LP49の内部構造

H_2O

平成 21 年 6 月 30 日

目 次

第 I 部　LP49 の概要 4

第 1 章　LP49 開発の目的 5

第 2 章　プログラム構成 7

第 3 章　サービスと抽象ファイル 9
- 3.1　サービス部品：　サーバントとサーバ . . . 9
- 3.2　抽象ファイルオブジェクト . . . 10
- 3.3　サーバント、サーバの名前空間の接続 . . . 10

第 4 章　Plan9 と LP49 の主要な対比 11

第 II 部　HVM 階層 13

第 5 章　LP49 の立ち上げ 14

第 6 章　Pager: ページフォルトの処理 16

第 III 部　CORE 階層 19

第 7 章　CORE 層の基本機能 20
- 7.1　マルチスレッドサーバ . . . 20
- 7.2　プロセス管理のシステムコール処理 . . . 20
- 7.3　ファイル系システムコール . . . 20
- 7.4　名前空間の管理 . . . 21
- 7.5　サーバント . . . 22
- 7.6　サーバとの連携の仕組み . . . 22
- 7.7　IP プロトコルスタック . . . 23

第 8 章　システムコールの受付と実行 syssrv-l4.c 25
- 8.1　マルチスレッドサーバの実現法 . . . 25
- 8.2　システムコールのパラメータ、データ引き継ぎ . . . 26

第 9 章　プロセス管理 27
- 9.1　プロセス管理テーブル Proc (src/9/port/portdat.h) . . . 27
- 9.2　プロセス管理プログラム Proc (proc-l4.c, sysproc-l4.c in src/9/port/) . . . 27

第 10 章 抽象ファイル操作とサーバント 28
10.1 オブジェクト指向としてのとらえ方 . . . 28
10.2 Dev テーブル Dev (src/9/port/portdat.h) . . . 29
10.3 オブジェクトハンドラ (Chan テーブル) . . . 29
10.4 サーバントの一般構造 . . . 30
10.5 Servant はファイルシステムである . . . 33
10.6 代表的なサーバント . . . 34

第 11 章 名前空間の管理 35
11.1 名前サーチ：パス名から目的オブジェクトを求める . . . 35
11.2 チャネルプログラム chan.c . . . 35
11.3 マウントの仕組み . . . 36

第 12 章 サーバ接続とマウントサーバント 38
12.1 9P: サーバとの通信プロトコル . . . 38
12.2 サーバリンクとサーバ登録サーバント /srv . . . 39
12.3 マウントサーバント devmnt.c の内容 . . . 40
12.4 サーバ記録簿サーバント devsrv.c の内容 . . . 43
12.4.1 サーバアクセス処理の具体例 . . . 44
12.5 リモートサーバの接続法 . . . 45

第 13 章 サーバ接続の管理 46
13.1 永続的サーバの立ち上げ時の処理 . . . 46
13.2 オンデマンドサーバの立ち上げ時の処理 . . . 48
13.3 サーバ要求の待ち受け . . . 49
13.4 動的なサーバ接続要求 . . . 50

第 14 章 デバイスドライバ 51

第 15 章 物理メモリアクセス 53

第 16 章 例外処理 54

第 IV 部 ライブラリー 55

第 17 章 ライブラリー 56
17.1 各ライブラリの概要 . . . 56

第 V 部 Init プロセスとシェル 58

第 18 章 init プロセス 59

第 19 章 qsh: デバッグ用簡易シェル 60

第 20 章 rc: Plan9 の高機能シェル 61

第 VI 部　サーバ 62

第 21 章 代表的なサーバー 63

第 22 章 u9fs サーバ: Linux のファイルシステムをマウントする 65

第 23 章 サーバプログラムの作り方 66

23.1 Ramfs2 サーバのソースコード： src/cmd/lesson/ramfs2.c 66
23.2 ramfs2 の起動と利用 68

第 24 章 Server のための基本技術 70

24.1 多重処理の実現 70
24.2 多重処理の実現法 70
24.3 サーバのクライアントへの見せ方 (その 1 パイプ) 71
24.4 サーバのクライアントへの見せ方 (その 2 TCP 接続) 71

第 VII 部　今後の課題 72

第 25 章 今後の課題 73

25.1 進行中 73
25.2 Wish リスト 73

第I部

LP49の概要

第1章　LP49開発の目的

OS はソフトウェアシステムのベースであり、その適不適はシステムの性能 (機能・効率・信頼性) ばかりでなく、システムの開発コスト (開発期間・拡張性など) を大幅に左右する。組込みシステムを始めとする制御システムやサーバシステムは、適用分野ごとに要求条件が異なる、高い信頼性が要求される、多様なプログラムの短期間開発が要求される、などの要求のために、いわゆる汎用 OS では不十分である。

また、OS 技術はソフトウェア技術の根幹であり重要な研究テーマである。しかし、汎用 OS の世界は既存プログラムの活用が第一優先されるので、新規 OS を開発しても世間に受け入れられない。このためにかつては唯一 IBM を追従しえた日本の大手ベンダーも、OS の研究開発からは手を引いてしまった。ところが、組込みシステム用 OS の場合は、前記要求に答えられれば既存 API から脱却することが可能である。

一般に技術革新は専用解から生まれ、それが一般化されて汎用解として普及していく。制御システム・サーバ用 OS は、この意味からも基盤ソフトウェア技術の優れた研究ターゲットである。また、ソフトウェア技術は、優れた研究成果を踏み台として、かつ文献資料だけではなくソースコードを通して進歩してきた。本試作システムは、L4 マイクロカーネル (独 Karlsruhe 大学) と Plan9 (米 Bell 研) に敬意を表して LP49 と呼んでいる。

本研究は、以下の目的をもって行っている。

制御システム (組込みシステム) やサーバに適したシンプルなＯＳ 汎用 OS は巨大化・複雑化しているが (Ex. Windows-vista: 5000 万行、Redhat 7.1 Linux: 3000 万行)、制御システムやサーバに必要な機能は、その極一部でしかない。制御用、サーバ用に必要充分な機能をもったコンパクトな OS の意義は高い。

障害に対する頑強性の強化 → マイクロカーネル＋マルチサーバ構成 例えば電話交換システムは無停止運転が要求され、許容停止時間を 20 年間に 1 時間である。このように、高い信頼性が要求される。現在使われている大部分の OS はモノリシック型であるが、モノリシック OS では、部分障害 (例えばドライバのバグ) でもシステムクラッシュを導きがちである。障害が生じても、波及範囲を閉じ込めて部分再開を可能とするためには、マイクロカーネル＋マルチサーバ構成の OS が有利であるが, まだその技術が確立されたとはいえない。本研究では、以下の技術の確立を狙う。

- マイクロカーネル以外は、全てユーザモード（プロセッサの非特権命令）で実行させる。
- 名前空間管理、ファイルサービスなどのいわゆる OS サービスは、個別のユーザモードプロセスとして実現し、個別再開を可能とする。
- デバイスドライバ (OS クラッシュの原因の 7 割はドライバと言われる) も、ユーザモードで動作させる。

拡張性の強化 組込み系を始めとする制御システムは適用分野毎に要求機能が異なるので、要求に応じて機能の拡張や余分機能の削減を用意に実現できることが望まれる。本研究では、機能追加は、ユーザモードプロセスの追加などにて、容易に行えるようにする。

システム連携の機能 例えば自動車では数十台の CPU が、連携しながら動作している。今後の組込みシステムは、益々多様な連携動作が要求される。この連携の仕組みこそが、これからの勝負所である。従来から、分散処理は基盤ソフトウェアの重要な課題で多面的に研究されてきたが、現在実用化されているものはまだまだ融通性や機能が不十分である。OS が提供する分散処理で普及しているきものは、分散ファイルシステム (例えば、 NFS ファイルシステム、Andrew ファイルシステム) 位である。また、分散処理ミドルウェアの

CORBA は、仕様が巨大化して使いにくく、融通性が不十分である。JAVA-RMI も、リモートプロシージャコールが出来るだけである。

これからの連携処理機能としては、分散リソースの体系的な管理、一纏まりの関連リソースの export/import、環境変数を含む動作環境の連携、群処理など、従来の分散 OS からは飛躍した機能が要求されよう。

UNIX の次にくる OS として Bell 研で研究開発された OS Plan9 は、新しい視点を提示しており、制御システム用連携処理としても、大いに参考になる。

プログラム開発の容易化 現在までの大部分の制御プログラムや組込みプログラムは、性能を上げるため、並びにハードウェア制御などを行うために、カーネルモードプログラムであった。カーネルモードプログラムは、記述に高いスキルやノウハウが必要なうえ、デバッグが大変難しい。その上, プログラムバグがあると、システム全体がクラッシュしがちである。本研究では、マイクロカーネル以外は全てユーザモードプログラムとし、ほとんどのサービス機能はユーザモードプロセスの追加で実現でき、デバインスドライバもユーザモード化する。これにより、大幅にプログラム開発が容易化・効率化される。

実時間性能の維持 汎用 OS においてモノリシック構成が主流なのは、マイクロカーネル型 OS に比して効率が良かったからである。モノリシック OS のシステムコールはトラップで実装できるのに対し、マイクロカーネル構成のそれはメッセージ通信となる。トラップの方がメッセージ通信よりも高速だからである。しかし、最近のマイクロカーネル技術の進歩により、その差異は縮まっている。本研究で採用している L4 マイクロカーネルは、大変に高速である。その上、プロセッサ性能が非常に向上してきている。システムの総合的オーバヘッドがモノリシック型に比べて 1 割以下ならば、マイクロカーネル型の頑強性・拡張性とプログラム開発の容易化の利点が活きる。

遠くを見るための巨人の肩 → L4 マイクロカーネルと Plan9 OS のソースの活用 OS 全体をスクラッチから作るには、膨大の工数を要する。多くの OS 研究が研究カーネルを作った段階で息切れしてしまっている。そこでできる限り L4 マイクロカーネルと Plan9 のソースコードを活用することとした。ドイツ Karlsruhe 大学の L4 マイクロカーネルは、本研究がカーネルに要求する機能をほぼ揃えている、性能が非常に優れている、コンパクトである、オープンソースであるといった特長をもつ。

また、Bell 研で研究開発された Plan9 は、融通性の高い分散処理を実現、適切なコンポーネント化という特長を持っている。また、幸いなことに現在はオープンソースコードになっている。

ソースコードの公開 ソフトウェアの研究には、ソースコードの公開が有効である。コンパクトで自在に修正などが可能な組込み用 OS を提供する。OS を学習したり、手直ししてみたい人にコンパクトなソースを提供する。GNU 環境にて Plan9 流プログラムの開発を経験できる。

第2章　プログラム構成

LP49 のプログラム構成を図 2.1 に示す。ソースコードは、WEB サイト http://research.nii.ac.jp/H2O/LP49 にて公開している。

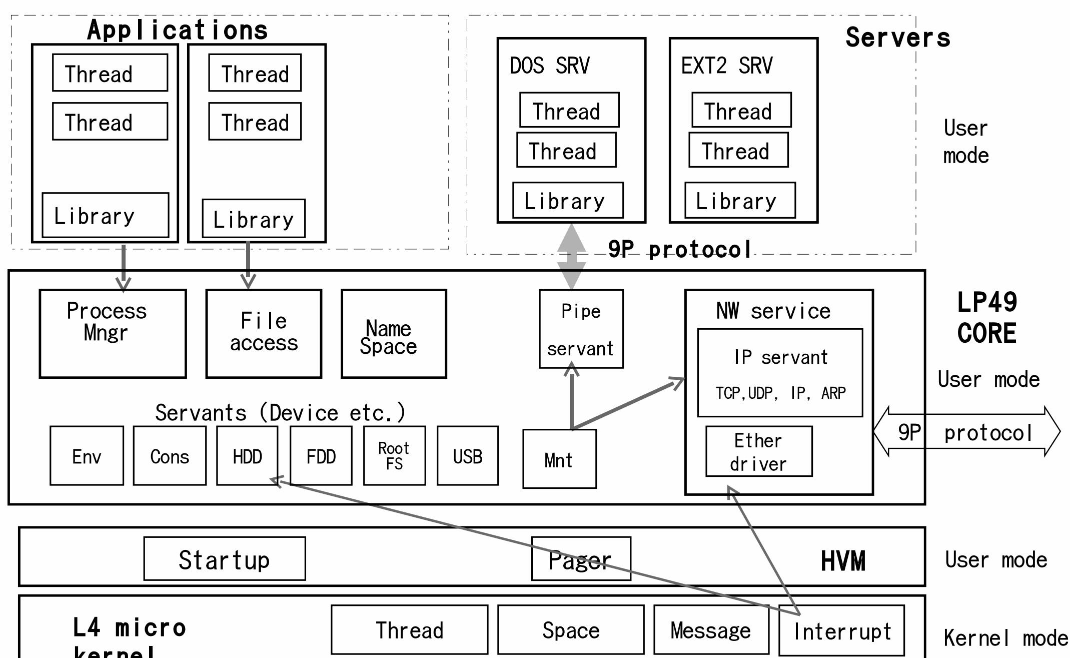


図 2.1: LP49 全体構成

LP49 は、以下の階層からできている。

マイクロカーネル階層 (カーネルモード) L4 マイクロカーネルそのものであり、ここだけがカーネルモードで動作している。マルチスレッド、タスク (=プロセス) 論理空間、スレッド間通信、ページマップの諸機能を提供している。

HVM 階層 (ユーザモードプロセス) これはユーザモードで走る特別タスク (L4 が管理するプロセス）である。LP49 の各プロセスの立ち上げ、スレッド制御と論理空間制御、並びにページャースレッド (`Pager`) の機能を持っている。(L4 マイクロカーネルでは、安全性を上げるために`L4_ThreadControl()` と `L4_SpaceControl()` は、L4 が最初に生成したプロセスにだけ許されている。)

ここに `Pager` は、スレッド実行中にページフォールトが生ずると起動され、ページ割り付けを行うユーザ

モードのスレッドである。各スレッドは、生成時に`Pager`を指定しておく。スレッドの実行中にページフォールトが生ずると、L4 マイクロカーネルは登録された Pager スレッドに Page Fault message を送る。そこで`Pager`は、適切なページを割り当てる。

`Pager`をユーザが記述でき、また Page のネストもできるので、フレキシブルなメモリ管理を行うことができる。

HVM は、将来 Hypervisor Monitor 階層として Virtual Machine に発展させることを考えて付けた名前であるが、現時点ではその機能は持っていない。

CORE 階層/CORE タスク (ユーザモードププロセス) Plan9 のカーネル相当の機能を提供している、ユーザレベルプロセスである。つまり、プロセッサの非特権モードで走るので、安全性が高い。システムコール (実際にはメッセージ) を受けて、所望の処理を行う。デバイスドライバもここに含まれる。

(将来的には、複数のユーザレベルプロセスに分割する可能性がある)

サービス階層 (ユーザモードプロセス) OS サービスを行うファイルサーバー類も普通の応用プログラムも、同等のユーザモードプロセスである。ファイルサーバは、後述の様に 9P プロトコルを喋れる点がちがうだけである。

第3章　サービスと抽象ファイル

3.1　サービス部品：　サーバントとサーバ

LP49 が提供するサービスには、DOS ファイルサービス、EXT2 ファイルサービスのような（どちらかというと）高位サービルと、ハードウェアデバイス制御、リソース制御、環境変数記憶、サービス記録簿といった低位サービスとがある。LP49 では、前者は独立したプロセスによって提供され、後者は CORE 層内のプログラムによって提供される。両者を識別する名前があったほうが便利なので、ここでは前者を「サーバ」、後者を「サーバント」と呼ぶことにする。

