SISTEMAS DISTRIBUÍDOS

Notas de Aula

Prof. Dr. Gilberto Nakamiti

Introdução

Sistema distribuído fracamente acoplado

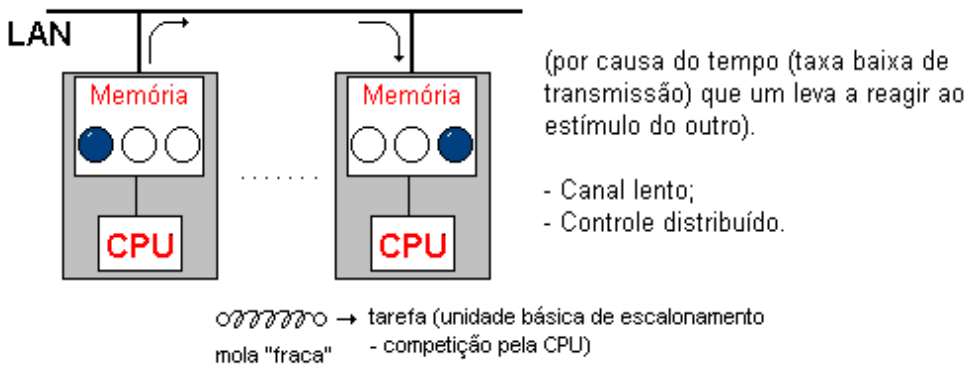


Figura 01 – Sistema fracamente acoplado

Sistema distribuído fortemente acoplado

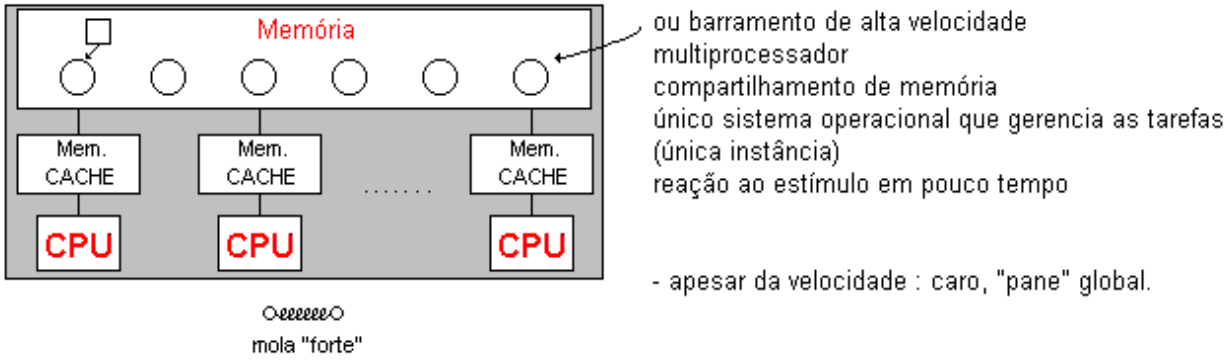


Figura 02 – Sistema fortemente acoplado

Granularidade do sistema distribuído (“grão de paralelismo”)

Pseudo-paralelismo (por Ter uma só CPU) – para sistemas fracamente acoplados.

- Grão

número médio de ciclos de CPU para executar uma tarefa
(# instruções de uma tarefa típica) fina, baixa – tarefas pequenas

- Granularidade x tempo de comunicação entre tarefas

maximizar tempo de processamento de uma tarefa t para redes
tempo de comunicação

Sistema distribuído fracamente acoplado (SDFA)

É um conjunto de N tarefas que executam em M>1 processadores conectados através de via de dados lenta (ex.: rede local). As tarefas cooperam na solução do problema para o qual o sistema foi concebido.

Motivação para o desenvolvimento de SDFA

- custo dos processadores tradicionais (ex.: estações de trabalho PCs);
- topologias de interconexão (LAN, MAN, WAN(acop. + fraco));
- diversidade de processadores – dedicados;
- confiabilidade ("graceful degeneration" – degradação amena);
- demanda tecnológica de outras áreas (ex.: projeto concorrente, área médica)

Requisitos básicos para o desenvolvimento de SDFA

- promover (fazendo uso de hardware e software):
  - comunicação entre tarefas (troca de informações)
    . ponto a ponto

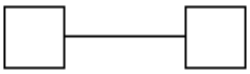


Figura 03 – Ligação ponto a ponto

. compartilhamento de informação (multiponto)

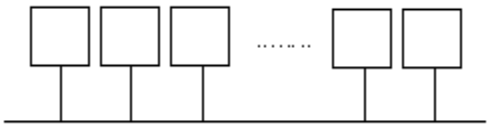


Figura 04 – Ligação multiponto

- controle entre tarefas
  . execução de tarefas (depurar, matar, ...)
  . disciplina de acesso aos recursos compartilhados
  (ex.: semáforo, transação, flags de leitura/escrita etc.)

