分散 OS 開発学習キットとしての LP49

H_2O

平成 22 年 3 月 5 日

概 要

Control systems (e.g. embedded systems, home servers) are becoming more and more sophisticated, networked and complex. Software productivity and dependability are the first concern of their design. OS ' s for control systems must support distributed processing, fail-robustness and ease of program development. LP49 is a component-oriented OS with micro-kernel and multi-server architecture. We have adopted the L4 micro-kernel because of its performance and flexibility. Plan 9 had devised excellent distributed processing facilities (e.g. 9P protocol, private name space, user-mode servers), and we have largely adopted concepts and source code from Plan 9. LP49 component architecture will be effective to improve dependability.

1 はじめに

今や組み込みシステムやホームサーバには、NW 接続された多様な機器を連携動作させる分散処理機能が必須である。OS はシステムの性能 (機能・効率・信頼性) のみならずプログラム開発コスト (開発しやすさ、開発期間・拡張性など) を大きく左右する。信頼性向上の近道はシンプル化であり、目的に最適な分散 OS を作りたい人にはシンプルな分散 OS 開発キットは意義がある。

また、OS はソフトウェア技術の集大成であり、最高のソフトウェア教材である。学習用 OS は、重要機能を含み、構成が簡明で、容易に機能追加でき、プログラム規模も小さいことが望まれる。学習用 OS としては、Minix が大変すぐれているが、残念ながら分散処理機能は目的としていない。

このように、分散 OS の適切な開発学習キットは大変意義が高い。本報告は、以上の観点から分散 OS 開発学習キット LP49 について紹介する。

2 本 OS 開発学習キットの意図

シンプルな分散 OS 開発学習キットに向けて、以下を行っている。

組込みシステムに適したシンプルなＯＳ 組み込みシステムに必要な機能をもったコンパクトな OS。

耐障害性 部分障害が生じても、システムクラッシュせず、障害部分だけの再開でサービスを継続できるようにする。このためには、マイクロカーネル＋ユーザモードのマルチサーバ構成とする。

拡張性の強化 機能追加は、OS カーネルを修正せず、できるだけ外付けプログラムで行えるようにする。

分散処理・連携機能 分散リソースの融通性の高い管理と制御、別ノードの名前空間の可視化、実行環境の連携、などを実現する。

プログラム開発の容易化 カーネルモードプログラムは開発が困難である。マイクロカーネル以外は、デバイスドライバも含めてユーザモード化する。

L4 と Plan9 のソースコード活用 OS 全体をスクラッチから作るには、膨大な工数を要する。Karlsruhe 大学の L4 マイクロカーネルは、簡潔で優れたスレッド・メッセージ性能を持っている。また、Bell 研で開発された Plan9 は、融通性の高い分散処理を実現している。かつ両者ともオープンソースであるので、ソースコードを活用して工数削減をはかった。

使い慣れたプログラム開発環境　　特別な環境が必要では敷居が高い。使い慣れた GNU 環境でコンパイルからテスト走行まで出きるようにする。

3　基本発想

3.1　マイクロカーネル

OS の疎結合モジュール化、デバイスドライバを含むプログラムのユーザモード化、個別プロセス再開による耐障害強化、マルチスレッドプログラミングの容易化の観点からマイクロカーネル型 OS とした。マイクロカーネルは、スレッドとメッセージの性能が優れ、かつシンプルな L4 を利用した。

3.2　マルチサーバー構成

OS 全体を管理するモジュールは、L4 マイクロカーネルの 1 タスク (論理空間+スレッド) としてユーザモードで実行させている。これを “LP49-CORE” と呼ぶ。

3.3　すべてのリソース・サービスをファイルトリーとして扱う

Unix では、すべてのリソースが階層的ファイルトリー上に名前付けされており（名前空間)、ファイルとしてアクセスできることを目指したが、この理念はネットワーク (NW) を始めとして早期に破綻した。LP49 では Plan9 を踏襲して、NW も含めてすべてアクセス対象を名前空間 (ディレクトリー) 上の名前で識別でき、ファイルインタフェース (open(), read(), write(), 等) で操作されるオブジェクトとして扱っている。つまり、Plan9/LP49 では、一般のファイルシステムのみならず全てのリソースやサービスも、操作法はファイルシステムとして統一されており、各々の名前空間を有している。

例えば, インターネットサービスは以下の様にディレクトリを形成している。ここに /tcp/0, /tcp/1,,, は個々のコネクションを表す。

```
--+-- tcp/ -----+-- clone
  |             |-- stats
  |             |-- 0/ ---+-- ctl
  |             |         |-- data
  |             |         |-- local
  |             |         :
  |             |
  |             |-- 1/ ---+-- ctl
  |             |         |-- data
  |             :         |-- local
  |                       :
  |
  |-- udp/ -----+-- clone
  |             |-- stats
  |             |-- 0/ ---+-- ctl
  |             |         |-- data
  |                       |-- local
  |                       |
  |--- arp/ ----
  :
```

3.4　サービス部品：サーバとサーバント

OS が提供するサービスには、DOS ファイルサービス、EXT2 ファイルサービスといった高位サービスと、ハードウェア駆動、モジュール間通信、NW 接続、サーバ登録簿といった低位サービスとがある。

前者は、個々に独立性が高く、規模も大きくなりがちなので、サービス毎にユーザモードで走るプロセスとして実現する。これを**サーバ**と呼ぶ。サーバはメッセージインタフェースなので、ローカルでもリモートでも同等に使える。ユーザモードプロセスなので、プログラム開発の容易化のみならず、障害時もそのサーバだけを停止・再開することで耐障害性も強化される。

後者は、共通部品的であり、より実行速度が重視されるので、独立したプロセスとはせず LP49-CORE 内のモジュールとして実装することとした。このモジュールを**サーバント**と呼んでいる。サーバントは、統一インタフェースを持つコンポーネントである。機能的には、同一サービスをサーバで実装することもサーバントで実装することも可能である。