サーバもサーバントも、内部要素にアクセスするための「部分名前空間」(directryr tree) を持っており、クライアントはそれを自分の名前空間にマウントすることにより、普通のファイルインタフェースで目的要素にアクセスして、内容の読み書き・制御を行える。

(1) サーバント

サーバントはハードウェアデバイスなど低位サービスを提供するための機能であり、Plan9 のカーネルデバイスファイルシステム (Kernel device file systems) に相当する。但し、対象がハードウェアデバイスだけではなく、サーバ登録簿、環境変数記憶、プロトコルスタックなどを含むので「サーバント」と呼んでいる。

サーバントは、CORE 階層内に存在し、ローカルプロシージャコールされる。(将来はスレッドを持たせてメッセージインタフェースにするかもしれない。)

サーバントプログラムを識別するための名前として『'#'+英字』の形式の名前を持つ。以下に代表的なサーバントとその名前を示す。

```
#c  コンソール
#M  サーバマウントプログラム
#S  ハードディスク
#f  フロッピーデバイス
#e  環境変数（env）
#s  サービス記録簿（service registry）
#l（小文字エル） Ether ドライバ
#I（大文字アイ） プロトコルスタック
#|（縦バー）    Pipe
#R  Root ファイルシステム  (Plan9 の #R とは異なる)
#U  USB ホストコントローラ
#v  VGA コントローラ
...
```

(2) サーバ

DOS ファイルシステム, EXT2 ファイルシステムなど高位サービスを提供するサーバであり、各サービスごとに独立したユーザモードのプロセスである。クライアントプロセスとは、9P プロトコルを使って会話する。つまりメッセージインタフェースである。つまりサーバとは、9P プロトコルを喋れるユーザプログラムにすぎない。9P メッセージは、サーバリンクと呼ばれる TCP 接続あるいは pipe を通して運ばれる。各サービスのサーバリンクはサービス "記録簿 (/srv/*)" に登録される。クライアントは、サーバ登録簿から目的のサーバを見つけて、これを自分の名前空間にマウントすることで、サーバの持つ名前空間に普通のファイルインタフェースでアクセスできるようになる。

同一のサービスをサーバとして実装することも、サーバントとして実装することも可能である。プログラムを作る際の参考に、RAM ファイルサービスについて、サーバによる実装を src/9/cmd/simple/ramfs.c、サーバントとしての実装を src/9/port/devrootfs.c として載せてある。

3.2 抽象ファイルオブジェクト

LP49/Plan9 では、ほとんど全てのリソース（実ファイルからネットワーク接続まで）を "抽象ファイル" として扱っている。つまり、create(), open(), write(), read(), stat() ,,, により、統一的に操作できる。

○ サーバント、サーバに収容されている。

○ オブジェクト

○ (To be described.)

3.3 サーバント、サーバの名前空間の接続

サーバントもサーバも自分の名前空間 (directory tree) を有しており、抽象ファイル（リソース）を収容している。

サーバントやサーバの名前空間を、ユーザプロセスの名前空間に接続 (bind, mount) することで、ユーザプロセスから使えるようになる。bind する際にアクセス権限チェックを行う (未実装)。

【サーバントの名前空間への接続例】

```
Ex.   bind -a #f  /dev
          ← フロッピーが /dev/fd0disk, /dev/fd0ctl として見える
      bind -a #S  /dev
          ←ハードディスクが /dev/sdC0/... として見える
      bind -a #l  /net
          ← プロトコルスタックインタフェースが/net の下に見える。
```

第4章　Plan9 と LP49 の主要な対比

表 4.1: LP49 と Plan9 の対比

分類	Plan 9	LP49
Micro kernel	No	Yes
並行処理	プロセスはコルーチン、 スレッドはメモリ域共用のプロセス	L4 Process L4 Thread
システムコール	Trap カーネル＋ユーザプロセス	L4 メッセージ マルチスレッドサーバ
〃データ入力 〃データ出力	Plan9 カーネルが APL 空間を直接アクセス	L4 ページマップ L4 ページマップ
ドライバ	カーネルモード	ユーザモード
言語仕様	Plan9 独自の C 無名フィールド, 無名パラメータ typedef, USED(), SET() #pragma, 自動ライブラリリンク	GCC
コンパイラ	Plan9 独自 C コンパイラ	GCC
Utility	Plan9 の Linker, Assembler, mk	GCC, gld, gmake
Binary	a.out 形式	ELF 形式

LP49 の試作に当たっては、Plan9 のソースコードのうち有効利用できるものは活用したが、事はそう簡単ではなく、かなりのパワーを要した。その主要な事項を以下に記す。

コルーチンモデルと L4 スレッドの違い 並列処理の実現法が、Plan9 はコルーチンモデルであるのに対し、LP49 は L4 マルチスレッドモデルであるので、細部でいろいろと対処せざるを得なく、相当量の修正が必要となった。

言語仕様、開発環境の違い Plan9 の C 言語は ANSI-C ではなく、無名フィールド、無名パラメータなどの独自の拡張を行った独特の C である。プログラム開発環境も Plan9 上の独自環境である。我々は、L4 マイクロカーネルをコンパイルでき、使い慣れた GNU 環境を使いたい。そこで、GCC のフロントエンドを修正して Plan9 の C 言語をサポートすることも考えたが、GCC の頻繁な version up に追随するコストも考慮して、Plan9 のソースを人手で Plan9-C 言語から GNU-C 言語に変換する方を選んだ。

システムコールの実装法 Plan9 のシステムコールは、モノリシック OS と同様なトラップで実現している。つまり、ユーザモードのクライアントプロセス はトラップを起こしてカーネルモードに入り、カーネル内での処理待ち中断が比較的簡単に実現できる。

これに対し、LP49 ではクライアントプロセスとサーバプロセスはメッセージでやり取りする全く別のプロセスであり、あるクライアントプロセスの仕事でサーバプロセスが待ち合わせると、他のクライアントの要求を受け入れることができなくなってしまう。この対処には、Minix のような特殊技巧 (suspend-resume) を使ったり、マルチスレッドサーバとする必要がある。LP49 では、マルチスレッドサーバを実装した。このための Plan9 ソースの修正は、かなりを要した。

継ぎ足しプログラムの問題 Plan9 カーネルプログラムの要であるチャネルコード (chan.c), デバイスコード (dev.c) などは、最初の版から継ぎ足しで拡張されてきたので、非常に分かりにくいプログラムになっている。学習用 OS をも目指している LP49 にとってこれは問題で、いつか時間をとって書き直しをしたいと思っている。

簡明なプロトコルスタック プロトコルスタックは、Stream モジュール型 → X-kernel 型 → 現在の queue 接続型と、3 回にわたって全面的に作り直されているので、簡明なロジックになっている。学習用 OS として使える。

グラフィックユーザインタフェース、VGA 周り これから着手。

第II部

HVM階層

第5章　LP49の立ち上げ

HVM 階層は（L4 マイクロカーネルから見ると）ユーザモードプロセスであり、以下の機能を実現している。(将来、CORE 階層のプロセス管理機能は、HVM に移す可能性がある。)

(1) LP49 の立ち上げ

(2) スレッドの生成、タスクの生成 L4 マイクロカーネルの `L4_ThreadControl()`, `L4_SpaceControl()` を使って、`CORE` 層、`QSH shell`, `dossrv` の各プロセスを生成する。

(3) Pager スレッド (ページャースレッド)（`src/9/pc/hvm/mx-pager.c`） プログラムの実行中にページフォールトが生じると、L4 マイクロカーネルは本 Pager スレッドに page fault メッセージを送る。本 Pager スレッドは、適切なページを選択して page fault したアドレスにマッピングする。現 `Pager` は仮実装のためチェック機能が弱くプログラム構造も汚い。ページャー実装は L4 マイクロカーネルの利用において最も興味深い点の一つであるので、将来全面的に作り直して以下を実現する予定である。

- 保護・セキュリティの強化
- Copy-on-write
- Page cache
- プロセス管理ページの分離

LP49 の立ち上げは、以下のように進む。

(1) Grub

LP49 のブートローダは　GNU の Grub である。ブートディスクの “boot/grub/menu.lst” ファイルの内容を以下に示す。

【Boot-CD の boot/grub/menu.lst ファイルの内容】

```
#######
title = LP49: CD BOOT (Hvm + Pc + Init + DosSrv + 9660Srv)
kernel=/l4/kickstart-0804.gz
module=/l4/l4ka-0804.gz
module=/l4/sigma0-0804.gz
module=/boot/hvm.gz
module=/boot/pc.gz
module=/boot/init.gz
module=/boot/dossrv.gz
module=/bin/9660srv
```

この内容にしたがい、Grub は以下を行う。

1. `kernel` 及び `module` にてて指定されたファイルを (`.gz` が付いている場合は unzip して) 順番に物理メモリにコピーする。
2. `kernel` として指定された `kickstart` の開始番地に制御を渡して起動する。

(2) L4 の kickstart

`kickstart` は、L4 マイクロカーネルの startup プログラムであり、GRUB によって物理メモリ上にコピーされたロードモジュールを用いて以下の処理を行う。

1. L4 マイクロカーネル本体 `l4ka` を立ち上げる。
2. L4 の元締め Pager `sigma0` を立ち上げる。
3. LP49 の `hvm` モジュールを、L4 マイクロカーネルのプロセスとして立ち上げる。

(3) LP49 の hvm の立ち上げ処理

LP49 の `hvm` は、以下の順番で LP49 を立ち上げる。

1. `sigma0` から LP49 が使用するメモリページをもらってメモリ管理を初期設定する。
2. LP49 の Pager である mx-pager スレッドを生成・起動する。
3. GRUB がメモリー上にコピーしたロードモジューロを用いて、以下の順に各プロセスを生成・起動する

pc LP49core プロセス。

init `Init` プロセス。シェルプロセスは、ここから `fork()` される。

dossrv DOS ファイルはすぐにつかうので、ここで DOS ファイルサーバを立ち上げておく。

9660srv CD の ISO ファイルサーバである。すぐに CD を読む必要があるので、ここで 9660 ファイルサーバを立ち上げておく。

第6章　Pager: ページフォルトの処理

L4 では、各スレッドは生成時に pager を指定する。スレッド実行中にページフォールトが生じると、指定された `pager` に page fault message を送る。この際の page fault の処理を行うのが本 `pager` である。ソースコードは、“`src/9/hvm/mx-pager.c` である。

また、L4 は安全のためにスレッドの生成やメモリ空間制御は L4 から直接起動されたプロセスだけが行えるようにしている。従って、スレッド生成とメモリ空間制御も `pager` が担っている。

`Pager` は、他のプロセス内でページフォールとが生じたときに、以下に述べる手順で各要求を処理する。

1. `L4-sigma0` から利用可能な全メモリページをもらい、それらを `hvm` の論理空間の 2 Giga 番地以降に各メモリページを『論理アドレス＝物理アドレス+ 2 Giga』になるようにマップする。

   (2 Giga 番地以降へのフラットマッピングは仮手法であり、将来作り直す予定。)

2. 要求メッセージが来るのを待って、所定の処理を行う。ここに、要求メッセージとしては、ページフォールト、メモリ空間制御、スレッド制御などがある。

現在のページャは実験用に短時間で書いたものであり、内容的に不十分であり、プログラム構造も汚れている。プロセスが使えるメモリ域のチェックも, 0 ページアクセスのチェックも行っていない。同一プログラムプロセス間でのコードセグメントの共用も行っていないし、コピーオンライトもまだ実装していない。ページャは、今まで蓄積された経験に基づいて全面的に作り直す予定である。L4 マイクロカーネルの利用において、ページャはもっとも工夫のしがいのある面白い機能である。また、もっとも神経を使うプログラムでもある。

`Pager` スレッドの処理概要を以下に示す。

```
void mxpager_thread(void)
{
    L4_ThreadId_t  src;
    L4_Word_t      pfadrs, pf_page, ip;
    L4_Fpage_t     fpage;
    L4_MapItem_t   map;
    L4_Msg_t       _MRs;
    L4_MsgTag_t    tag;
    :

    tag = L4_Wait(& src);  // 最初のメッセージを受ける。
    targettask = TID2PROCNR(src);

    while (1) {
        L4_MsgStore(tag, &_MRs);
        label = L4_MsgLabel(&_MRs);

        if (label == THREAD_CONTROL)  { //スレッド制御要求の場合
            :
            rc = L4_ThreadControl(dest, space, sched, pager, utcbloc);
            :
            tag = L4_ReplyWait(src,  &src);
            continue;
        }
        else if (label == SPACE_CONTROL)  { //論理空間制御要求の場合
            :
            rc = L4_SpaceControl(space, control, kipFpage, utcbFpage, redirector, &oldcntl);
            :
            tag = L4_ReplyWait(src,  &src);
```

```
    continue;
}
else if (label == ASSOCIATE_INTR) { //割り込みハンドラ登録の場合
    :
    rc = L4_AssociateInterrupt(intrThread, intrHandler);
    :
    tag = L4_ReplyWait(src,  &src);
    continue;
}
else if (label == DEASSOCIATE_INTR)  {
    :
    rc = L4_DeassociateInterrupt(intrThread);
    :
    tag = L4_ReplyWait(src,  &src);
    continue;
}
else if (label == PROC2PROC_COPY) {
    :
    proc2proc_copy(fromproc, fromadrs, toproc, toadrs, size);
    :
    tag = L4_ReplyWait(src,  &src);
    continue;
}
else if (label == PROC_MEMSET)   {
    :
    rc = proc_memset(procnr, adrs, value, size);
    :
    tag = L4_ReplyWait(src,  &src);
    continue;
}
else if (label == MM_FORK)   {
    :
    rc = mm_fork(parent_tid, child_tid, th_max, pager, ip, sp);
    :
    tag = L4_ReplyWait(src,  &src);
    continue;
}
//-----------------------------------------------------------
else if (label == MM_FREESPACE)   {
    :
    rc = free_space(procnr);
    :
    tag = L4_ReplyWait(src,  &src);
    continue;
}
else if (label == ALLOC_DMA)
{
    略
    continue;
}
else if (label == PHYS_MEM_ALLOC)
{
    略
    continue;
}
//============== Page fault ==============================
else {  // ページフォールトの場合はここにくる
    :
    pfadrs = L4_MsgWord(&_MRs, 0);
       //pfadrs: ページフォールトの番地
    ip = L4_MsgWord(&_MRs, 1);
      //ip: 命令番地
    client_task = TID2PROCNR(src);
    pf_page = pfadrs & 0xFFFFF000;
    targetpage = (unsigned)get_targetpage(client_task, pfadrs, ip);
        // map するための適切なページを求める
    :
    fpage = L4_Fpage(targetpage, 4096);
    L4_Set_Rights(&fpage, L4_FullyAccessible);
    map = L4_MapItem(fpage,  pf_page);
         // map: ページマップ情報を編集
    L4_MsgClear(&_MRs);
```

```
            L4_MsgAppendMapItem(&_MRs, map);
            L4_MsgLoad(&_MRs);
            tag = L4_ReplyWait(src,  &src);
                // ReplyWait: 返答を返し、次のメッセージを待つ.
                //     返答時にページがマップされる.
        }
    }
}
```

第III部

CORE階層

第7章　CORE層の基本機能

“LP49core” は、Plan9 カーネル相当の機能を提供する”ユーザモード”で走る L4 プロセスである。

LP49core は、APL プロセスからシステムコールメッセージを受けて処理するので、この観点からは APL プロセスがクライアント、LP49core がサーバとして働く。サーバプロセスがは発行するシステムコールも同様である。また、LP49core は 9P メッセージをサーバプロセスに送って処理をいらいするので、この観点からは LP49core がクライアント、サーバプロセスがサーバとして働く。