Dificuldades para o desenvolvimento de SDFA

- separação entre tarefas
  - tarefas com espaço de endereçamento disjuntos
  - tarefas gerenciadas por sistemas operacionais distintos (pelo menos duas instâncias ou 2 SOs diferentes)
- prover transparência
  Transparência:
  - quanto ao acesso (mesmas primitivas para acessar recursos locais e remotos)
  - quanto a localização (mover um recurso entre nós (host/processador) da rede sem afetar operacionalidade)
  - quanto à concorrência (mesmo mecanismo de controle de acesso - ex.: através de determinado semáforo)
  - quanto à replicação (não se sabe com qual cópia está lidando)
  - quanto à falha (tarefas passam a ser executadas em outro nó a partir do mesmo ponto ou instâncias da mesma tarefa em nós diferentes)
  - quanto à migração (migração de tarefas para balancear cargas de processamento)
  - quanto ao desempenho (mesmo desempenho independentemente da situação / configuração do sistema)
  - quanto à escala (aumento de desempenho com aumento de recursos)

- O que controla cada HOST

aplicações soft-real-time	- sistema operacional com capacidade de comunicação inter-tarefas inter-hosts (ex.: UNIX + TCP/IP) - sistema operacional de rede (netowork OS) (ex.: UNIX ou SUN OS + NFS (Network file System) + RPC (Remote Process Call)) - (facilita a comunicação inter-tarefas inter-hosts) - sistema operacional distribuído (mgração, disparar e controlar tarefas em hosts remotos)

aplicações hard-real-time (aplicações militares, ...)	- sistema operacional para tempo real com capacidade de serviços - comunicação inter-tarefas inter-hosts

Tarefa

Tarefa é um processo.

Processo

Ë a unidade de escalonamento que possui um conjunto próprio de recursos.