3.5 マルチサーバと 9P プロトコル

サーバのプロトコル（メッセージインタフェース）は、サーバが提供できる機能と性能を決定する。独自プロトコルは、いくら強力であっても普及させることは至難である。Plan9 の **9P プロトコル**は、`attach()`, `walk()`,`open()`, `create()`, `read()`, `write()`, `clunk()`, `remove()`, `stat()`, `wstat()` 等のメッセージからなり、低レベル制御も可能で融通性が高いので、これを採用した。9P プロトコルを採用した副産物として、少ない修正で Plan9 のサーバを LP49 に移植することも可能になった。

3.6 名前空間とその接続

前述のように、サーバもサーバントもリソースはファイルとして抽象化されており、自分の名前空間を持つので、UNIX でいうファイルシステムである。

図 1 に示すように、ルートファイルシステム (**RootFS**: 実はこれもサーバント) を出発点とする**名前空間**に、サーバやサーバントを接続（マウント）することにより、プロセスに見えるようになる。“/dev” には各種デバイスサーバントが、“/net” にはプロトコルスタックが接続されている。サーバはリモートの可能性もあるので、マウントの仕組みは後で説明する。

3.7 プロセス個別名前空間

UNIX ではファイルシステムの mount は root のみが行え、名前空間は全プロセスで共通である。これに対し、Plan9/LP49 の名前空間は、各プロセス毎に自前の名前空間（**個別名前空間**）を持つことができる。名前空間はアクセス保護の役目を持つので、緻密なセキュリティー管理を実現できる。

4 本 OS の構造 概要

4.1 階層構造

図 2 に示すように、LP49 は以下の階層からできている。

1. L4 マイクロカーネル
   L4 マイクロカーネルが、スレッド制御、タスク空間 (プロセス相当) 制御, スレッド間通信、ページ制御、割り込み検出と通知を行っている。マイクロカーネルのみがカーネルモードで走り、それ以外はデバイスドライバも含めてユーザモードで走る。

2. HVM
   LP49 のスタートアップと、page-fault を処理するページャー (Pager) スレッドの機能を持っている。将来、VM 機能を追加して Hypervisor Module 化する予定で、HVM と名づけた。

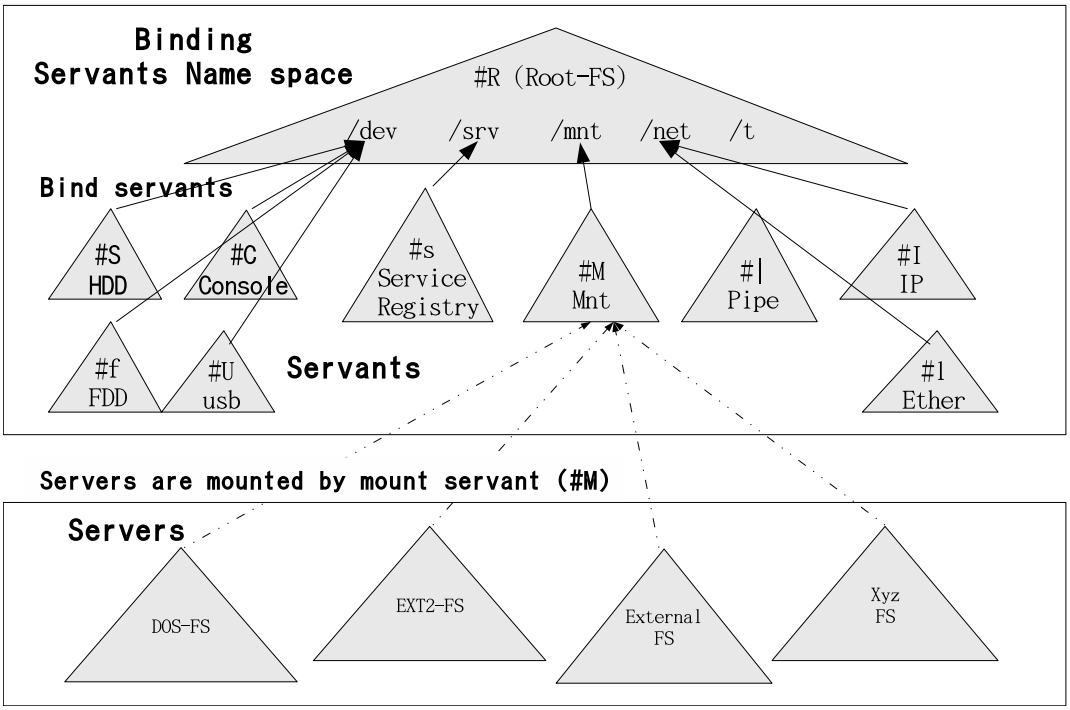


図 1: サーバ、サーバントと名前空間

3. LP49-CORE
   応用プログラム (以下 APL と呼ぶ) のシステムコールを受けて, 適切なサーバやサーバントに処理を行わせる。ネットワークサービスや、デバイスドライバも含まれる。

4. サービス階層
   OS サービスを行うサーバー類も普通の応用プログラム (APL) も、同等のユーザモードプロセスである。サーバは、9P プロトコルを喋れる APL である。

4.2 サーバント

サーバントはハードウェアデバイス, サーバ登録簿、環境変数記憶、pipe, プロトコルスタックなど低位サービスを提供する。サーバントは LP49-CORE プロセス内のモジュールであり、attach(), init(), open(), read(), write(),,, といったプロシージャインタフェースで呼ばれる。