7.1　マルチスレッドサーバ

モノリシック OS (Plan9 も含めて）では、クライアントプロセスがシステムコールを実行すると、トラップが生じて、カーネルモードになり、カーネルがクライアント空間を直接アクセスできる。

これに対し、LP49core はユーザモードで実行される L4 のプロセスであり、クライアントプロセスとは独立した論理メモリ空間を持ち、システムコールはメッセージ通信によって実現される。

そこで、LP49core では、マルチスレッドサーバを導入している。

- クライアントからのシステムコール (L4 メッセージで運ばれる) を受け付ける。
- システムコールの処理中には、中断が生じうる。中断が生じた場合には、別のクライアントプロセスのシステムコールを受け入れるために、“マルチスレッドサーバー” としている。
- このために、CORE プロセスはシステムコールメッセージを受けとると、スレッドプールから空きスレッドを取り出して、そのスレッドに処理を任せる。

7.2　プロセス管理のシステムコール処理

プロセス管理のシステムコールとしては、`fork()`，`spawn()`，`exec()`，`exits()` 等がある。ここに、`spawn()` は `fork()` と `exec()` を合わせたものである。

LP49 のプロセスは、L4 論理空間と L4 スレッドから構築される。従って プロセス管理システムコールの要は、L4 の `L4_ThreadControl()` 及び`L4_SpaceControl()` 実行である。

現行（新版）の L4 は、security 強化の観点から`L4_ThreadControl()` 及び`L4_SpaceControl()` は L4 が直接起動したプロセス、つまり `HVM` の中で呼び出すことができない。そこで、`CORE` は `HVM` にプロセス管理要求メッセージを送り、`HVM` が L4 を呼び出すようにしている。

7.3　ファイル系システムコール

Plan9/LP49 では、ほとんど全てのリソースは “ファイル” として抽象化されている。（本資料の大部分では、“リソース” と “抽象ファイルオブジェクト” とを同じ意味で使っている。）つまり、各リソースは “名前空間” (Directory tree) からたどることが出来、`create()`，`open()`，`read()`，`write()`，`close()`，`stat(()`，`wstat()` 等の使い慣れたシステムコールで処理を行える。

サーバントならびにサーバは、各々自分の名前空間を持っており、そこに各抽象ファイルオブジェクトがつながっている。

7.4 名前空間の管理

(1) 名前空間の接続

プロセスは、名前空間の directory tree をたどることで各リソースにアクセスできる。プロセスがアクセスできる名前空間を、“プロセスの名前空間” と呼ぶことにする。

Unix 同様に、名前空間のルートとなるのが “ルートファイルシステム” である。ルートファイルシステムはサーバント (#R) であって、定形のマウントポイントを含む Directory tree である。ルート FS サーバントのソースコードは、“`src/9/port/devrootfs.c`” である

サーバントやサーバの名前空間をプロセスの名前空間に接続することにより、プロセスはサーバントやサーバのリソースにアクセスすることが可能になる。逆にいえば、プロセスの名前空間に接続されていないサーバントやサーバは、プロセスには見えないので、アクセス権制御の役目も果たす。（厳密にいえば、特定のプロセスには、#S, #s などのサーバント名によるアクセスを許可することもできる。）

つまり、名前空間を接続する際に厳密なアクセス権チェックを行うことで、アクセスできる名前を厳密に制御することができる。

名前空間の接続の模様を、図 7.1 に示す。この図において、一番上の三角はルートファイルシステム (#R) の名前空間を表す。記憶メディア (#S), コンソール (#c), USB (#U) などのサーバントは、`/dev` に接続されている。IP サーバント (#I) と Ether サーバント (#l) は、`/net` に接続されている。

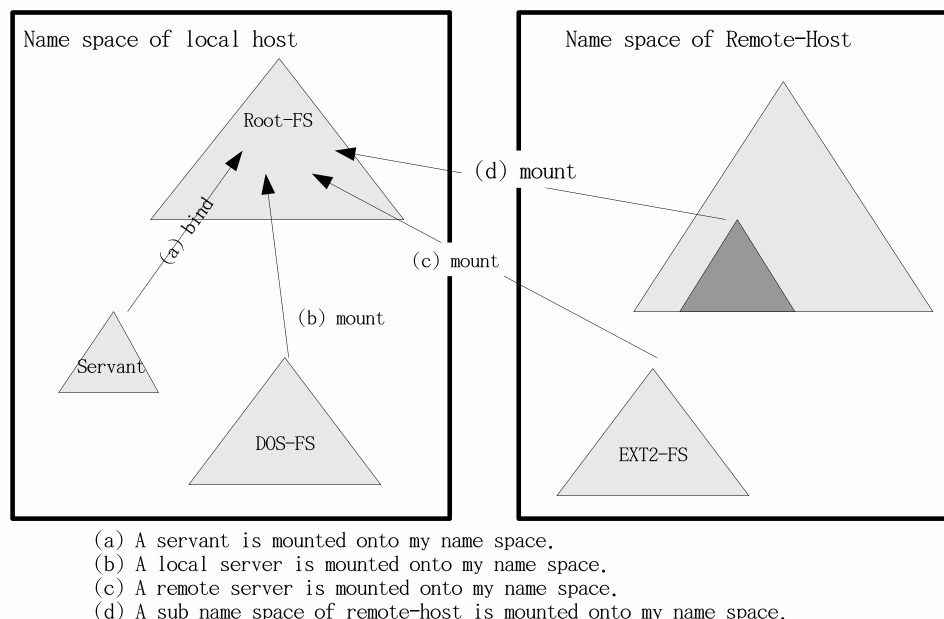


図 7.1: 名前空間とマウント

サーバ登録サーバント#s) は、サーバリンク (pipe あるいは TCP 接) を登録するデータベースである。各ファ

イルサーバは、“/srv/サービス名” というファイルを生成して、自分のサーバリンクをそこに書き込んでおく。例えば DOS ファイルサーバは /srv/dos, EXT2 ファイルサーバは/srv/ext2、CD(ISO9660) ファイルサーバは /srv/9660 といった具合である。

たとえば CD の全体 (/dev/sdD0/data) を, プロセス名前空間の “/t” に接続するコマンドは、以下のとおりである。

```
LP49[/]:  mount  -a  /srv/9660  /t  /dev/sdD0/data
```

(2) プロセス個別名前空間

Plan9/LP49 では、個々のプロセス毎に自分独自の名前空間 (Directory tree) を持つことができる。これは rfork(int flags) 命令のフラッグに RFNAMEG, RFNAMEG 指定の有無による。

RFNAMEG 指定の場合 親プロセスの名前空間のコピーをつくる。

RFCNAMEG 指定の場合 名前空間をクリアーする。

どちらも指定しない場合 親プロセスと名前空間を共有する。つまり UNIX の fork() と同じ。

また、同一の directory に複数の directory をマウント・バインドすることができる。これを “ユニオンマウント” と呼ぶ。例えば、/bin. /dev, /net には、一般に複数の directory をマウントしている。このユニオンマシントが、プログラム論理を分かりにくくしている原因の一つではある。

(3) その他

LP49/Plan9 では、各プロセスが個別の名前空間 (Directory tree) を持てる。名前空間の修正は, プロセス生成時、並びに bind あるいは mount コマンドによって行える。

UNIX では名前空間のマウントには root 権限が必要であるが、LP49/Plan9 では root 権限は不用である。

バインド サーバントや部分前空間ををマウントポイントに接続する。

マウント サービスサーバをマウントポイントに接続する。

【例】

```
mount -a   サービス a   /tmp  ...
mount -a   サービス b   /tmp  ...
   →  同一の /tmp ディレクトリに、任意個をマウントできる。
   属性:  -a: after, -b: before, -c: create, 無指定: replace
```

7.5 サーバント

7.6 サーバとの連携の仕組み

ファイルサービスなどのいわゆるＯＳサービスは、CORE 階層ではなく、ユーザモードプロセスであるサーバによって提供される。サーバは、9P プロトコルを喋れる普通のユーザモードプロセスにすぎない。

クライアントプロセスがシステムコールを行うと、その相手がサーバであった場合には、9P プロトコルメッセージに変換して、パイプ (同一ノード内の場合) あるいは通信コネクション (別ノードの場合) を介して、目的サーバに届けられる。

サーバは、内部にディレクトリートリーを持っているため、ファイルシステムとして change directory することで各要素をたどれる。

7.7 IP プロトコルスタック

IP プロトコルスタックは連携処理の要であるが、一般にプログラムは複雑でサイズも大きい。Minix ですら、inet 部分は簡明とはいえない。幸いなことに、Plan9 のプロトコルスタック (ソースは src/9/ip/*) は、比較的簡明で拡張性も高い。

IP プロトコル処理スタックのソース規模は約 20K 行と大きいので、サーバとして実装することも考えられるが、現時点では LP49core のサーバントとして組み込んでいる。

IP サービスは、`IP` サーバント (#I) から提供される。また、Ether カード制御は `Ether` サーバント (#l) から提供される。

なお。IP プロトコルスタックの詳細は、別資料 "**LP49 NW 説明書**" を参照されたい。

(1) IP サーバント (#I, devip.c)

プロトコルスタックは IP サーバント (devip) からアクセスでき、以下のトリー構成をもつファイルシステムになっている。個々の論理コネクションもファイルとして表現されており、`clone` ファイルをオープンすることで生成される。例えば `tcp/clone` を `open()` すると、順次論理チャネル `tcp/0`, `tcp/1`, `tcp/2,,,,` が生成される。

モジュール構造としては、`TCP`, `UDP`, `IP`, `ARP`, `ICMP` などといったプログラム毎にモジュール化が行われている。

```
--+- tcp/ ----+- clone
  |           |- stats
  |           |- 0/----+- ctl
  |           |        |- data
  |                    |- local
  |                    |
  +- udp/ ---+- clone
  |          |- stats
  |          |- 0/ ---+- ctl
  |          |        |-data
  |                   |-local
  |                   |
  +- ipifc/ -----+- clone
  |              |- stats
  |              |- 0/--
  |              |
  +-- ndb/ ----
  |
  +-- arp/ ----
```

(2) Ether サーバント (#l, devether.c)

第8章　システムコールの受付と実行 syssrv-l4.c

8.1　マルチスレッドサーバの実現法

LP49core は、プロセスのシステムコールに対してはサーバとして、サーバプロセスに 9p メッセージで仕事を依頼する場合にはクライアントとして機能する。

モノリシック OS では、APL のシステムコールはトラップを使って OS カーネルに飛び込むのに対し、本 OS のシステムコールは、APL から LP49core へのメッセージ通信となる。システムコールの仕組みを、図 8.1 に示す。

(1) ライブラリによる L4 メッセージ化

APL のシステムコールは, 使い慣れた関数呼び出し (`open(...), read(...), write(...),,,`) である。ライブラリは、これを L4 メッセージに変換して LP49core に送り、返答メッセージを待つ。APL と LP49core は別論理空間なので、アドレス引き継ぎは使えない。システムコールの引数は L4 メッセージの値コピー, バッファー域は L4 メッセージのページマップ機能を使って引き継いでいる。

(2) LP49core マルチスレッドサーバ

システムコールの処理は中断が生じうる。複数の処理を並行して処理するために、マルチスレッドサーバ方式を実装している。

要求メッセージは `Mngr` スレッドに送られる (L4 メッセージの宛先はスレッドである)。`Mngr` スレッドは、スレッドプールから空き `clerk` スレッドを割り当てて、それに処理を行わせる。

(3) サーバントアクセスの仕組み

システムコールの対象がサーバントである場合は、図 8.1 に示すように `clerk` スレッドがサーバントの関数を呼び出す。

(4) サーバアクセスの仕組み

システムコールの対象がサーバである場合は、図 8.1 に示すように `clerk` スレッドは **Mnt サーバント** を呼ぶ。`Mnt` サーバント (#M) はサーバをマウントするための仕組みであり、システムコール引数から 9P メッセージを編集して、目的サーバがノード内の場合は Pipe サーバント, 別ノードの場合は TCP コネクションを経由して、Remote Procedure Call を行う。

○　Explanation : to be added

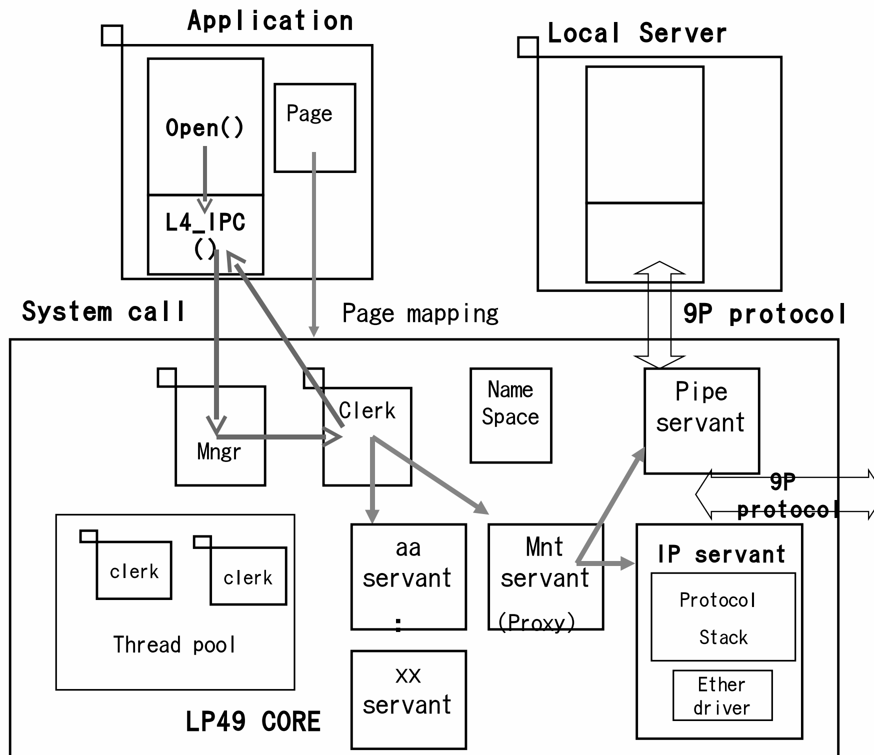


図 8.1: システムコール

8.2 システムコールのパラメータ、データ引き継ぎ

LP49 のシステムコールは、クライアントプロセスから CORE プロセスへのメッセージ通信となる。L4 マイクロカーネルの高速なメッセージ通信が活躍している。データ引き継ぎは、以下のように実装している。

(1) 文字列パラメータの引き継ぎ

Plan9 文字列パラメータは call-by-reference で送り、カーネルがユーザ空間に直接アクセス。

LP49 パラメータは全て call-by-value で送る (L4 の string copy)

(2) バッファデータの引き継ぎ

Plan9 アドレスを引き継いで、カーネルがユーザ空間に直接アクセス。

LP49 該当バッファを含むクライアントページを LP49core にマッピンクして、読み書きさせている。L4 の Page mappink 機能は、Flex page (サイズが 2^n ページで、かつ開始アドレスが 2^n バウンダリでナケレバならない) 単位なので、場合によってはかなりの空間がマップされる可能性がある。また、データサイズが小さい場合は、L4 の string copy を使った方が効率的である。これは簡単な改善課題であるので、試みられたい。

第9章　プロセス管理

9.1　プロセス管理テーブル Proc (src/9/port/portdat.h)

