファイル読み出しを、図 8.3 に示す。

システムコール `read_sg()`, `write_sg()` 等は、複数のバッファークッシュをまとめて、1 回のデータ転送で済ます手法である。たとえば 16 KB のデータを読み出す場合でも、1 回のデータ転送しか生じない。複数のバッファークッシュに対して一度で読み出しを行う Scatter/gather 型ファイル読み出しを、図 8.4 に示す。

### 8.4 ドライバプログラム

ファイルサーバーは、HDD-IDE ドライバ, Floppy ドライバ, RAMDISK ドライバを含んでいる。各々のドライバプログラムは、専用のスレッドを有しており、ファイルサーバーのメインスレッドとはメッセージで結ばれている。



- (1) A request message denoting "read(fd, buf, sz)" is sent to the FS.
- (2) File Server looks up target buffer caches.
- (3) File Server sends a message to the Service thread in APL to setup the data reception.
- (4) File Server sends data to the Service thread. (repeat for each buffer cache)
- (5) Completion message is returned.

図 8.3: 単純なファイルアクセス

## 8.5 DMA 転送

## 8.6 データ構造とプログラムロジック

ファイルサーバーのプログラムロジックは、基本的には Minix と同じであるので、Tanenbaum 先生の著書「オペレーティングシステム」を参照されたい。なお、データ構造をみればプログラムロジックは自ずから理解出来るものなので、以下に主要なデータテーブルを示す。