サーバントもいわゆるファイルシステムであり、自分の名前空間をもつ。つまり、サーバント内の各要素はディレクトリ (つまり名前空間) で名前付けされており、ファイルインタフェースで操作できる。サーバントは #+<英字> の形式のサーバント名をもつ。代表的サーバントを以下に示す。括弧内は サーバント名である。

```
コンソール (#c), ハードディスク (#S), フロッピーデバイス (#f),
環境変数 (#e), サーバ登録簿 (#s), Remote Procedure Call (#M),
Ether ドライバ (#l), プロトコルスタック (#I), Pipe (#|),
ルートファイルシステム (#R), USB ホストコントローラ (#U),
VGA コントローラ (#v)、    ...
```

“bind” コマンドにより、サーバントをプロセスの名前空間に結合することにより、プロセスからサーバントにアクセスできるようになる。

```
bind [-abc]  サーバント名  マウントポイント
```

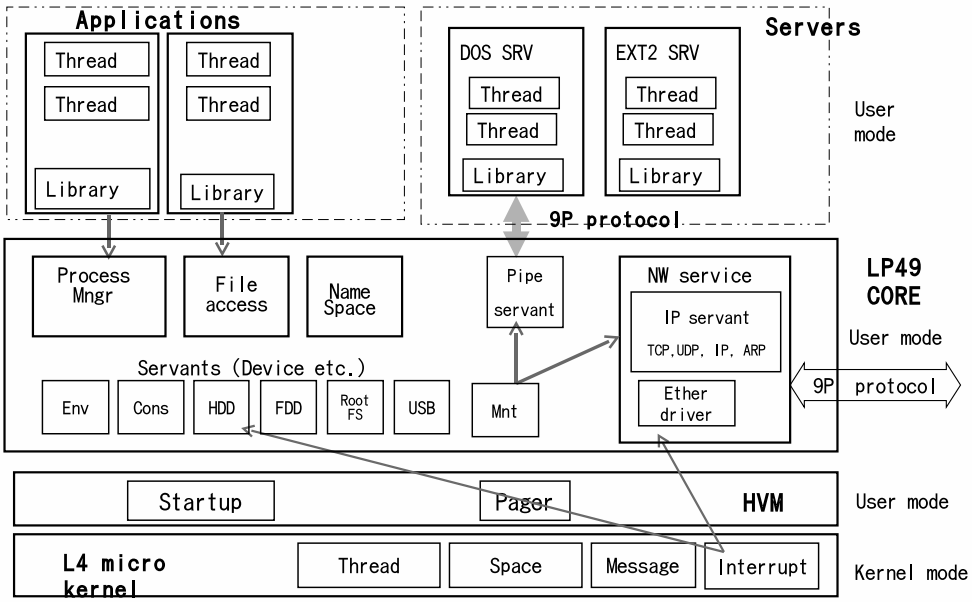


図 2: LP49 全体構成

4.3 サーバ

サーバはサービスごとに独立したユーザモードのプロセスであり、9P メッセージ (9P プロトコル) を受信してサービスを実行する。LP49-CORE とサーバの間で 9P メッセージを運ぶ接続を**サーバリンク**と呼んでいる。サーバリンクは、ローカルサーバの場合は pipe (LP49 の pipe は 双方向である)、リモートサーバの場合は TCP/IP 接続を用いる。

サーバは自分のサーバリンクを “サーバ登録簿” に登録しておく。サーバ登録簿はサーバントの一つ (#s) で、 立ち上げ時に “/srv“ として接続されている。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、サーバリンク名 (ex. /srv/dos) を自分の名前空間にマウントする。これにより、サーバの名前空間がクライアントの名前空間に接続され、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。

```
mount [-abc]  サーバリンク名  マウントポイント  場所
```

4.4 システムコールの仕組み

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 LP49-CORE 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 4 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 LP49-CORE に送り、返答メッセージを待つ。APL と LP49-CORE 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。

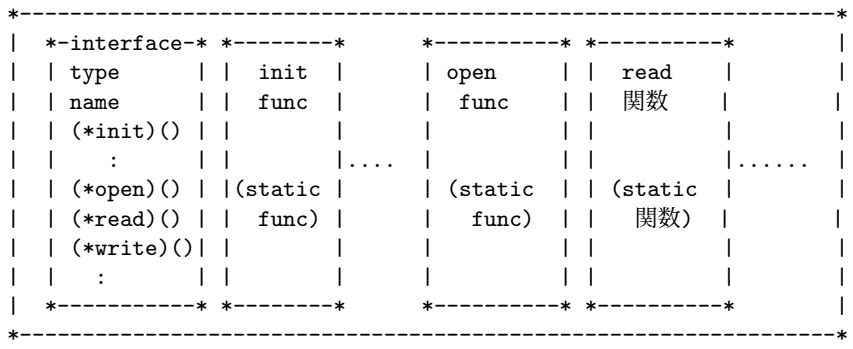


図 3: サーバントの構造

(2) LP49-CORE マルチスレッドサーバ

LP49-CORE の位置付けは、プロセスに対してはサーバであり、サーバプロセスに大してはクライアントといえる。

システムコールの処理は中断が生じうる。複数の APL からの要求を並行して処理するために、マルチスレッドサーバを実装した。

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 4 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 4 に示すように clerk スレッドは **マウントサーバント** を呼ぶ。マウントサーバント (#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。

4.5 Pager の仕組み

LP49 のプロセスは論理空間で保護されている。L4 マイクロカーネルでは、スレッドの実行中に pagefault が生じると、登録されている **Pager** スレッド に Pagefault メッセージが送られる。Pager は L4 ユーザがプログラムできるスレッドであり、Pagefault メッセージの引数を見て適切なページをマップするなど、自前の論理を組み込むことができる。L4 スレッドは、生成時に Pager を指定しておく。

Pager は pagefault メッセージに応じたメモリマップを考えて L4 の map 機能を行うだけの普通のスレッドである。LP49 の 通常の Pager は HVM 階層に含まれているが、LP49 ユーザが独自 Pager を追加することも可能である。

現在の Pager は基本機能のみ実現しているが、Pager は大変融通性に富んでおり、Pager の機能強化により次のような色々な展開が可能である。

ページキャッシュ .....