Proc テーブルはプロセス対応に存在しており、LP49 プロセスの管理情報を記録している。HVM 層や CORE 層は L4 のプロセスであるが、LP49 のプロセスではないので、本 Proc テーブルの対象ではない。スケジューリングやコンテクスト切替は本テーブルの役目ではなく L4 マイクロカーネルが行う。現 Proc テーブルは LP49 では不要な Plan9 用フィールドが残っているが、そのうちにソースプログラムから削除する予定である。

Proc テーブル主要フィールドを以下に示す。

```
*------ Proc ---------*
| L4_thread_t  thread | ← L4 スレッドの ID を記憶
|   :                 |
| Proc  *parent       | ← 親プロセスへのポインタ
|   :                 |
| Pgrp  *pgrp         | ← 名前空間へのポインタ（複数プロセスで共用可）
| Egrp  *egrp         | ← 環境変数空間へのポインタ（複数プロセスで共用可）
| Fgrp  *fgrp         | ← ファイル記述子テーブルへのポインタ（複数プロセスで共用可）
|   :                 |
*---------------------*
```

9.2　プロセス管理プログラム Proc (proc-l4.c, sysproc-l4.c in src/9/port/)

(1) proc-l4.c

○　説明：To be added

(2) sysproc-l4.c

○　説明：To be added

第10章　抽象ファイル操作とサーバント

10.1　オブジェクト指向としてのとらえ方

Plan9 と同様、LP49 では殆どのリソースを “ファイル” として抽象化している。個々の “抽象ファイル” は、サーバント内あるいはサーバ内に存在し、名前空間 (directory tree) に登録されている。抽象ファイルは、`create()`, `open()`, `read()`, `write()`,,, 等の統一ファイルインタフェースで操作できるオブジェクトである。

サーバ内のファイルは、サーバをマウントすることによって、LP49core がアクセスできるようになる。サーバのマウントを行うのは**マウントサーバント** (`#M`) である。サーバ内ファイルにアクセスしようとすると、マウントサーバントはそのファイルに対応した**代行 (Proxy) オブジェクト**を割り当てる。代行オブジェクトは操作を受けると、9P メッセージに変換してサーバに処理を依頼する。サーバアクセスに付いては、後で詳しく述べる。

このように LP49core からは、サーバントがファイル操作の要となる。サーバントは、オブジェクト指向の観点からは、図 10.1 に示すように “抽象ファイル” はインスタンスオブジェクト、“サーバント定義” はクラス定義に相当する。各サーバントは同一のインタフェースを有する。

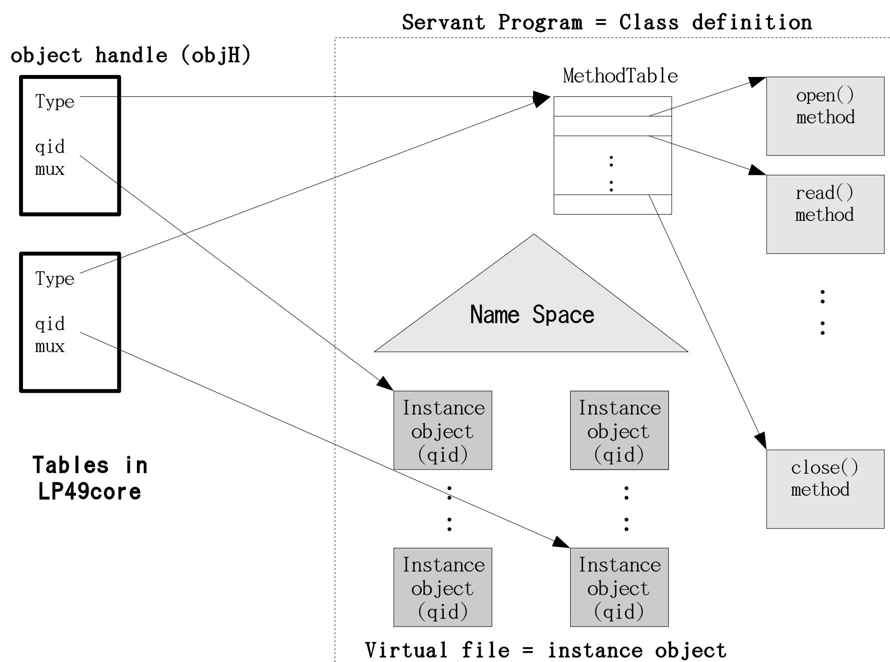


図 10.1: 仮想ファイルのオブジェクトモデル

- 各サーバントプログラムはクラス定義に相当し、attach(), init(), open(), read(), write(),,, 等のメソッドコードを持ち、メソッドテーブルを介して呼ばれる。
- “抽象ファイル” は、インスタンスオブジェクトであり、サーバントの名前空間に登録されている。以下では **VF(抽象ファイル) オブジェクト**と呼ぶ。

- VF オブジェクトは、qid という値によりサーバント内でユニークに識別される。Qid は Unix の inode 番号に相当する。
- 一般のオブジェクト指向言語と違って、同一クラス (サーバント) 内でも VF オブジェクトのデータ構成は同一とは限らない。各メソッドは VF オブジェクトの qid から、そのデータ構成を判定し、対応した処理を行う。

LP49core 内では、VF オブジェクトは**オブジェクトハンドル**を介してアクセスされる。オブジェクトハンドル (`objH`) は、サーバント識別情報 (`type` フィールド), VF オブジェクトの qid (`qid` フィールド), その他の管理情報が載ったテーブルである。**Plan9 のソースコードを流用したので、プログラム上では Chan(nel) タイプのテーブルとなっている。以下 Chan テーブルはオブジェクトハンドルと理解されたい。**LP49core は、オブジェクトハンドルの `type` フィールドからサーバントの各メソッドをアクセスし、`qid` フィールドからインスタンスを決定する。ここでは VF オブジェクト α を指すオブジェクトハンドルを "`objH`$\{\alpha\}$" と表記する。

オブジェクトハンドルは、対象を `open()`, `create()` した時や、change directory された時に割り当てられ、参照カウンタが 0 になったときに消去される。

10.2 Dev テーブル Dev (src/9/port/portdat.h)

"`Dev` テーブル" とは、上で述べたメソッドテーブルであり、サーバントプログラム毎に存在する。

`Dev` 構造体の定義を以下に示す。これがサーバントの統一インタフェースである。

```
【抽象ファイルのインタフェース】
struct Dev
{
    int     dc;    // サーバントを識別する 1 文字。#I の'I' など。
    char*   name;  // サーバントの名前
    void    (*reset)(void);
    void    (*init)(void);
    void    (*shutdown)(void);
    Chan*   (*attach)(char*);
    Walkqid*(*walk)(Chan*, Chan*, char**, int);
    int     (*stat)(Chan*, uchar*, int);
    Chan*   (*open)(Chan*, int);
    void    (*create)(Chan*, char*, int, ulong);
    void    (*close)(Chan*);
    long    (*read)(Chan*, void*, long, vlong);
    Block*  (*bread)(Chan*, long, ulong);
    long    (*write)(Chan*, void*, long, vlong);
    long    (*bwrite)(Chan*, Block*, ulong);
    void    (*remove)(Chan*);
    int     (*wstat)(Chan*, uchar*, int);
    void    (*power)(int);  // power mgt: power(1)
    int     (*config)(int, char*, DevConf*);
};
```

`Dev` テーブルは各メソッドへのリンク表であり、C++言語の仮想関数テーブル `vtbl` に相当する。各サーバントは、`Dev` インタフェースの実装と考えられる。

10.3 オブジェクトハンドラ (Chan テーブル)

前述のように、LP49 では処理対象を「抽象ファイル」、つまり "`Dev` インタフェースを持つオブジェクト" として扱っている。

LP49core 内では、各オブジェクトはオブジェクトハンドル経由でアクセスされる。LP49 は Plan9 のソースコードに修正を行っているので、オブジェクトハンドルは Chan (channel の意) タイプのテーブルである。定義は src/9/port/portdat.h に含まれている。

Chan テーブルは、LP49core が抽象ファイルにアクセスするのに必要な情報を載せたテーブル (抽象ファイルアクセスの把手＝ “ハンドラ”）である。例えばファイルを open( ) すると LP49core 内部では Chan テーブルが用意され、write( ), read( ) などの操作は Chan テーブルを経由して行われる。

Chan テーブルの主要フィールドを以下に示す。

```
+----- Chan ----------+
|     :               |
| vlong  offset       | ← ファイル内オフセット
|    :                |
| ushort  type        | ← デバイスタイプ（Unix の major# に相当）
| ulong   dev         | ← デバイスの番号（Unix の minor# に相当）
| ushort  mode        | ← ファイルアクセスモード
|    :                |
| Qid     qid         | ← ユニークな識別子（Unix の inode 番号に相当）
| int     fid         | ← サーバマウントで多重化識別に使う
|    :                |
| Mhead   *umh        | ← マウントサーチに使う
|    :                |
| Mnt     *mux        | ← サーバマウントの多重化に使う
|    :                |
| Cham   *mchan       | ← サーバリンク
|    :                |
| Path   *path        | ← パス名
+---------------------+
```

図 10.2 に示すように、Chan テーブルはファイルとして抽象化されたオブジェクトを統一的に扱うためのハンドルである。

Chan.type フィールドは、Dev テーブルへのリンク情報になっている。Dev テーブルはサーバント対応に用意されており、このテーブルを見て実行するメソッドを決定するわけである。UNIX でいえば Chan.type フィールドは、major 番号に相当する。

UNIX でいえば, Chan.dev フィールドは minor 番号に。Chan.qid フィールドは inode 番号に相当する。

Unix においてファイルが {Major 番号、Minor 番号, inode 番号 } によってユニークに識別されるように、LP49 においては全抽象ファイルが {type、dev, qid} によってユニークに識別される。

また、処理によっては Chan._u フィールドに、個別データがつながれる場合もある。

read( ), write( ) などの操作は、type フィールド経由で devfloppy.c で定義されている fdread( ), fdwrite( ) などが呼び出される。

例えば /dev/fd0 (フロッピードライバ) を open( ) すると、LP49core は/dev/fd0 を見つけ、それに対するアクセスハンドルとして Chan テーブルを生成し、各フィールドを設定する。Chan.type フィールドには、FD サーバント devfloppy の番号が設定される。

10.4 サーバントの一般構造

サーバントのソースプログラムはソフトウェア部品の好例であり、各々 devxxx.c というファイル名のソースモジュールである。なお dev.c はサーバント間で共通に使われる関数を定義している。

サーバントのプログラム構造は、オブジェクト指向としてとらえると理解しやすい。各サーバントは、ファイルインタフェースを持つオブジェクト（以下 “仮想ファイルオブジェクト” と呼ぶ）の集まりを管理している。

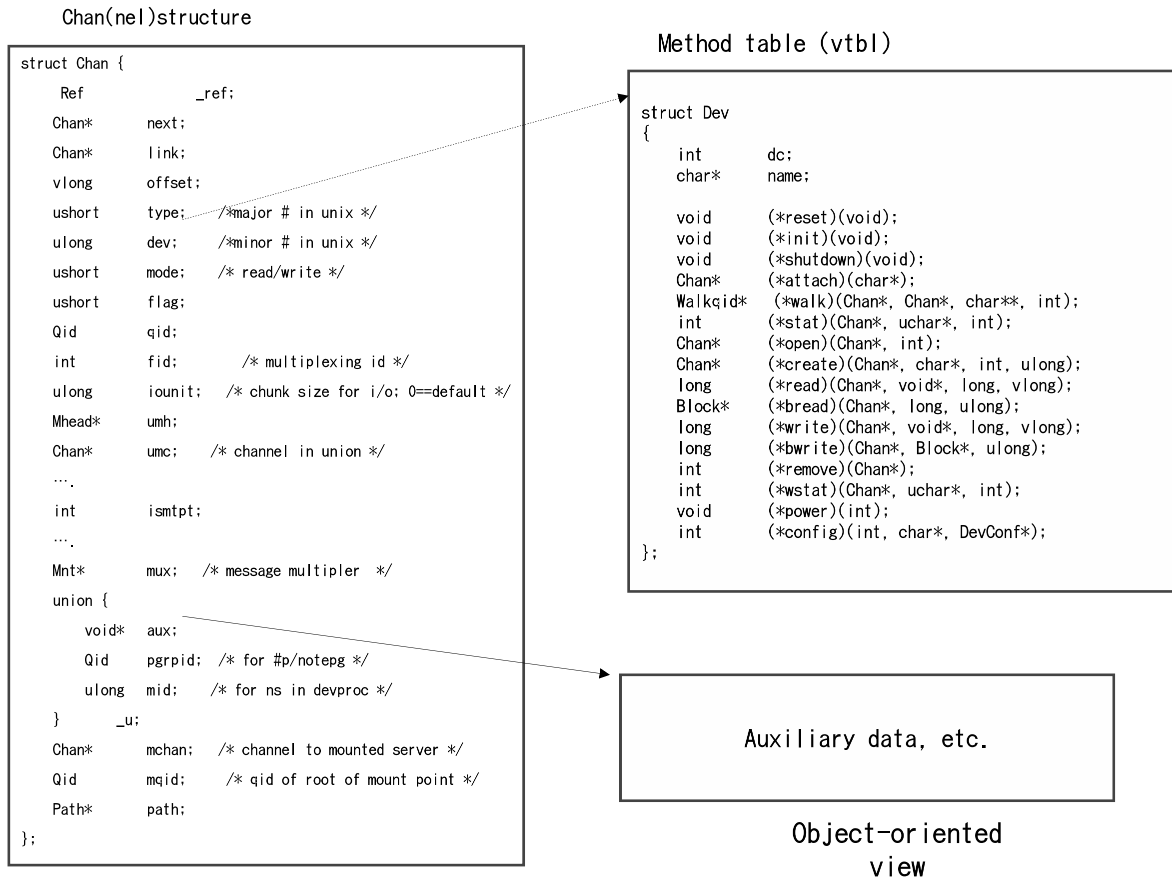


図 10.2: Chan 構造体とオブジェクト構成

このファイルインタフェースとは、Dev 構造体で定義されている関数 (`create()`, `open()`, `write()`, `read()`, `stat()`,,,,) を意味する。

各仮想ファイルオブジェクトは、サーバントの名前空間 (directory tree) につながれており、path 名でアクセスすることができる。

Unix では、各ファイルが "inode" 番号によりファイルシステム内でユニークに識別されるように、LP49/Plan9 では, 個々の仮想ファイルオブジェクトは `Qid` により、サーバント内でユニークに識別されている。`Qid` は、以下の構造体である。

```
typedef struct Qid {
   uvlong  path;  //サーバント内でユニークな番号
   ulong   vers;  // version 番号。内容が更新される毎に+1
   uchar   type;  //オブジェクトのタイプ
} Qid;
```

仮想ファイルオブジェクトのインタフェースは統一されているが、同一サーバント内でもメソッド内容は同じ必要はない。例えば、メソッドはオブジェクトの `Qid` が与えられると、`Qid.type` を見て、それに応じた処理を行う。