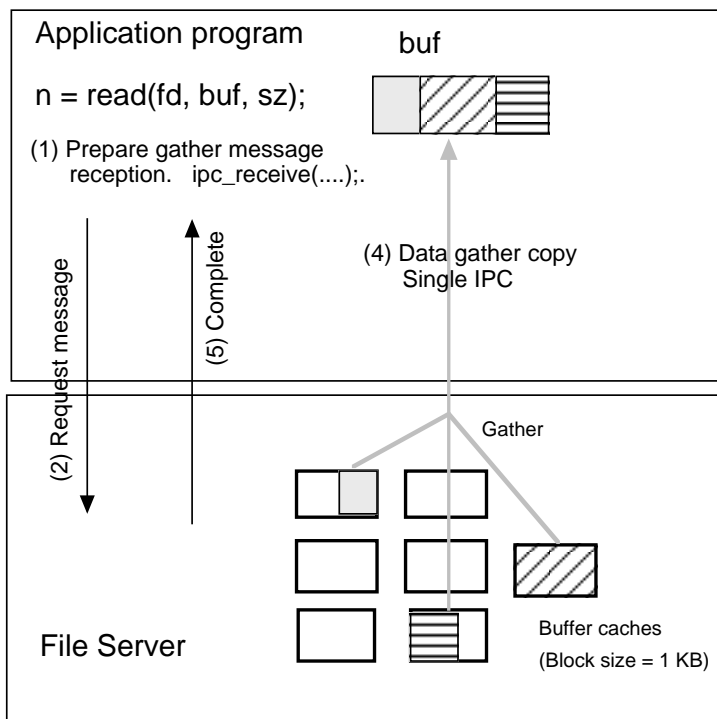


図 8.4: 高速化なファイルアクセス

```

【struct buf】
+-----+
|UNION{
| データブロック      | 1024 bytes
| DirectoryBlock
| 間接リンク
| Bit map ブロック
| } b;
|-----+
buf_t ----+ -b_prev;
|-----+
| b_hash;             | buf_t
| b_blocknr;         | block number
| b_dev;             | major | minor device
| b_dirt;            | CLEAN or DIRTY
| b_count;          | number of users of this buffer
+-----+
struct buf *buf_hash[NR_BUF_HASH]; /* the buffer hash table

【dmap】
+-----+
| ::
|-----+
| dmap_open;         | Open 関数
| dmap_rw;           | read/write 関数
| dmap_close;       | close 関数
| dmap_task;        | Thread-ID
+-----+
| ::                | dmap[]; // indexed by major device number

【filep[128]】
+-----+
| filp_mode          | RW bits
| filp_flags         | flags from open and fcntl
| filp_count         | how many file descriptors share this slot?
| inode *filp_ino    | pointer to the inode
| filp_pos           | file position
+-----+

【fproc テーブル】
+-----+
| struct fproc {
| fp_umask;          | mask set by umask system call
| inode *fp_workdir  | pointer to working directory's inode
| inode *fp_rootdir  | pointer to current root dir (see chroot)
| +-----+
| | filp *           | the file descriptor table
| | filp[OPEN_MAX]
| +-----+
| fp_realuid        | real user id
| fp_effuid         | effective user id
| fp_realgid        | real group id
| fp_effgid         | effective group id
| fp_tty            | major/minor of controlling tty
| fp_fd             | place to save fd if rd/wr can't finish
| *fp_buffer        | place to save buffer if rd/wr can't finish
| fp_nbytes         | place to save bytes if rd/wr can't finish
| fp_cum_io_partial | partial byte count if rd/wr can't finish
| fp_suspended      | set to indicate process hanging
| fp_revived        | set to indicate process being revived
| fp_sesldr         | true if proc is a session leader
| fp_task           | which task is proc suspended on
| fp_pid            | process id
| fp_cloexec        | bit map for POSIX Table 6-2 FD_CLOEXEC
+-----+

【inode テーブル】
+-----+
| i_mode            | file type, protection, etc.
| i_nlinks          | how many links to this file
| i_uid             | user id of the file's owner
| i_gid             | group number
| i_size            | current file size in bytes
| i_atime           | time of last access (V2 only)
| i_mtime           | when was file data last changed
| i_ctime           | when was inode itself changed (V2 only)
| +-----+
| | i_zone[10]      | zone numbers for direct, ind, and dbl ind
| |
| |
| +-----+
|
| 以下のフィールドは、DISK 上には存在しない。
| i_dev            | which device is the inode on
| i_num            | inode number on its (minor) device
| i_count          | # times inode used; 0 means slot is free
| i_ndzones        | # direct zones (Vx_NDZONES)
| i_nindirs        | # indirect zones per indirect block
| super_block *i_sp | pointer to super block for inode's device
| i_dirt           | CLEAN or DIRTY
| i_pipe          | set to I_PIPE if pipe
| i_mount          | this bit is set if file mounted on
| i_seek          | set on LSEEK, cleared on READ/WRITE
| i_update        | the ATIME, CTIME, and MTIME bits are here
+-----+

【Lock】
+-----+
| lock_type;        | F_BDLOCK or F_WRLock; 0 means unused slot
| lock_pid;         | pid of the process holding the lock
| inode *lock_inode; | pointer to the inode locked
| lock_first;      | offset of first byte locked
| lock_last;       | offset of last byte locked
+-----+

【Super block】
+-----+
| s_ninodes         | # usable inodes on the minor device
| s_nzones         | total device size, including bit maps etc
| s_imap_blocks     | # of blocks used by inode bit map
| s_zmap_blocks     | # of blocks used by zone bit map
| s_firstdatazone  | number of first data zone
| s_log_zone_size  | log2 of blocks/zone
| s_max_size       | maximum file size on this device
| s_magic          | magic number to recognize super-blocks
| s_pad           | try to avoid compiler-dependent padding
+-----+

```



## 第9章 INETサーバー (INETタスク)

総じて Minix のプログラムは教科書としても使える簡明さを備えているが、通信プロトコルのプログラム類は複雑であり、残念ながら簡明とは言えない。OS 構造論の観点からは、通信プロトコル処理プログラムは、x- Kernel のプロトコルスタックを移植したかったか、人手不足であるため、Minix のソースコードを移植して実現した。

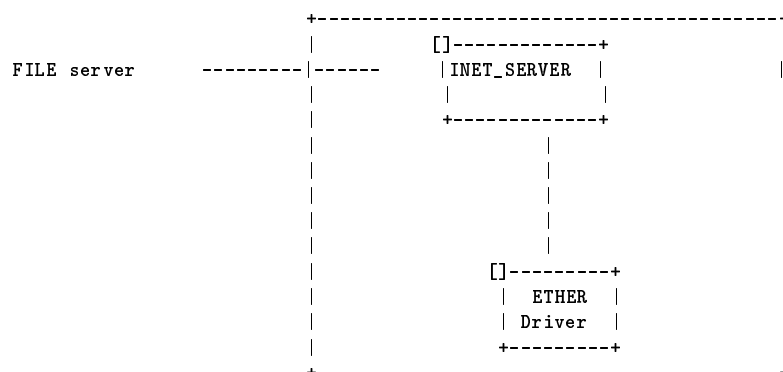


図 9.1: INET サーバーの構成