ランザクションメモリ ....

共用メモリ ....

性能向上 ....

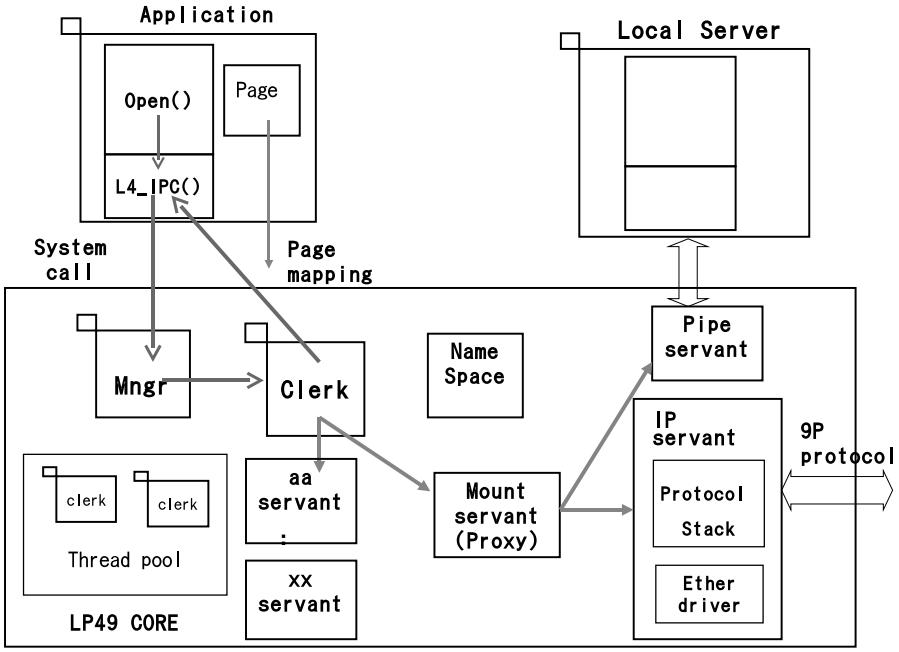


図 4: システムコール

5　分散処理と名前空間

5.1　名前空間の機構

名前空間の構成法を図 5 に示す。LP49-CORE は、ルートファイルシステム (RootFS) を有している。図中の (a) では、サーバントを RootFS の適当なマウントポイントに結合 (bind) することにより、サーバントの名前空間がホストの名前空間に接続され、サーバントのサービスを受けられるようになる。サーバントは、procedure call で呼び出されるので、自ホスト内に限られる。

同様に (b) では、サーバーをマウント (mount) することにより、サーバの名前空間がホストの名前空間に接続され、サーバのサービスを受けられるようになる。サーバは remote procedure call で呼び出されるので、自ホスト内 (b) でもリモートホスト上 (c) でも、まったく同様にアクセスできる。

また、マウントはサーバ単位だけではなく、図 5-(d) の remote-host の名前空間の”部分名前空間”を別のホストのマウントポイントにマウントすることも可能である。この仕組みについては、後で説明する。

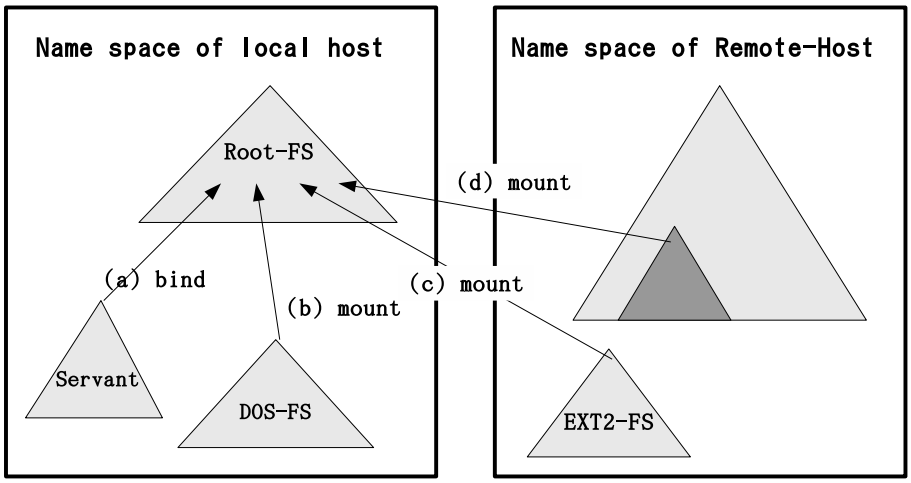
5.2　サーバ登録簿とサーバのマウント

前述のように、サーバと LP49-CORE の間で 9P メッセージを運ぶ接続がサーバリンクである。サーバリンクは、ローカルサーバの場合は pipe, リモートサーバの場合は TCP/IP 接続である。

サーバ登録簿は “#s” という名前のサーバントであり、初期設定により名前空間の “/srv” に接続されている。各サーバは、サーバリンクをサーバ登録簿サーバント (/srv/*) に登録する。例えば EXT2 ファイルサーバは、“/srv/ext2” というファイルを作って、そこにサーバリンクの file descriptor を書き込んでおく。

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アクセスできるようになる。

```
[例]    mount -a /srv/dos /c /dev/sdC0/dos
```



(a) A servant is mounted onto my name space.
(b) A local server is mounted onto my name space.
(c) A remote server is mounted onto my name space.
(d) A sub name space of remote-host is mounted onto my name space.