サーバントのソースプログラム `devxxx.c` は、一般に以下のような内容を持っている。

```
/********* devxxx.c **********/
```

```
#include        .....
     :

ststic void xxxinit(void)   // init メソッド
{     .......           }

static Chan* xxxattach(char* spec)  // attach メソッド
{       .......         }

static Walkqid* xxxwalk(Chan* c, Chan *nc, char** name, int nname)
{       .........       }

static int  xxxstat(Chan* c, uchar* dp, int n) // stat メソッド
{        ........       }

static Chan* xxxopen(Chan* c, int omode) // open メソッド
{        ........       }

static void xxxclose(Chan* _x) // close メソッド
{   .........           }

static long xxxread(Chan* c, void* buf, long n, vlong off) //read メソッド
{        ........       }
    :

static long xxxwrite(Chan* c, void* buf, long n, vlong off) //write メソッド
{        .......       }

Dev xxxdevtab = {   //この Dev テーブル経由で呼ばれる
      'r',        //このサーバントを識別する１文字。#S の'S' など。
      "xxx",      // サーバントの管理名
      devreset,  // dev.c の関数
      xxxinit,
      devshutdown,// dev.c の関数
      xxxattach,
      xxxwalk,
      xxxstat,
      xxxopen,
      devcreate,  // dev.c の関数
      xxxclose,
      xxxread,
      devbread,  // dev.c の関数
      xxxwrite,
      devbwrite, // dev.c の関数
      devremove, // dev.c の関数
      devwstat   // dev.c の関数
};
```

各メソッドは、外部から Dev テーブル経由でアクセスされる。プロセスが devxxx にアクセスする仕組みを以下に示す。

```
Fgrp  *fgrp = Proc(プロセス) テーブルの fgrp の値;
Chan *c = fgrp->fd[fd]; //プロセスの fd グループテーブル を fd で index して
                         //Chan テーブルを求める。

n = devtab[c->type]->read(c, ...); // chan->type 値で devtab を index して
                                   // devxxx の Dev テーブルにアクセスし、
                                   // xxxread(...) を実する。
```

メソッドは、Chan 引数から Qid 情報を求めて目的オブジェクトにアクセスし、Qid.type に応じた処理を行う。

10.5 Servant はファイルシステムである

(1) 名前空間

サーバントの各要素はディレクトリトリー (つまり名前空間) に載っており、ファイルとしてアクセスや操作を行うことができる。だから、サーバントもファイルシステムである。change directory して各要素をたどれる。

サーバント名をプロセスの名前空間に結合 (bind) することにより、プロセスからサーバントの各要素もアクセスすることができるようになる。

例えば IP サーバント (#I, Internet プロトコル) は、以下のように名前空間を構成している。IP サーバントは、プロセス名前空間の通常 /net に接続される。

```
--- net/ ---+-- tcp/ ---+-- stat
            |           |-- clone
            |           |-- 0/ --+-- ctl    ←  コネクション-1
            |           |        |--  data
            |           |        |--  local
            |           |        :
            |           |-- 1/ --+--ctl     ←  コネクション-2
            |           |    :   |--  data
            |           |        |--  local
            |           |        :
            |
            +-- udp/ ---+-- stat
            |           |-- clone
            |           :
            :
```

個々のコネクションもファイルとして表現されており、clone ファイルをオープンすることで生成される。例えば tcp/clone を open() すると、順次 tcp/0, tcp/1, tcp/2,,,, が生成される。

(2) qid

サーバント内の各要素は、下記構造の "Qid" によってユニークに識別される。これは、UNIX ファイルシステムの各ファイルが inode 番号によってユニークに識別される事に該当する。

```
typedef struct Qid {                                    +-8 -*--32-+--64----*
   uvlong  path;     // 64bits                          |type|vers | path   |
   ulong   vers;     // 32bits                          *----*-----*--------*
   uchar   type;  ← QTDIR, QTAPPEND, QTEXECL, QTMOUNT, QTAUTH, ,,,
} Qid;
```

qid の各フィールドの意味は以下のとうりである。

path 64 ビットのユニーク値

vers Version 番号で、更新されるごとにカウントアップされる。

type ディレクトリ (QTDIR), 追加のみ (QTAPPEND) などの属性

(3) デバイスの設定コマンド

各デバイスは、一般に設定コマンドを受け付けるファイル (ctl ファイルなど) を持っている。デバイスのモードなどを設定するには、このファイルに ascii 文字のコマンドを書き込めばよい。ascii 文字なので、人間が読むことができ、また CPU の Endian にも左右されない。

10.6　代表的なサーバント

(1) コンソールサーバント #c

(2) ストレッジサーバント #S

(3) FDD サーバント #S

(4) USB サーバント #U

(5) 環境変数サーバント #S

(6) Proc サーバント #p

(7) Pipe サーバント #|

(8) Root ファイルサーバント #R

(9) サーバ登録簿 #e

(10) マウントサーバント #M

マウントサーバントは、システムコールを 9P メッセージに変換してサーバと通信を行うプログラムである。重要機能であるので、別途詳細に説明する。

(11) Ether サーバント #l

(12) IP サーバント #I

(13) VGA サーバント #V

第11章　名前空間の管理

11.1　名前サーチ：パス名から目的オブジェクトを求める

ここでの名前サーチとは、パス名 (例： “/a/b/c/d”) から目的の仮想ファイルオブジェクトを見つけることをいう。

LP49/Plan9 では、プロセスに見える名前空間は、サーバントやサーバの名前空間を接続 (マウント）したものである。また、同一のマウントポイントに複数をマウントすることができる (“ユニオンマウント”) という特長がある。このために名前サーチは、かなり複雑な処理となっている。

サーバント内の名前サーチは、各サーバントの役目である。各サーバントは以下の形式の walk() 関数を持っている。walk は change directory 思ってよい。　先に説明したように、directory を含めて仮想ファイルオブジェクトは全て Chan テーブルで表現されている。Chan テーブルは、仮想ファイルオブジェクトのハンドルである。

```
static Walkqid*  walk(Chan *startChan, Chan *targetChan, char *names[], int numOfNames);
    startChan: [入力]　名前サーチの開始点を表す Chan テーブル
    targetChan:[出力]　目的オブジェクトを表す Chan テーブル
    names: [入力]　名前の配列
    numOfNames: [入力]　名前の配列の様子数
    返り値： directory のたどり方の情報
```

サーバント内の名前サーチは、この関数を呼ぶことで完結する。プロセスに見える名前空間は、多数のサーバントやサーバの名前空間がマウントされたものであるので、全体としての名前サーチは、マウントポイントを見つけながら各サーバントやサーバをたどっていく必要がある。その処理の中心となるのが、chan.c プログラムである。

11.2　チャネルプログラム chan.c

chan.c は、各プロセスの名前空間管理の中核をなすプログラムである。ソースコードは、“src/9/port/chan.c” である。本プログラムに含まれる代表的な関数を以下に示す。

(1) Chan * namec(char *name, int amode, int omode, ulomg permission);

パス名 (name) からチャネル (Chan) テーブルを求める関数である。Unix の namei( ) 関数に相当するが、Unix の namei( ) は inode 番号を返すのに対し、本 namec( ) は Chan テーブルを返す。

(2) int walk(Cham **cp, char **names, int nnames, int nomount, int *nerror);

”Change directory” を行う関数である。

例えば、第 1 パラメータの Chan サーブルが "/a" を指しているとし、第 2 パラメータの names が {"b", "c"} だったとすると、"/a"から順次 "b", "c" を探して行き、第 1 パラメータに"/a/b/c"の Chan テーブルを返す。

(3) cmount(...), domount(...), findmount(...), など

マウントの処理を行う関数である。具体的な処理内容は、次のセクションで説明する。

Plan9 の chan.c は、開発スタート時から拡張に拡張をつづけてきたプログラムだけに、大変に込み入って難解になっている。解読にチャレンジしてみてほしい。現 LP49 では Plan9 の chan.c を最小量修正で利用しているが、LP49 phase2 では名前空間の管理法を根本的に見直して処理を徹底的に簡素化する予定である。

11.3 マウントの仕組み

プロセス個別名前空間は Plan9 の一大長所であり、プロセス毎に個別の名前空間を構成できる融通性、アクセス権保護に非常に効果的である。LP49 も強力なこの仕組みを継承している。

また、同一のマウントポイトに複数のファイルトリーをマウントできる "ユニオンマウント" も、 Plan9 および LP49 の融通性の一つである。Unix では同一のマウントポイントに複数のファイルトリーをマウントすると、最後のもののみがアクセス可能で、それ以外は隠されてしまうのに対し、ユニオンマウントでは (同一名前でない限り) 複数のファイルトリーが同一ディレクトリに現れる。

名前空間管理の中核をなすのは、" Pgrp, Mhead, Mount" の各テーブル (ソースは "src/9/port/portdat.h") と、"chan.c" プログラム (ソースは "src/9/port/chan.c") である。

プロセステーブルは名前空間テーブル Pgrp へのポインタをもっている。名前空間を共用するプロセス群は、同一の Pgrp テーブルにポイントしている。

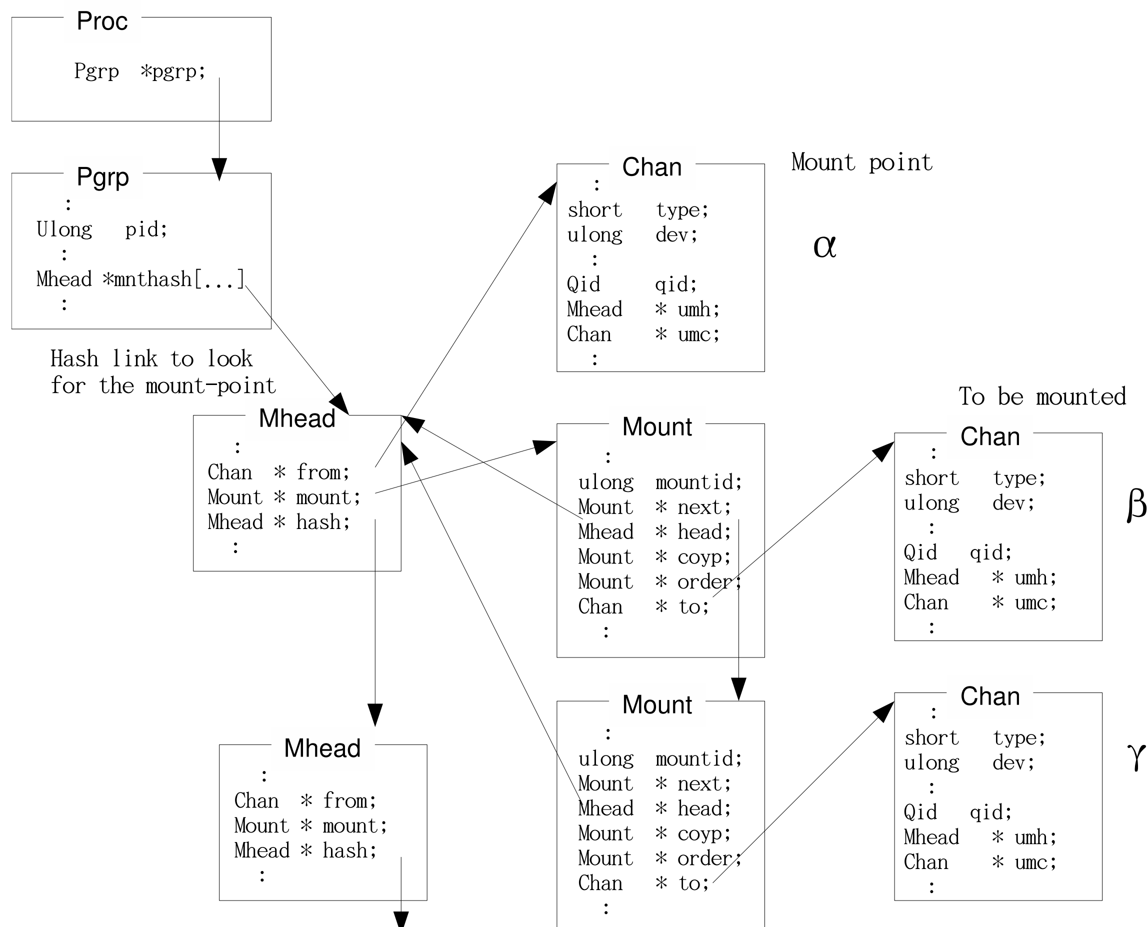


図 11.1: 名前空間マウントのデータ構造

名前空間のマウントのデータ構造を図 11.1 に示す。この例では、マウントポイント「α」に「β」と「γ」がユニオンマウントされている。α, β, γ は `Chan` テーブルで表現されている。

この様に (マウントポイントを含めて）全てのオブジェクトは `Chan` テーブルで表現されている。`Chan` テーブルは、オブジェクトアクセスのハンドルと説明したとうりである。

目的オブジェクトの識別は、サーバント種別 (`Chan.type`) とサーバント内ユニーク値 (`Chan.qid`) が合致するか否かでおこなわれる。これを "検索情報" とよぶことにする。

今パス名に "α" が含まれているとする。マウントの解決は、以下の様に行われる。

1. `Proc`(プロセス) テーブルの `pgrp`(process group) フィールドから、`Pgrp` テーブルにアクセスする。`Pgrp` テーブルは、このプロセスが属する名前空間を表す。

2. "検索情報" のハッシュ値で `Pgrp.mnthash[...]` を index して `Mhead` のハッシュリンクにアクセスする。

3. `Mhead.from` フィールドから `Chan` テーブルにアクセスする。これは、マウントポイントを意味する。これが "α" と一致したら、マウントポイントが見つかった分けである。一致しなかった場合は hash リンクをたどる。

4. "α" と一致した場合は、これがマウントポイントなので、ここにマウントされている部分トリー上で探すことになる。

   具体的には、`Mhrad.mount` から `Mount` テーブルにアクセスし、`Mount.to` をたどってマウントされている部分トリーの `Chan` テーブルにアクセスする。こうして、β, γ を調べる。

5. β, γ のどちらが先にサーチされるかは、mount/bind コマンドのオプション (`-a: after, -b: before,` 無指定：`replace`) による。

第12章　サーバ接続とマウントサーバント

12.1　9P: サーバとの通信プロトコル

9P プロトコルは、レイア的には NFS 分散ファイルサーバのプロトコルと同等であるが、NFS がステートレス指向であるのに対し、9P はステートフルである。制御システム用として 9P が充分であるか否かは、これからの検討課題である。

9P プロトコルの一覧を以下に示す。

フィールド `[n]` は、このフィールドのデータ長が n byte であることを、`[s]` は可変長であることを示す。`Txxx` はコマンド名、`Rxxx` は返答名であり、1 バイトを占めている。

`tag` フィールドと `fid` フィールドは多重処理のための識別子であり、以下の内容をもつ。File descriptor (FD 値) はサーバ側が決定するのに大して、`tag，fid` はクライアント側が与える。

tag　要求メッセージと返答メッセージの対応をとる番号。

fid　各 "仮想オブジェクトファイル" を識別する番号。

例えば『`size[4] Tversion tag[2] msize[4] version[s]`』は、`size[4]`=本メッセージのトータルサイズ、`Tversion`=プロトコル version チェックコマンド、`tag[2]`=コマンドと応答の対応をとるタグ、`msize[4]`=許容最大メッセージサイズ、`version[s]`=version 名、を意味する。

```
size[4] Tversion tag[2] msize[4] version[s]   → プロトコルの version
size[4] Rversion tag[2] msize[4] version[s]
size[4] Tauth tag[2] afid[4] uname[s] aname[s]    → 認証
size[4] Rauth tag[2] aqid[13]
size[4] Rerror tag[2] ename[s]
size[4] Tflush tag[2] oldtag[2]
size[4] Rflush tag[2]
size[4] Tattach tag[2] fid[4] afid[4] uname[s] aname[s]    → マウント
size[4] Rattach tag[2] qid[13]
size[4] Twalk tag[2] fid[4] newfid[4] nwname[2] nwname*(wname[s]) → change directory
size[4] Rwalk tag[2] nwqid[2] nwqid*(wqid[13])
size[4] Topen tag[2] fid[4] mode[1]                → Open( )
size[4] Ropen tag[2] qid[13] iounit[4]
size[4] Tcreate tag[2] fid[4] name[s] perm[4] mode[1]
size[4] Rcreate tag[2] qid[13] iounit[4]
size[4] Tread tag[2] fid[4] offset[8] count[4]
size[4] Rread tag[2] count[4] data[count]
size[4] Twrite tag[2] fid[4] offset[8] count[4] data[count]
size[4] Rwrite tag[2] count[4]
size[4] Tclunk tag[2] fid[4]
size[4] Rclunk tag[2]
size[4] Tremove tag[2] fid[4]
size[4] Rremove tag[2]
size[4] Tstat tag[2] fid[4]
size[4] Rstat tag[2] stat[n]
size[4] Twstat tag[2] fid[4] stat[n]
size[4] Rwstat tag[2]
```