図 5: 名前空間とマウント

これは、DOS ファイルサーバ (“/srv/dos”) を使って、HDD (“/dev/sdC0”) の DOS partition を “/c” にマウントしている。

5.3 プロトコルスタック

NW (network) 機能は OS の最重要機能の一つであり、OS 技術者/研究者はプログラム構造とロジックを完全にマスターしておくべきである。Plan9 の NW プログラムは, stream, xkernel などの研究成果が反映され簡明である、Plan9 用に開発された Ether ドライバを少ない修正で流用できる、ことから、LP49 は基本的に Plan9 を継承している。

ネットワーク関連のプログラム構成を、図 6 に示す。IP サーバント (“#I”) を介して、TCP, UDP, IP など各プロトコル毎にモジュール化されたプログラムが動作している。IP サーバントは Internet サービスのインタフェースであり、ユーザには以下のトリー構成をもつファイルシステムとして見える。　IP サーバントは普通 “/net” に接続するので、“/net/tcp” は TCP を, “/net/udp” は UDP を意味する。個々の接続も名前空間に現れ、例えば /net/tcp/0, /net/tcp/1 ,,,　は、TCP 接続を表す。

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

Ether サーバント (#l) は、 Ether card のサービスインタフェースであり、該当した Ether driver を呼び出している。

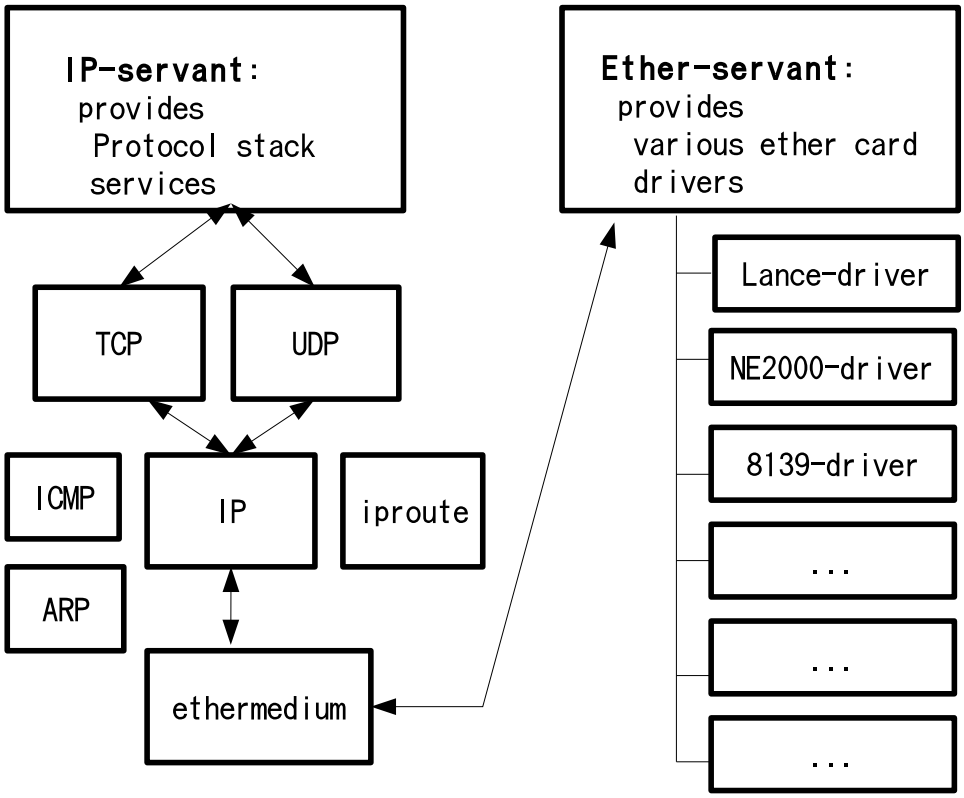


図 6: IP サーバントと Ether サーバント

5.4 名前空間の export/import

リモートホストの名前空間の “部分” 名前空間をマウントすることも可能である。図 5 の (d) は、リモートホストの部分名前空間をマウントしている例である。

例えばリモートホストの /dev ディレクトリーをローカルホストにマウントすると、/dev に接続されているリソースにアクセスすることが可能になる。全てのオブジェクトはファイルインタフェースで操作できるので、このことはリモートホストのデバイスも操作できることを意味する。

同様にして、Plan9 ユーザグループから移植した U9FS というプログラムを Unix 上で走らせることにより、UNIX のファイルシステムの部分空間を LP49 上にマウントすることも可能である。

6 能動オブジェクトライブラリ

6.1 目的

本ライブラリ libl4thread は、L4 マイクロカーネル上で以下の機能を提供するものである。

能動オブジェクト（並行オブジェクト）支援： 能動オブジェクト（自前のスレッドを持ち、他と並列実行できるオブジェクト) を容易に実現できる。

分散処理； リモートノードとのメッセージ交信を容易に行えるようにする。

非同期メッセージ： L4 は同期メッセージのみを提供しているが、分散処理のためには非同期メッセージも望まれる。

Future reply： 相手スレッドに非同期メッセージで要求を送り、非同期で返される返答メッセージを受信する仕組み。

メッセージの alternative 受信：

CSP モデルサポート： Hoare の CSP モデルは、もっとも重要な並列処理モデルである。Plan9 の libthread ライブラリも CSP モデルのサポートを意図しているが、必ずしも簡明とは言えない。本ライブラリ libl4thread は、CSP 的処理も簡明に記述できることを狙っている。

マルチスレッド処理記述の容易化： L4 は融通性と効率に優れるが、使いこなすには相応のスキルを要する。本ライブラリにより、L4 の機能をより簡便に使うことができる。

L4 は、プロセス間ページマップなどの高度な機能も有しているが、使いこなすには相応のスキルを有する。本ライブラリでは、このような高度機能を容易に使えるように追々機能追加をしていく予定である。

6.2 能動オブジェクト

オブジェクト指向では、オブジェクトに何かをやってもらうにはメッセージを送る。つまり、**メッセージパッシング**することにより、そのオブジェクトのメソッドが実行される。メッセージパッシングとはいうものの、Java や C++で普通に作ったオブジェクトのメッセージパッシングは関数呼び出しに過ぎず、せいぜい実行すべきメソッドが動的に決定されるだけである。つまり、オブジェクトのメソッドは呼び出した側のスレッドによって実行される。(特に区別する場合は、このようなオブジェクトを、**受動オブジェクト**と呼ぶ。)

(メッセージパッシングという言葉に釣られて、Java や C++のオブジェクトが並列動作すると誤解している人もいるようですが、それは違います。)

能動オブジェクトとは、自前のスレッドを有したオブジェクトであり、メッセージを受けると自前のスレッドで目的メソッドを実行する。メッセージパッシングは、関数呼び出しではなく、文字通りスレッドへのメッセージ転送であり、各能動オブジェクトは並行動作できる。

能動オブジェクトのメッセージ転送の形態には、一般に次の 3 種類がある。M をメッセージ、C を送り手オブジェクト、S を受け手オブジェクトとする。

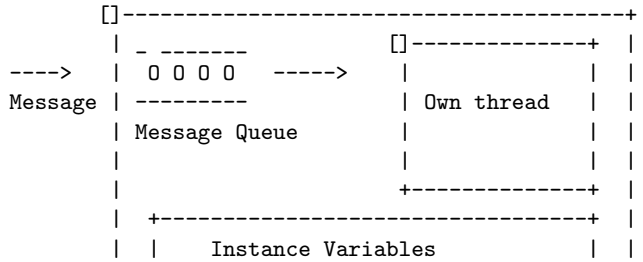
過去 (past) 型： C は S に M を送ったら、M が S によって実行されるのを待たずに、C は自分の処理を続行する。

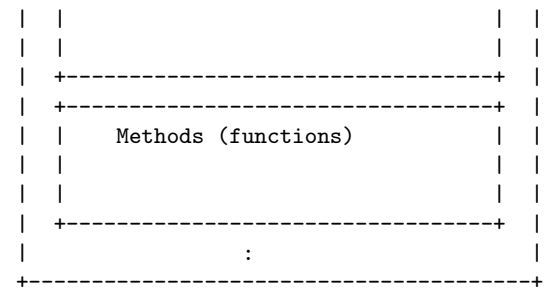
現在型 (now)： C は S に M を送ったら、S から何らかの返答が戻ってくるまで待つ。

未来型 (future)： C は S に M を送ったら、返答が戻るのを待たずに、自分の処理を続行する。S はメッセージ M を処理して、応答を未来変数に入れる。C が S からの応答を使う場合には、未来変数をアクセスする。未来変数に応答が届いていなければ、届くまで自動的にまつ。

能動オブジェクトの仕組みを下図に示す。

【能動オブジェクトの仕組み】





能動オブジェクトの送られたメッセージは、メッセージ行列に挿入される。オブジェクトの自前スレッドは、メッセージ行列からメッセージを取り出しては、指定された処理を実行する。

6.3 能動オブジェクトの定義例

能動オブジェクトのインスタンス変数域は、L_thcb域に連続して割り当てる。このために、第一要素を「L_thcb　_a;」とする構造体で定義する。

---【例】----------

```
typedef struct Alpha{                                   //[1]
    L_thcb  _a;       // L_thcb の継承に相当
    char    name[16];
    int     age;
    int     personalId;
} Alpha;

void alphafunc(Alpha *self)                             //[2]
{
   ..... thread body .....
   ....  self->age = 20; ....
}

Alpha     *a1;
L4_ThreadId_t  tid;

a1 = (Alpha*)malloc(sizeof(Alpha));                     //[3]
.... You can set value to instance vars like a1->age = 20; ...

l_create_thread(alphafunc, 4096, a1);                   //[4]
   // a1 の値は alpafunc() の第 1 引数に引き継がれる
   .....
tid = a1->_a.tid; //このようなアクセスももちろん可能
   .....
l_kill_thread(a1);                                      //[5]
free(a1);
```

[**1**] Alpha は能動オブジェクトのデータタイプを定義している。第一要素は「L_thcb　_a;」である。

[**2**] `alphafunc()` は Alpha オブジェクトのスレッドが実行する関数である。第 1 引数には、[4] で述べるように能動オブジェクトのデータ域のアドレスが引き継がれる。従って、`self->name` などとしてインスタンス変数にアクセスできる。
メッセージ受信とその処理の書き方は、後で説明する。

[**3**] 能動オブジェクトのメモリ域はユーザが割り当てを行う。

[**4**] 能動オブジェクトの生成は、第 3 引数に能動オブジェクト域を指定して、
`l_create_thread(alphafunc, 4096, a1);`
のように行う。生成が成功すれば第 3 引数の値が、失敗すれば `nil` が反る。第 3 引数の値は、スレッドが実行する関数 [2] の第 1 引数として引き継がれる。

[**5**] 能動オブジェクトの削除は `l_kill_thread()` で行う。能動オブジェクト域の解放は、ユーザ責任である。

6.4 同期メッセージの送受信

```
---【Signature】------------------------------------------------------------------
L_msgtag  l_send0(L_thcb *thcb, int msec, L_mbuf *mbuf);           //[1]
  Ex. L_mbuf *mbuf;
      mbuf = l_putarg(nil, mlabel, ‘‘i2s1’’, e1, e2, &buf, sizeof(buf));
      msgtag = l_send0(thcb, INF, mbuf);

L_msgtag  l_recv0(L4_ThreadId_t *client, int msec, L_mbuf **mbuf);  //[2]
     (第 1 引数のタイプは L4_ThreadId_t であり、l_reply0() の返答先として使われる。
      L_thcb* でないことに注意。)
  Ex. msgtag = l_recv(&client, INF, nil);
      n = l_getarg(nil, ‘‘i2s1’’, &x, &y, &buf2, &sz2);

L_msgtag  l_recvfrom0(L4_ThreadId_t client, int msec, L_mbuf **mbuf); //[3]
  Ex. msgtag = l_recvfrom(client, INF, nil);

L_msgtag  l_reply0(L4_ThreadId_t client, L_mbuf *mbuf);             //[4]
  Ex. msgtag = l_reply0(client, mbuf);

L_msgtag  l_call0(L_thcb *thcb, int smsec, int rmsec, L_mbuf *mbuf); //[5]
  Ex. msgtag = l_call(thcb, INF, INF, mbuf);
-------------------------------------------------------------------------------
```