これから分かるように、9 P プロトコルは要求と返答が対になった同期型のやりとりである。(更に多様な分散処理をサポートするために、将来的には非同期型の拡張も考えられる。)

12.2 サーバリンクとサーバ登録サーバント /srv

(1) サーバリンク

LP49core とサーバは、9P メッセージを pipe あるいは TCP 接続を通して通信し合う。サーバに接続された pipe あるいは TCP 接続を "**サーバリンク**" と呼んでいる。サーバリンクは多重化されている。つまり、複数の 9P メッセージが一つのサーバリンクの中を流れる。

(2) サーバ登録サーバント /srv

サーバ登録サーバント (#s) は、サーバリンクを登録しておくデータベース (サーバ登録簿) であり、`/srv` に接続されている。サーバ登録サーバントの各エントリーは、`/srv/ServiceName` という名前のファイルである。これらのファイルの中身は、LP49core が理解できるサーバリンク情報であり、人間が読むものではない。

【例】

```
/srv/dos    (DOS ファイルサーバ)
/srv/ext2   (EXT2 ファイルサーバ)
/srv/9660   (ISO9660 形式 CD ファイルサーバ)
```

サーバ登録サーバントのソースコードは、`src/9/port/devsrv.c`) である。これは簡単なプログラムである。

(3) ローカルサーバの場合のサーバ登録

ローカルサーバは、サーバリンクとして一般に pipe を使う (もちろん TCP 接続も使えるが)。

大部分のサーバは、立ち上げると pipe を生成して自動的に `/srv/ServiceName` に登録する。この "ServieName" は起動時に指定することが出きるし、指定しなければ default の名前が使われる。

自動登録でないサーバの場合は、`/src/SrviceName` ファイルを `create` して、そこに pipe あるいは　TCP 接続の file descriptor を書き込むことでサーバ登録が行われる。

(4) リモートサーバの場合のサーバ登録

リモートサーバの場合は、TCP 接続がサーバリンクとなる。

○ announc, listen, accept

○ /bin/srv

(5) サーバのマウント

サーバリンクをクライアントプロセスの名前空間にマウントすることで、サーバの名前空間が見えるようになる。

【マウントコマンド】

mount　[option...]　/srv/ServiceName マウントポイント　付加指定

(参考) サーバのマウント

Qsh シェルでは、サービス記録簿サーバント (#s) を/srv にフロッピーディスクサーバント (#f) を/dev に接続し、dossrv サーバを機動してそのサーバリンク (パイプ) を/srv/dos として登録し、そしてそのサービスリンクを/tmp にマウントしている。この模様をを以下に示す。

```
<< Qsh shell が dossrv を/tmp にマウントする模様 >>
  bind  #s /srv
      →   /srv がサーバ登録簿となる

  bind  -a #f  /dev
      →   /dev にフロッピーがバインドされる。
          これにより、 /dev/fd0ctl,  /dev/ctl が見えるようになる。

  dossrv
     →  dossrv (DOS ファイルサーバ) は、起動されるとパイプを生成し/srv に登録する。
        これにより /srv/dos が作られる。

  mount -a  /srv/dos  /tmp   /dev/fd0disk
     →  DOS ファイルサーバ (/srv/dos) を /tmp ディレクトリにマウントする。
        DOS ファイルサーバは、フロッピー (/dev/fd0disk) にアクセスする。
        これにより、 /tmp はフロッピーのファイルシステムをアクセスするようになる。
```

12.3　マウントサーバント devmnt.c の内容

サーバをマウントして Remote Procedure Call を行っているのが “mnt” サーバント (#m) である。ソースコードは src/9/port/devmnt.c である。このプログラムの実装はかなり複雑であるが面白いので、ぜひ解読されたい。

(1) システムコールから 9P メッセージへの変換

サーバに対するシステムコールは devmnt.c に伝えられて、そこで 9P メッセージに変換される。(Cf. libc ライブラリの convS2M(), convM2S())。

devmnt.c の関数呼び出しと 9P メッセージの関係を図 12.1 に示す。

(2) サーバのマウント

サーバ”sss” のサーバリンクが、サービス記録簿に (/srv/sss) として登録されているものとする。サーバの部分ファイルトリー “/b” をプロセスの/m/n にマウントするものとする。

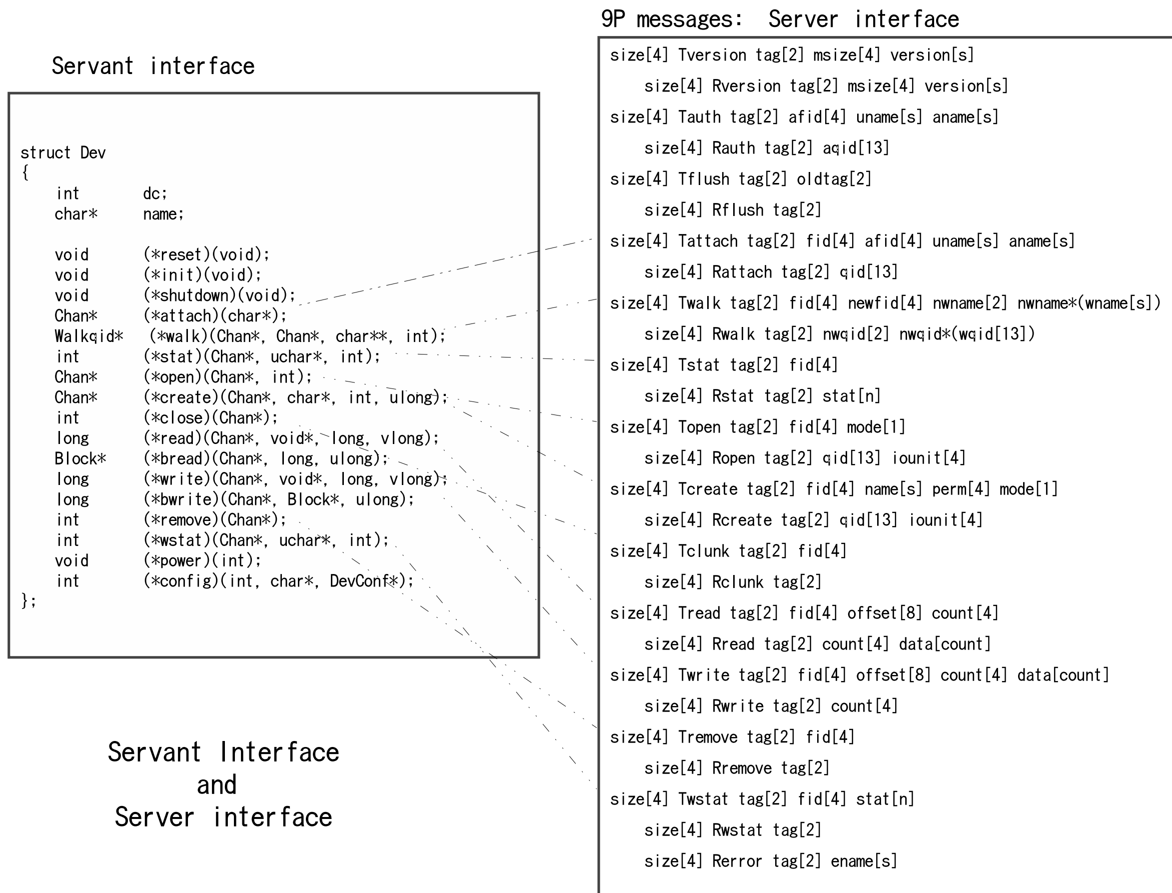


図 12.1: 9P メッセージへの変換

```
mount -a  /srv/sss  /m/n   /b
```

→ sss サーバ（/srv/sss）の部分空間 /b を /m/n にマウントする。

これにより、サーバの/b 以下の部分ファイルトリーがプロセスの/m/n にマウントされ、サーバ上のファイルが見えるようになる。

(3) devmnt **のデータ構造**

サーバアクセスに関するデータ構造を図 12.2 に示す。

この図の各テーブルは、以下の意味を持つ。

- **Chan(SvrLink)** サーバリンクを表すハンドル (Chan テーブル)。サーバリンクが pipe か　TCP コネクションかにより、Chan.type フィールドは pipe サーバント (#|), あるいは　IP サーバント (#I) をさしている。
- **Mnt** サーバリンク上に複数の 9P メッセージを多重化するためのテーブル。
- **Mntrpc** ここの 9P メッセージは Mntrpc テーブルで表され、Mnt テーブルの待ち行列 Mnt.queue に挿入される。
- **Chan(/m/n)** サーバのマウントされるサブトリーの入り口点 /b を表すハンドル。サーバの/b に対する**代理 (Proxy) オブジェクト** として機能する。Chan.type フィールドは マウントサーバント (#M) をさしている。

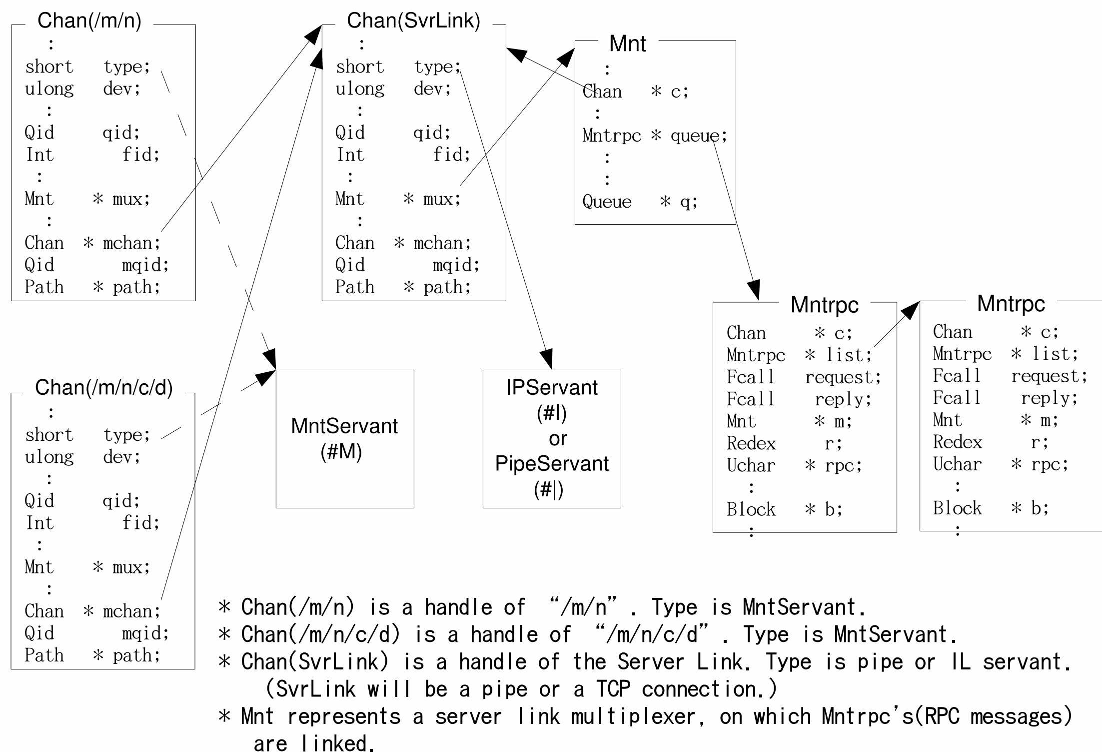


図 12.2: Mnt サーバント devmnt.c のデータ構造

- **Chan(/m/n/c/d)** サーバのサブトリーのオブジェクト “c/d” を表すハンドル。サーバの “c/d” に対する代理 (Proxy) オブジェクトとして機能する。`Chan.type` フィールドは マウントサーバント (#M) をさしている。

(4) マウントの処理概要: mntattach()

サーバのマウント処理の中核は、`devmnt.c` の `mntattach()` である。`sss` サーバ の部分空間`/b` を `/m/n` にマウントすると、以下のように処理が進む。

1. マウントにより、サーバリンクのハンドル `Chan(SvrLink)` が割り付けられる。
2. `Chan(SvrLink)` は、多重化のためのテーブル `Mnt` と結合される。
3. サーバ向け操作は 9P メッセージに変換されて `Mntrpc` テーブルに登録される。
4. `Mntrpc` テーブルは `Mnt` テーブルの queue につながれて、順次サーバに送られ、返答メッセージがくるのを待つ。
5. サーバから正当な返答メッセージを受けたら、サーバのマウント始点を表すハンドル `Chan(/m/n)` を割り当て、`Chan(SvrLink)` にリンクする。`Chan(/m/n)` は代行 (Proxy) オブジェクトであり、`Chan(/m/n)` に対する操作は 9P メッセージに変換されて RPC が行われる。

(5) open() の処理概要: mntopen(), mntwalk()

サーバファイルの open 処理の中核は、`devmnt.c` の `mntopen()`, `mntwalk()` である。
次に、現在`/m/n` にいて `open("c/d",,,)` した場合のデータ構造を説明する。以下のように処理が進む。

1. まず、"c/d" に行くために、9P メッセージ Twalk（,, "c", "d" ,,,）がサーバに送られる。
2. サーバから正当な返答メッセージを受けたら、サーバのマウント始点‘‘/b’’表すハンドル Chan(/m/n/c/d) を割り当て、Chan(SvrLink) にリンクする。Chan(/m/n/c/d) は代行 (Proxy) オブジェクトであり、Chan(/m/n/c/d) に対する操作は 9P メッセージに変換されて Remote Procedure Call が行われる。

(6) 多重処理の仕組み

○ Explanation: to be added.