6.5 非同期メッセージの送受信

```
---【Signature】------------------------------------------------------------------
L_msgtag  l_asend0(L_thcb *dest, int msec, L_mbuf * mbuf);     //[1]
     (第 2 引数 msec は、無効果である。l_send0() との対称性のため)
  Ex. mbuf = l_putarg(mbuf, mlabel, ”i2s1 ”, e1, e2, &buf, sizeof(buf));
      msgtag = l_asend0(dest, 0, mbuf);
```

```
L_msgtag  l_arecv0(L_thcb  *mbox , int msec, L_mbuf **mbuf);    //[2]
```
○ 第 1 引数 mbox が nil の時は、自スレッドが持つメッセージボックスから受信する。\\
非 nil の場合は、mbox が指定するメッセージボックスから受信する。

○ 引数 ”msec ” は、 INF (-1) または 0.
INF: メッセージが到着するまで待つ。
0: すでにメッセージが到着していなければ、即戻る。
○ 受信メッセージバッファの L_mbuf.sender に、送り主の L4_ThreadId_t が入っている。

```
Ex. L_mbuf *mbuf;
    L4_ThreadId_t  client;
    mbuf = l_arecv0(nil, INF, &mbuf); // 自メッセージボックスから受信
    mbuf = l_arecv0(mbox, INF, &mbuf); // mbox が指すメッセージボックスから受信
    n = l_getarg(mbuf, ”i2s1 ”, &x, &y, &buf2, &sz2);
    client = mbuf->X.mr1;

L_msgtag  l_areply0(L4_ThreadId_t client, L_mbuf * mbuf);       //[3]
```
（注意）第 1 引数は L4_ThreadId_t である。L_thcb* タイプではない。
```
Ex. mbuf = l_putarg(mbuf, mlabel, ‘‘i1s1’’, e1, &buf, sizeof(buf));
    n = l_asend0(client, mbuf);

int  l_arecvreply0(int rectime, L_mbuf **mbuf, int tnum, ...); //[4]
```
○ 非同期返答の受信は、次節で説明する。

```
L_msgtag  l_acall0(L_thcb  *thcb, int recmsec, L_mbuf *mbuf);   //[5]
Ex. rc = l_acall0(dest, INF, INF, mbuf);

-------------------------------------------------------------------------
```

6.6 非同期返答メッセージ

(1) 非同期返答メッセージの返送と受理

```
---【Signature】-----------------------------------------------------------------

L_msgtag  l_areply0(L_thcb * mbuf);       //[3]
```
○ 第 1 引数は、L4_ThreadId_t タイプである。 L_thcb* ではない。
```
Ex. mbuf = l_putarg(mbuf, mlabel, ‘‘i1s1’’, e1, &buf, sizeof(buf));
    n = l_asend0(client, mbuf);

int  l_arecvreply0(int rectime, L_mbuf **mbuf, int tnum, ...); //[4]
```
○ 第 1 引数：(== 0) 待たない。既に返答が到着していれがそれが返される。
(!= 0) 返答が届くまで待つ。
○ 第 2 引数: ここに返答の載ったメッセージのアドレスが代入される。

○ 第 3 引数：Tally（割符）の数
○ 第 4... 引数： Tally のならび。

○　返り値：マッチした Tally の番号

```
L_msgtag  l_acall0(L_thcb  *thcb, int recmsec, L_mbuf *mbuf);   //[5]
  Ex. rc = l_acall0(dest, INF, INF, mbuf);

--------------------------------------------------------------------------------
```

7　LP49 と Plan9 の対比

LP49 では、Plan9 の財産をできる限り活用するようにしたが、実施してみるとかなりの修正を必要とした。(表 1)

(1) 構造の違い

L4 を使ったマイクロカーネル構造にしたため、マルチスレッドサーバ化、システムコール, ...

(2) 言語仕様/開発環境の差

Plan9 は、独自拡張された C 言語で書かれている。特に構造体の無名フィールド (フィールド名を省略でき、その場合コンパイラがデータタイプから目的フィールドを自動的に探す）が多用されている。

(3) サーバとドライバ

サーバとドライバプログラムは、Plan9 から少しの修正で移植が可能だった。

表 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

8　本 OS キットの開発環境展

OS の開発および学習には、使い慣れた開発環境が望まれる (Plan9 が普及しない理由は独自の開発環境にもある)。また OS の走行テストも、NW 機能を含めて実機の前に仮想マシンで行えることが望まれる、

LP49 の開発環境を図 7 に示す。LP49 のプログラム開発は、Linux ホスト上の GNU ツールだけで行っている。LP49 の走行テストは、オープンソースエミュレータ QEMU の中で行っている。QEMU はネットワーク機能も提供しており、1 台のホストマシンの上で、LP49-LP49 通信、LP49-Linux 通信などを行える。1 台のホストマシン上で、コンパイルから NW を含む走行試験まで時間を待たずに (make clean; make; から LP49 立ち上げまで 40 秒）できることは、非常に好都合である。