12.4 サーバ記録簿サーバント devsrv.c の内容

(1) サーバリンクの登録法

サーバ記録簿サーバントは /srv に接続されている。

一般にサーバは、立ち上がり時にサーバリンクとして pipe を生成する (LP49 の pipe は、両方向である)。その一方をサーバにつなぎ、他方を /srv/<ServerName>としてサーバ記録簿サーバントに登録する。

たとえば、あるサーバが “/srv/abc” としてサーバ登録簿に登録する場合の手順は以下のようになる。

```
int  pipefd[2];
int  fd;
char buf[20];
  :
pipe(pipefd);  //パイプを生成する
  :
fd = create("/srv/abc", OWRITE|ORCLOSE, 0666); // "/srv/abc"を生成
sprint(buf, "%d", pipefd[1]);
write(fd, buf, strlen(buf));  // "/srv/abc"に fd を書き込む
  :
サーバは pipefd[0] にて 9p メッセージの読み書きを行う
  :
```

クライアントが、このサーバを “/mnt/abc” にマウントするには以下のコマンドによる。

```
mount フラッグ /srv/abc  /mnt/abc  付加指定
```

(2) devsrv.c の内容

サーバ登録で “/srv/abc” ファイルに “fd” を書き込むと、以下の処理を行う devsrv.c の srvwrite() が呼ばれる。

```
Chan  *c1;
Srv   *sp;
  :
c1 = fdtochan(fd, -1, 0, 1);  // fd で指定された Chan テーブル
```

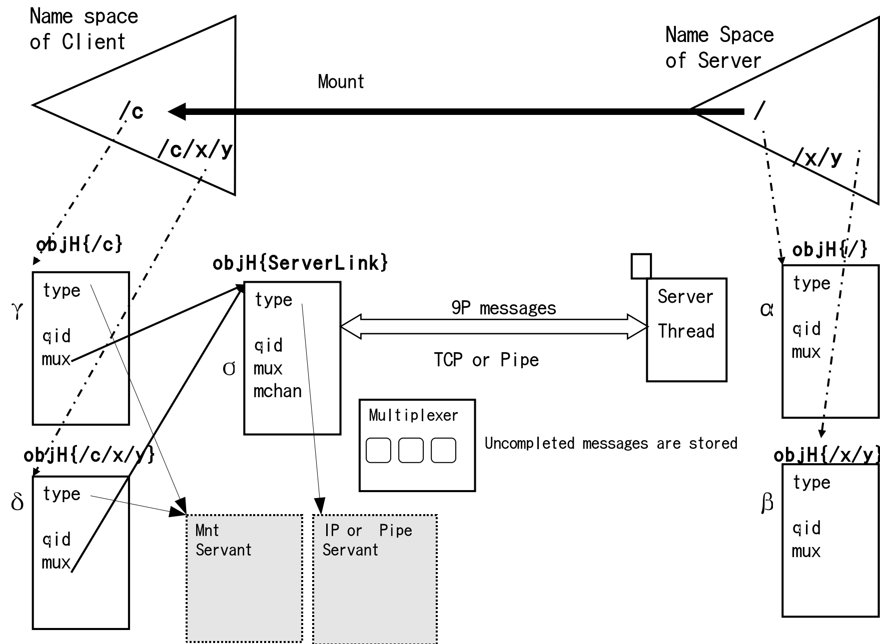


図 12.3: リモートオブジェクトアクセス

```
  :
sp = srvlookup(....);  // sp は "/srv/abc" を指す
  :
sp->chan = c1;  // "/srv/abc" に fd で指定された Chan テーブルをつなぐ
  :
```

この仕組みにより、サービス登録簿からサーバリンクが求まる。

12.4.1 サーバアクセス処理の具体例

サーバの “/” をプロセスの名前空間 “/c” にマウントし、“/c” 及び “/c/x/y” にアクセスした場合の処理内容を図 12.3 に示す。

1. サーバ登録簿への登録: サーバ登録簿にサーバリンクを登録することにより、“objH{ サーバリンク }” (図の σ) が割り当てられる。

2. サーバのマウント: mount コマンドによりサーバをマウントする。
   LP49[/]: mount -ac /srv/dos /c /dev/sdC0/dos

   /srv/dos はサーバリンクを, /c はマウントポイントを, /dev/sdC0/dos はサーバ名前空間のマウント開始位置を、-ac は書き込み可能で after にマウントすることを意味する。

   これにより、サーバとの間で attach メッセージなどのやり取りを行い、“objH{/c}” (図の γ) テーブルを割り当て、以下の設定を行う。

   - type フィールドには Mnt サーバントを設定。つまり、objH{/c} のクラスを Mnt サーバントである。
   - mchan フィールドには objH{ サーバリンク } を設定。

- qid フィールドには サーバからもらった `qid` を設定。この `qid` は、サーバ内のマウント開始位置オブジェクトの `qid` である。

以上により、`objH{/c}` は、サーバ内のマウント開始位置オブジェクト (図の α) の **Proxy オブジェクト**となる。こうして、`objH{/c}` への操作はマウントサーバントによって **9P メッセージ**に変換され、サーバリンクに送り出される。サーバからの返答を受けたらそれをクライアントに返す。

3. サーバ名前空間での操作: 例えば `/c` にて "`open(‘‘x/y’’, OREAD)`" を実行したとする。この場合は、サーバとの間で `walk, open` メッセージなどをやり取りして、"`objH{/c/x/y}`" (図の δ) テーブルを割り当て、以下の設定を行う。

   - type フィールドには マウントサーバントを設定。
   - mchan フィールドには `objH{` サーバリンク } を設定
   - qid フィールドには サーバからもらった `qid` を設定。この `qid` は、サーバ内の`/b/c/d` オブジェクト (図の β) の `qid` である。

   以上の設定により、`objH{/c/x/y}` は、サーバ内の`/x/y` オブジェクトの Proxy オブジェクトとなる。

12.5 リモートサーバの接続法

(1) announce, listen, accept ライブラリ関数

○ Explanation: To be added.

(2) listen1 コマンド

○ Explanation: To be added.

(3) srv コマンド

○ Explanation: To be added.

第13章　サーバ接続の管理

サーバは、DOS ファイルサーバ、 Ext2 ファイルサーバの様に永続的に立ち上げておくものと、Extfs サーバの様に接続要求対応に立ち上げるものとがある。後者は Unix でいえば “inetd” によって起動されるプログラムに対応する。LP49 では、“inetd” に似た役目をするのは　“listen1, listen” である。

13.1　永続的サーバの立ち上げ時の処理

永続的サーバをクライアントに見えるようにするには、以下のいずれかによる。

- サーバをサーバ登録簿に登録し、クライアントがマウントする。
- サーバがプロセスの名前空間にマウントする。
- ネットワーク上に announce して接続要求がくるのを待つ。

(1) サーバ登録簿への登録

DOS ファイルサーバなど永続的サーバは、立ち上がり時にサーバ登録簿に自分のサーバリンクを登録する。例えば DOS ファイルサーバの “/srv/dos)’’ には、サーバリンク情報が記録されている。クライアントは、これを用いてサーバを自分の名前空間にマウントする。

永続的サーバのアウトラインを以下に示す。ローカル使用が主体のサーバの場合には、マウントまで行うものもある。

Plan9 では、サーバ初期化プロセスが新しいプロセスを RFNOWAIT 付きで rfork() し, 実際のサービスは新プロセスに行わせている。そして、サーバ初期化プロセスは exit() している。つまり新プロセスがバックグラウンドで動作し、親プロセスはすぐに消えている。

LP49 では、サーバ初期化プロセスがそのまま継続してサービスを提供する。このためにサーバプロセスを Backgraoud プロセスとして起動することとする。

【サーバ登録簿に登録するサーバ】

```
void main(...)
{
    int  pipefd[2];
      :
    pipe(pipefd);
      :
    // 以下の様に pipefd[1]  をサーバ登録簿に登録
    fd = create("/srv/サーバ名" ...); //
    fprint(fd, "%d", pipefd[1]);
      :
      :
    srvloop(pipefd[0]);   // pipefd[1] はサーバが読み書きする。
}

void srvloop(int  fd)
{
    for(;;){
        n = read9pmsg(fd, ...);    // 9P メッセージを受ける
        if (n , 0) break;
```

```
            :
        convM2S(...);   // 9P メッセージを内部コードに展開する
            :
        (*fcalls[req->type])();  // メッセージに対応したメソッドを実行する
            :
        convS2M(...);   // 内部コードを 9P メッセージに変換する
            :
        write(fd, ....);  // 返答メッセージを送り出す
    }
}
```

(2) プロセス名前空間へのマウント

サーバが自分でマウントまで行う場合のプログラムである。mount() コールはサーバリンクを介してサーバとの間で attach メッセージの交信を行うので、両者は別のスレッドでなければならない。そこで、サーバスレッドを生成することが必要となる。

【サーバ登録簿に登録するサーバ】

```
void main(...)
{
    int  pipefd[2];
      :
    pipe(pipefd);
      :
    // 以下の様に pipefd[1]  をサーバ登録簿に登録
    fd = create("/srv/サーバ名" ...); //
    fprint(fd, "%d", pipefd[1]);
      :
    // srvloop() を実行するスレッドを生成する
    l4thread_create(srvloop, ,,);  // pipefd[0] が 9P メッセージの受け口

    mount(fd, -1, マウントポイント,,,);
      :
    sleep(...);
}

void srvloop(int  fd)
{
    for(;;){
        n = read9pmsg(fd, ...);    // 9P メッセージを受ける
        if (n , 0) break;
          :
        convM2S(...);   // 9P メッセージを内部コードに展開する
          :
        (*fcalls[req->type])();  // メッセージに対応したメソッドを実行する
          :
        convS2M(...);   // 内部コードを 9P メッセージに変換する
          :
        write(fd, ....);  // 返答メッセージを送り出す
    }
}
```

(3) ネットワーク上の announce, listen

(4) lib9p の `postmountsrv_l4()` **と**`listensrv_l4()`

13.2 オンデマンドサーバの立ち上げ時の処理

Extfs サーバなどオンデマンドサーバは、listen1 コマンド（次項で説明）で指定しておくことにより、接続要求が来たときにサーバが立ち上がる。従って、サーバ登録簿には登録する必要がない。

listen1 コマンド は、指定ポート番号を監視していて、接続要求が来たら TCP 接続（サーバリンク）を確立して、その fd をサーバに引き継ぐ。

オンデマンドサーバのアウトラインを以下に示す。

【オンデマンドサーバのスタート処理概要】

```
int  netfd;  // listen1 コマンドが netfd に NW 接続（サーバリンク）の fd を設定する。

void  main(...)
{
      ;
      ;
    for(;;){
        n = read9pmsg(netfd, ...);     // 9P メッセージを受ける
        if (n , 0) break;
          :
        convM2S(...);   // 9P メッセージを内部コードに展開する
          :
        (*fcalls[req->type])();  // メッセージに対応したメソッドを実行する
          :
        convS2M(...);   // 内部コードを 9P メッセージに変換する
          :
        write(netfd, ....);  // 返答メッセージを送り出す
    }
}
```

13.3 サーバ要求の待ち受け

(1) listen1 コマンド

```
char data[60], dir[40], ndir[40];  //%
char *progname;   //%

void main(int argc, char **argv)
{
    int ctl, nctl, fd;
    int  rc; //%

      :
      :
    ctl = announce(argv[0], dir);  //ポート番号を公知にする
      :

    for(;;){
        nctl = listen(dir, ndir);  //接続要求がくるのを待つ
        switch(rfork(RFFDG|RFPROC|RFNOWAIT|RFENVG|RFNAMEG|RFNOTEG)){
        case -1:
            reject(nctl, ndir, "host overloaded");
            close(nctl);
            continue;

        case 0:
            fd = accept(nctl, ndir);  //接続要求を受け付ける
                 :
            fprint(nctl, "keepalive");
            close(ctl);
            close(nctl);
            sprint(data, "%s/data", ndir);
            rc = bind(data, "/dev/cons", MREPL);   //% original is MREPL
            dup(fd, 0);
            dup(fd, 1);
            dup(fd, 2);
            close(fd);
            exec(argv[1], argv+1);  // 指定されたサーバを実行する
            fprint(2, "exec: %r");
            exits(nil);

        default:
            sleep(10000); //%% ? Festina Lente
            break;
        }
    }
}
```

(2) listen コマンド

13.4 動的なサーバ接続要求

(1) srv コマンド

src/cmd/simple/srv.c

```
void main(int argc, char *argv[])
    {
        int  srvfd, fd;
          :
          :
        dest = netmkaddr(dest, 0, "564");
        srvfd = dial(dest, 0, dir, 0);  //目的ノードの目的ポートに接続する
          :
        fd = create("/srv/サーバ名", OWRITE, 0666);
        sprint(buf, "%d", fd);
        write(fd, buf, strlen(buf));    //サーバ登録簿にサーバリンクを登録
          :
        if (マウントポイントの指定あり) {
             mount(fd, -1, マウントポイント, mountflag, "")
               :
        }
        :
        exits(0);
    }
```

第14章　デバイスドライバ

(1) ソースプログラム

デバイスドライバのソースプログラムは、`src/9/pc/*` にあるので、参照されたい。また、ドキュメント`LP49-yymmdd/doc/Pr`を参照されたい。

sdata.c ハードディスクドライバ

devfloppy.c フロッピードライバを含む

etherxxx.c `Ether card` のドライバ

usbuhcs.c USB ホストコントローラ

(2) 割込みハンドラ

(a) ソースコード

`src/libl4com/l4-p9-irq.c`

(b) 割込みハンドラ登録関数

```
void p9_register_irq_handler(int irq, void (*func)(void*, void*),void* arg, char *name);
void  intrenable(int irq, void (*f)(void*, void*), void* a, int tbdf, char *name);
void p9_unregister_irq_handler(int irq);
int intrdisable(int irq, void (*f)(void *, void *), void *a, int tbdf, char *name);
```

(c 割込みハンドラ登録の例

```
devether.c:448:
  intrenable(ether->_isaconf.irq, ether->interrupt, ether, ether->tbdf, name);
devi82365.c:671:
  intrenable(cp->irq, i82365intr, 0, BUSUNKNOWN, buf);
devpccard.c:641:
  intrenable(pci->intl, cbinterrupt, cb, pci->tbdf, "cardbus");
devusb.c:470:
  intrenable(uh->_isaconf.irq, uh->interrupt, uh, uh->tbdf, name);
kbd.c:534:
  intrenable(IrqAUX, i8042intr, 0, BUSUNKNOWN, "kbdaux"); //%
kbd.c:581:
  intrenable(IrqKBD, i8042intr, 0, BUSUNKNOWN, "kbd");  //%
main-l4.c:700:
  intrenable(IrqIRQ13, matherror, 0, BUSUNKNOWN, "matherror");
sdata.c:2189:
  intrenable(ctlr->irq, atainterrupt, ctlr, ctlr->tbdf, name);
```

(d) 割込み処理の手順

(d-1) 割り込みハンドラ登録

Cf. 上記説明

(d-2) 割込みハンドラスレッドの生成

```
main() -- in pc/main-l4.c
  :
  p9_irq_init()    -- in src/libl4com/l4-p9-irq.c
  :   ここで、割込みハンドラスレッドが生成される。
  :
```

(d-3) 割込み時の処理の流れ

```
デバイスが割込み発生
       --> L4 マイクロカーネルが割込みを検知
            --> 割込みメッセージを割込みハンドラスレッドに送る
                     --> 割込みハンドかの実行
```

第15章　物理メモリアクセス

(1) 関連ソースコード

```
src/9/port/xalloc-l4.c
src/9/port/alloc.c
```

(2) ヒープ域

```
(a) xalloc-l4.c includes:
   #define  XMAP_LADDR     ...
   #define  XMAP_PADDR     ...
   #define  XMAP_LOG2SIZE  ...

(b) (2 ** XMAP_LOG2SIZE) bytes are reserved for the heap area.

(c) Heap area is linearly mapped to physucal memory, and
   logical-physical address translation is very simple.
  <physical_addr> == <logical_addr> - XMAP_LADDR + XMAP_PADDR

(d) XMAP_LADDR, XMAP_PADDR  and XMAP_LOG2SIZE must satisfy
    the L4 Flex page constraint.
```

第16章　例外処理

Plan9 では、

```
if (waserror()) {
      .....
   nexterror();
}
     ....
poperror();
```

の形式からなる技巧的な例外処理を開発している。

これは、Java の構文『try{.......}catch(...){ ....} 』に類似した例外処理を OS と C 言語で実現しようというものである。実現の仕組み自体は、setjmp()/longjmp() と類似している。

しかしながら、以下の理由から LP49 ではこの手法は採用せずに、例外が生じたら関数のリターン値として通知することにした。

- 例外が生じた場合の制御フローを追いにくく、プログラム論理がわかりにくい。
- プロセス・スレッド単位に、複数レベルの例外ジャンプセーブ域を確保する必要があり、スレッドの多用を目指 した LP49 には適さない。

第IV部

ライブラリー

第17章　ライブラリー

17.1　各ライブラリの概要

(1) libl4com、libl4io

L4 マイクロカーネルの機能を使うためのライブラリ関数を含む。要リファイン。libl4io は、 LP49 のプリント機能が実装される前にプリントをするために作ったものである。

(2) libc

libc は、UNIX/Linux の libc 相当のライブラリ関数を含んでいる。LP49 のシステムコールに対応するライブラリ関数の場合は、パケットを編集して L4 のメッセージとして CORE 層に送信し、返答メッセージを待つ。

```
INLINE int syscall_iaiiim(int syscallnr, int arg0, void* arg1, int arg2, int arg3, int arg4,
                          L4_MapItem_t mapitem)
{
    L4_Msg_t     msgreg;
    L4_MsgTag_t  tag;
    int          result;
    unsigned     patrn = 0x411121; //  pattern("iaiiim");

    L4_MsgClear(&msgreg);
    L4_Set_MsgLabel(&msgreg, syscallnr);
    L4_MsgAppendWord(&msgreg, patrn);
    L4_MsgAppendWord(&msgreg, arg0);       //arg0 i
    L4_MsgAppendWord(&msgreg, (int)arg1);  //arg1 a
    L4_MsgAppendWord(&msgreg, arg2);       //arg2 i
    L4_MsgAppendWord(&msgreg, arg3);       //arg3 i
    L4_MsgAppendWord(&msgreg, arg4);       //arg4 i
    L4_MsgAppendMapItem(&msgreg, mapitem);  //map

    L4_MsgLoad(&msgreg);
    tag = L4_Call(SrvManager);
    if (L4_IpcFailed(tag))  return -1;

    L4_UnmapFpage(L4_MapItemSndFpage(mapitem));
    L4_MsgStore(tag, &msgreg);
    result = L4_MsgWord(&msgreg, 0);
    return result;
}

long     pwrite(int fd, void* buf, long nbytes, vlong offset) // iaiii
{
    union {
      vlong   v;
      ulong   u[2];
    } uv;
    uv.v = offset;
    L4_MapItem_t  map;
    map = covering_fpage_map(buf, nbytes, L4_Readable|L4_Writable);
    return syscall_iaiiim(PWRITE, fd, buf, nbytes, uv.u[0], uv.u[1], map);
}
```

(3) libbio

(4) libdbg

(5) lib9p

第V部

Init プロセスとシェル

第18章　init プロセス

LP49core が立ち上がると、ついで init プロセスが立ち上がる。

init プロセスは、以下のように名前空間を設定した上で qsh シェル、もしくは rc シェル を fork() する。init プロセスは、途中で

Which shell ? rc: hit 'r' RETURN; qsh: RETURN

と聞いてくるので、qsh シェルの場合は Return、rc シェルの場合は r を押して Return を打鍵する。

rc シェルは、現在実装中なので部分的に動作する。

1. コンソールサーバントを接続する
2. open(/dev/cons) をして、STDIN，STDOUT，STDDERR を開く。
3. FD サーバントを接続する
4. 環境変数サーバントを接続する
5. PROC サーバントを接続する
6. サービス記録サーバントを接続する
7. ルート FS サーバントを接続する
8. ......
9. ISO9660(CD) サーバをマウントする
10. ''/t/bin'' を ''/bin'' に接続する。
11. シェルを fork() する。

第19章 qsh: デバッグ用簡易シェル

qsh は、LP49 のデバッグのために作った疑似（quasi）シェルである。

qsh シェルは、LP49 のデバッグのために作った (dirty な) 簡易シェルである。

このため、create(), open(), read(), write(),,, など多くのシステムコールを組み込みコマンドで試すことができるようになっている。

組み込みコマンドの使用については、別資料‘‘LP49 の利用説明書’’を参照されたい。

第20章　rc: Plan9 の高機能シェル

"rc" は、Plan9 から移植した高機能シェルであり、現在開発途上である。
バイナリコードは /t/bin/rc である。qsh のなかからも、以下のように起動できる。

```
LP49[/]: cd /t/bin
LP49[/t/bin]: rc
```

To be explained.

第VI部

サーバ

第21章　代表的なサーバー

(1) RAMFS サーバー (src/cmd/simple/ramfs.c)

ストレッジメディアとして RAM を使ったシングルスレッドのファイルサーバーである。簡単なので、サーバの作り方の勉強にも役立とう。

(2) DOS サーバー (src/cmd/dosfs/*)

DOS ファイルシステムをサポートするサーバである。LP49 ブートフロッピーは DOS ファイルであり、ブートフロッピーに載っているコマンドをすぐに使えるように、本サーバは LP49 のブート時に立ち上がっている。

(3) EXT2 サーバー (src/cmd/ext2fs/*)

Linux の標準ファイルシステムである EXT2 ファイルシステムをサポートするサーバ。

(4) 9660 サーバー (src/cmd/9660fs/*)

ISO 9660 の CD ファイルシステムをサポートするサーバ。

(5) ノード間連携: リソースの exportfs/import

自分の名前空間（Name directry）の部分空間を、外部ノードに見えるように (export)、あるいは外部ノードの部分空間を自分の所から見えるようにする (import) 機能である。

○ src/cmd/exportfs，src/cmd/import

○自分の実行環境を相手に見えるようにする

○ export された環境をアクセスできる。

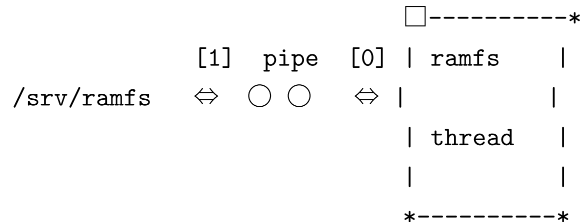
○例えば、terminal PC が CPU サーバに自分のリソースを見せて仕事をさせる。

(参考）QSH からのサーバの制御

（1） ramfs

○ ramfs をスタートさせる。

○ ramfs は、パイプ（Plan9 の pipe は両方向）を作る。



○パイプのサービス受け側を、/srv に登録する。→ /srv/ramfs が生成される。

プログラム上では /srv/ramfs を生成して、パイプの file descriptor 番号を書き込む。

```
pipe(p);
.....
fd = create("#s/ramfs", OWRITE|ORCLOSE, 0666);
sprint(buf, "%d", p[1]);
write(fd, buf, strlrn(buf));
```

→ /srv/ramfs の本当の中身は、 このパイプを指す channel 構造体。

○勿論、離れたマシン間では パイプの代わりに ネットワーク接続を使う。

（2） mount -c /srv/ramfs /tmp ""

○ /tmp に /srv/ramfs のルートディレクトリ（""）をマウントする。

○ APL が /tmp/* にアクセスすると、自動的に 9P protocol に変換され、パイプ経由で ramfs がアクセスされる。

（3） create -w /tmp/zzz 0777

○ /tmp(つまり ramfs) 上に zzz ファイルを、writable(-w)，permisson=0777 で生成する。

○ OS は ，/tmp/zzz の file descriptor を返す。 ここでは 3 とする。

（4） write 3 $sl

○ $sl は、SL を描いた QSH 内のファイル

○ /tmp/zzz に SL の絵が書き込まれる。

○ file position が進む。

（5） pread 3 0

○ 先頭番地から内容が読み出される。

第22章　u9fsサーバ: Linuxのファイルシステムをマウントする

第23章　サーバプログラムの作り方

23.1　Ramfs2 サーバのソースコード： src/cmd/lesson/ramfs2.c

サーバの共通処理は "lib9p" ライブラリが用意しているので、サーバは簡単に作ることができる。
以下は、RAM を記録媒体とするファールサーバである。これは、配布ファイルの
LP49-yymmdd/src/cmd/lesson/ramfs2.c である。

【src/cmd/lesson/ramfs2.c のソース】

```
#include <u.h>
#include <libc.h>
#include <auth.h>
#include <fcall.h>
#include <thread.h>
#include <9p.h>

static char Ebad[] = "something bad happened";
static char Enomem[] = "no memory";

typedef struct Ramfile Ramfile;
struct Ramfile {
    char *data;
    int ndata;
};

void fsread(Req *r)
{
    Ramfile *rf;
    vlong offset;
    long count;

    rf = r->fid->file->aux;
    offset = r->ifcall.offset;
    count = r->ifcall.count;
    if(offset >= rf->ndata){
         r->ofcall.count = 0;
         respond(r, nil);
         return;
    }
    if(offset+count >= rf->ndata)
        count = rf->ndata - offset;
    memmove(r->ofcall.data, rf->data+offset, count);
    r->ofcall.count = count;
    respond(r, nil);
}

void fswrite(Req *r)
{
    void *v;
    Ramfile *rf;
    vlong offset;
    long count;

    rf = r->fid->file->aux;
    offset = r->ifcall.offset;
    count = r->ifcall.count;
    if(offset+count >= rf->ndata){
        v = realloc(rf->data, offset+count);
        if(v == nil){
             respond(r, Enomem);
```

```
			return;
		}
		rf->data = v;
		rf->ndata = offset+count;
		r->fid->file->_dir.length = rf->ndata;  //% _dir.
	}
	memmove(rf->data+offset, r->ifcall.data, count);
	r->ofcall.count = count;
	respond(r, nil);
}

void fscreate(Req *r)
{
	Ramfile *rf;
	File *f;

	if(f = createfile(r->fid->file, r->ifcall.name, r->fid->uid, r->ifcall.perm, nil)){
		rf = emalloc9p(sizeof *rf);
		f->aux = rf;
		r->fid->file = f;
		r->ofcall.qid = f->_dir.qid;  //% _dir.
		respond(r, nil);
		return;
	}
	respond(r, Ebad);
}

void fsopen(Req *r)
{
	Ramfile *rf;

	rf = r->fid->file->aux;
	if(rf && (r->ifcall.mode&OTRUNC)){
		rf->ndata = 0;
		r->fid->file->_dir.length = 0;   //% _dir.
	}
	respond(r, nil);
}

void fsdestroyfile(File *f)
{
	Ramfile *rf;

	rf = f->aux;
	if(rf){
		free(rf->data);
		free(rf);
	}
}

Srv fs = {
	.open=  fsopen,
	.read=  fsread,
	.write= fswrite,
	.create= fscreate,
};

void usage(void)
{
	fprint(2, "usage: ramfs2 [-D] [-a address] [-s srvname] [-m mtpt]\n");
	exits("usage");
}

void main(int argc, char **argv)
{
	char *addr = nil;
	char *srvname = nil;
	char *mtpt = nil;
	Qid q;

	fs.tree = alloctree(nil, nil, DMDIR|0777, fsdestroyfile);
	q = fs.tree->root->_dir.qid;  //% _dir.
```

```
    ARGBEGIN{
    case 'D':
        chatty9p++;
        break;
    case 'a':
        addr = EARGF(usage());
        break;
    case 's':
        srvname = EARGF(usage());
        break;
    case 'm':
        mtpt = EARGF(usage());
        break;
    default:
        usage();
    }ARGEND;

    if(argc) usage();
    if(chatty9p)
        fprint(2, "ramsrv.nopipe:%d address:'%s' srvname:'%s' mtpt:'%s'\n",
               fs.nopipe, addr, srvname, mtpt);
    if(addr == nil && srvname == nil && mtpt == nil)
        sysfatal("must specify -a, -s, or -m option");

    // "-a addr" が指定された場合は、NW から接続要求を待つ
    // listensrv_l4() の中でサービススレッドが生成される
    if(addr)
        listensrv_l4(&fs, addr);   //%

    // "-s srvname" が指定された場合は/srv にサーバ登録をする
    // "-m mountpoint" が指定された場合はマウントまで行う
    // postmountsrv_l4() の中でサービススレッドが生成される
    if(srvname || mtpt)
        postmountsrv_l4(&fs, srvname, mtpt, MREPL|MCREATE); //%
    final_l4();  //%  exits(0);
}
```

23.2 ramfs2 の起動と利用

(1) サーバ登録の例

```
LP49[/]: /bin/ramfs2 -s ramfs2
    -->  /srv/ramfs2 が作られる
LP49[/]: mount -a /srv/ramfs2  /mnt
    -->  /mnt にマウントされる。
```

(2) マウントの例

```
LP49[/]: /bin/ramfs2 -m  /mnt
    -->  /mnt にマウントされる。
```

(3) NW 上に公開

```
LP49[/]: /bin/ramfs2 -a tcp!*!2345
    -->  ポート  2345 をアナウンスする。

----- 別マシン -------
LP49[/]: /bin/srv2  tcp!10.0.0.2!2345 ramfs2
    -->  /srv/ramfs2 が作られる
LP49[/]: mount -a /srv/ramfs2  /mnt
    -->  /mnt にマウントされる。
```

第24章　Serverのための基本技術

24.1　多重処理の実現

(1) 9P プロトコルの多重化識別子
サーバーは複数プロセスから同時に要求を受理する可能性がある。多重処理の実現法としては、一般に以下が考えられる。

(2) 多重処理
サーバは。9P プロトコルのメッセージを処理できるユーザモードのプロセスである。9P メッセージは TCP 接続あるいは pipe を通して運ばれる。サーバにつながれた TCP 接続および pipe をサーバリンクと呼ぶ。サーバリンクは CORE 層のサービス記録簿（/srv/*) に登録される。
クライアントはサーバ登録簿から目的サーバを見つけて自分の名前空間にマウントすることにより、サーバの名前空間にアクセスできるようになる。サーバは、複数のメッセージを並行して受ける可能性があり、大別して二つの作り方が考えられる。

24.2　多重処理の実現法

サーバーは複数プロセスから同時に要求を受理する可能性がある。多重処理の実現法としては、一般に以下が考えられる。

1. シングルスレッドによる逐次実行
   同時には、一つのメッセージのみを処理する。処理中に到着した次のメッセージは、前者が終わるまで待ち合わせされる。.....
2. シングルスレッドによる多重処理
   ....
3. マルチスレッドによる多重処理
   複数の並行要求を処理するために、複数ののスレッドを用意して，個々の要求を書誌する方式。スレッドはスレッドプールから割り当てるなり、動的に生成するなりする。.....
4. プロセス生成
   ....

24.3 サーバのクライアントへの見せ方 (その 1 パイプ)

§ Pipe を/srv に登録、マウント

◎ Cf. src/lib9p/srvmain-l4.c srv.c ◎ 立ち上げ内容

【参考】Plan9 オリジナル◎ Cf. src/lib9p/rfork.c post.c srv.c ◎ 立ち上げ内容

24.4 サーバのクライアントへの見せ方 (その 2 TCP 接続)

§ Listen() して L4 スレッド生成

◎ Cf. src/lib9p/srevmain-l4.c , srv.c

◎ 立ち上げ内容

第VII部

今後の課題

第25章　今後の課題

25.1　進行中

- rc シェルの完成 (進行中)
- rfork(), exec(), exits() の安定化
- exportfs/import の完成
- exception handler の完成
- L4 マルチスレッドライブラリのリファイン
- VGA ドライバ (とりあえず qemu の -std-vga)
- 資料化
- 評価

25.2　Wish リスト

- USB 2.0
- USB ブート
- VGA サポート
- IBE を使った認証
- HTTPD などの Plan9 コマンドの移植
- メモリ割り付けのリファイン (CORE 層の malloc、libc の malloc)
- Pager (src/9/hvm/mx-pager) の書き直し
- プロセス制御機能を CORE 層から HVM 層に移す。
- 難解プログラムの書き直し chan.c, devmnt.c,,,
- 関数型言語 Erlang による OS 記述
- 評価