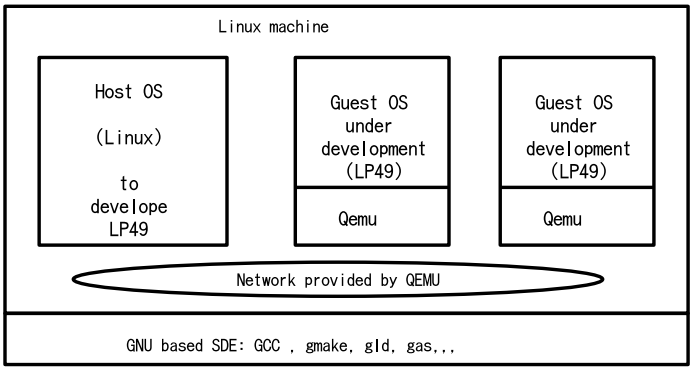


図 7: 開発試験環境

9 本 OS キットによる展開例

9.1 ソフトウェアバス

LP49 では、OS サービスはサーバもしくはサーバントによって提供される。LP49-CORE は、サーバント・サーバのマルチプレクサ、もしくは APL とサーバやサーバとの間を連携させるメッセンジャーに徹しており、これがシンプル化に役立っている。ただし、現方式ではシステムコールは全て LP49-CORE を経由して処理されるが、目的サーバが決まってしまえば、APL とサーバの間で直接メッセージをやり取りさせることもできるで、更なる簡潔化が可能である。具体的には、目的サーバの決定に関わる open(), create(), chdir()、close() を LP49-CORE が処理して APL-サーバ間を安全なチャンネルで接続させる。

また、9P プロトコルは同期型メッセージであるが、非同期型メッセージもサポートすることにより、広域分散をより効率的に行うことも可能と思われる。非同期メッセージの拡張として、サーバを関数型言語 Erlang で実装することも考えられる。

9.2 耐障害強化

プロセスを監視し、障害が見つかればそれを停止して再スタートする仕組みを容易に組み込める。また、簡単な Pager の機能追加により、プロセスに保持メモリ域を追加することができる。保持メモリ域には、プロセスが指定した適当なタイミングで無矛盾なデータのスナップショットを書き込んでおく。プロセスが障害になった場合には、保持メモリ域のデータを使って再開させることにより、比較的安全な roolback を行える。

Cf. Minix-3, Erlang

9.3 高性能化

現 LP49 には、高性能化の仕組みはほとんど組み込んでない。スレッドの優先度、システムコールの引数引き継ぎ等、容易に高性能化を図ることができる。また、Pager に機能追加をすることで、容易にページキャッシュを実現することも可能である。

9.4 認証システム

分散 OS において認証は非常に重要な課題である。
Plan9 は Kerberos を拡張した精妙な認証方式を実装しているが、学習用としては複雑である。
LP49 では, Identity-Based Encryption を使用した認証方式を検討する予定である。

10 関連研究

L4
Plan9
Minix は学習用マイクロカーネル OS として、非常に意義が高い。Linux は Minix から影響を受けて開発されたが、マイクロカーネルは採用されなかった。個人で OS を作るには、全てのカーネルデータが見えるモノリシック OS の方が作り易いし効率も良いという Linus の主張に一理はあるが、障害にたいする頑強性、プログラム保守性はマイクロカーネルが有利である。Minix は分散 OS としての機能は範囲外である。
SawMil は IBM で実施された L4 マイクロカーネルとマルチサーバからなる OS の研究プロジェクトである。Linux カーネルをサービス対応にモジュール分けしてマルチサーバ化することを狙ったが、Linux カーネルはモジュール分割が困難のため、プロジェクトは中第した。
L^4Linux は、L4 マイクロカーネルの上で Linux を動かす OS である。Linux はモノリシック構成のままであり、L4 を使った Virtual Machine といえる。
GNU の **Hurd** は、古くから挑戦しているマイクロカーネル OS である。マイクロカーネルとしては **Mach** を採用していたが、効率の観点から最近は L4 マイクロカーネルの採用を検討している。それが **L4-Hurd** である。
Unix 分散 OS ではない。

11 おわりに

LP49 の詳細な資料とソースコードは、WEB サイト http://research.nii.ac.jp/H2O/LP49 にて公開している。
(1) 学習用 OS としての観点
コメントを含むソース規模は、　HVM: 約２ K 行。LP49-CORE:　約 60K 行 (内 20K 行がプロトコルスタック)。libc ライブラリ: 約 30K 行。その他のライブラリ:x K 行。rc シェル: 約８ K 行。デバッグシェル: 2 K 行。DOSFS: 40 K 行。 EXT2FS: 30 K 行. extfs: 2 K 行である。機能に比べて充分にコンパクトであり、一人で全てをトレースできる。
本キットにより、LP49 だけでなく Plan9 と L4 のプログラム技術も理解できる。Plan9 は分散処理技術の宝庫であり、OS 技術者は熟知しているべきである。特にプロトコルスタックは他 OS に比して簡明であり、適切な教材といえる。
(2) 分散 OS 開発キットとしての観点

- 性能
  ping(LP49/qemu - LP49/qemu):4.7ms,
  ping(LP49/qemu - Linux/host):0.7ms,

remote-file-read(LP49/qemu - LP49/qemu):40ms,

CD-file-read(1KB: LP49/qemu): 690 μ sec. （初回は 25ms)
CD-file-read(1KB: 実機): 120 μ sec. （初回は 110ms)

RAM-file-read(50B: LP49/qemu): 180 μ sec.
RAM-file-read(50B: 実機): 31 μ sec.

RAM-file-read(4KB: LP49/qemu): 220 μ sec.
RAM-file-read(4KB: 実機): 48 μ sec.

- L4 Pager 機能を活用したページキャッシュを実装する予定なので、現 LP49 にはキャッシュサーバは未実装。
- システムコール
- ローカルファイルアクセス
- リモートファイルアクセス
- 分散処理機能
- 機能追加、開発のしやすさ

(3) 今後の予定