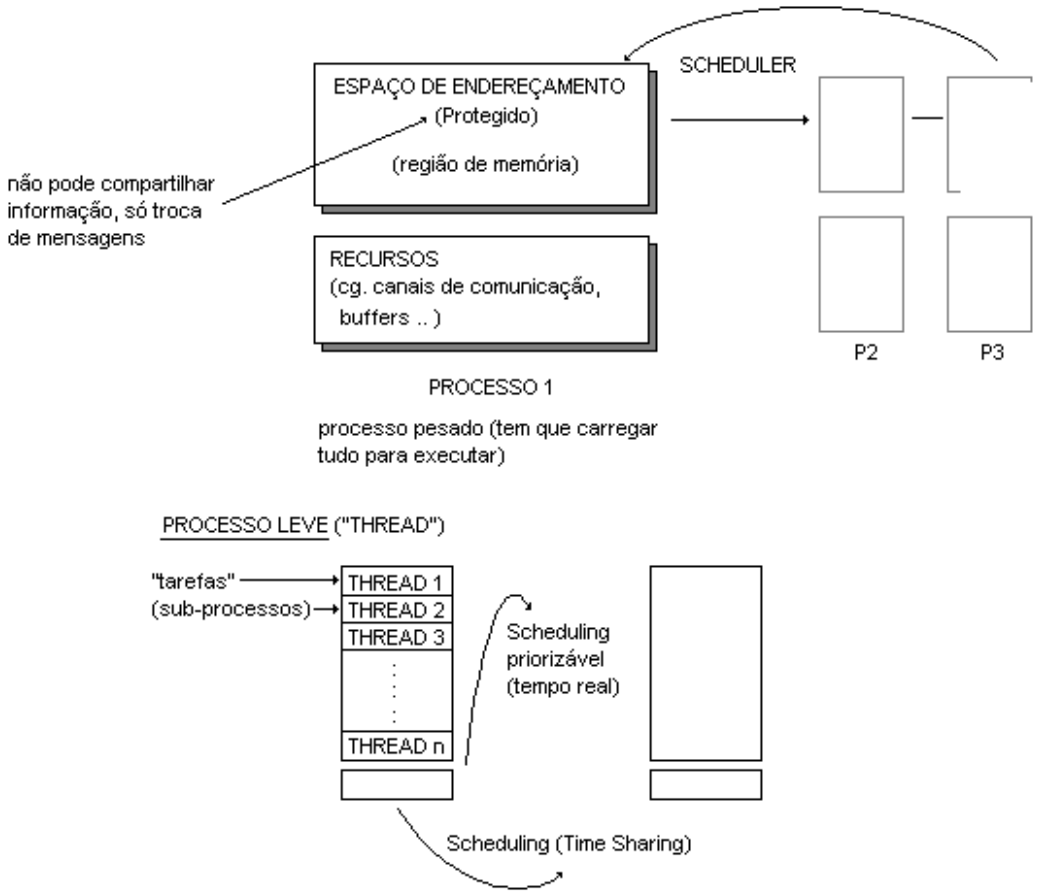


Figura 05 - Processo

Redes de computadores

Modelo OSI/ISO

O *Open System Interconection* ou Modelo de Referência para interconexão de sistemas abertos (RM-OSI/ISO) é um documento da ISO que tem como objetivo padronizar as arquiteturas de redes locais. Esse modelo consiste em sete camadas ou módulos a saber:

Camada 7	Aplicação
Camada 6	Apresentação
Camada 5	Sessão
Camada 4	Transporte
Camada 3	Rede
Camada 2	Enlace
Camada 1	Física

Camada de aplicação

Suporte a utilização de recursos distribuídos (serviços de alto nível) ex.: login remoto, transferência de arquivos, correio eletrônico, ODP (Open Distributed Processing) - Bancos de dados distribuídos.

Camada de apresentação

Transforma os dados representados internamente numa representação canônica (representação de rede). Protocolo EDP (External Data Representation).
Outras transformações:
- codificação secreta dos dados (criptografia);
- compressão dos dados.

Camada de sessão

Estabelece “conexões virtuais” entre dois processos comunicantes

Camada de transporte

Responsável pela entrega de mensagens através de circuitos virtuais ou de datagramas. A camada estabelece:
- tradução de endereço lógico para endereço físico;
- segmentação de mensagens;
- prover transmissão confiável para circuitos virtuais;

Camada de rede

Gera rotas para transmissão dos pacotes oriundos da camada de transporte (utilizada para WANs).

Camada de enlace

Transmite pacotes entre hosts efetuando detecção e recuperação de erros, controle de fluxo, etc.

Camada física

Controla os drivers de rede.

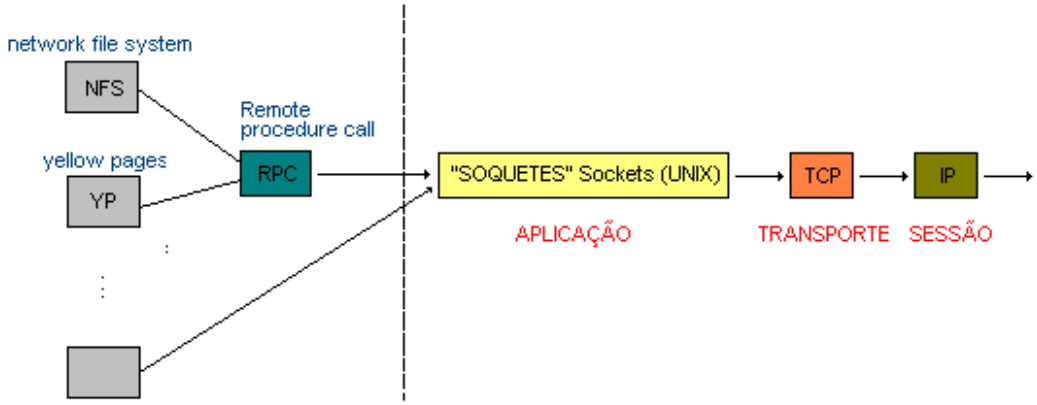


Figura 06 – Comunicação segundo o modelo OSI-ISO

Tecnologias (tipos) de redes

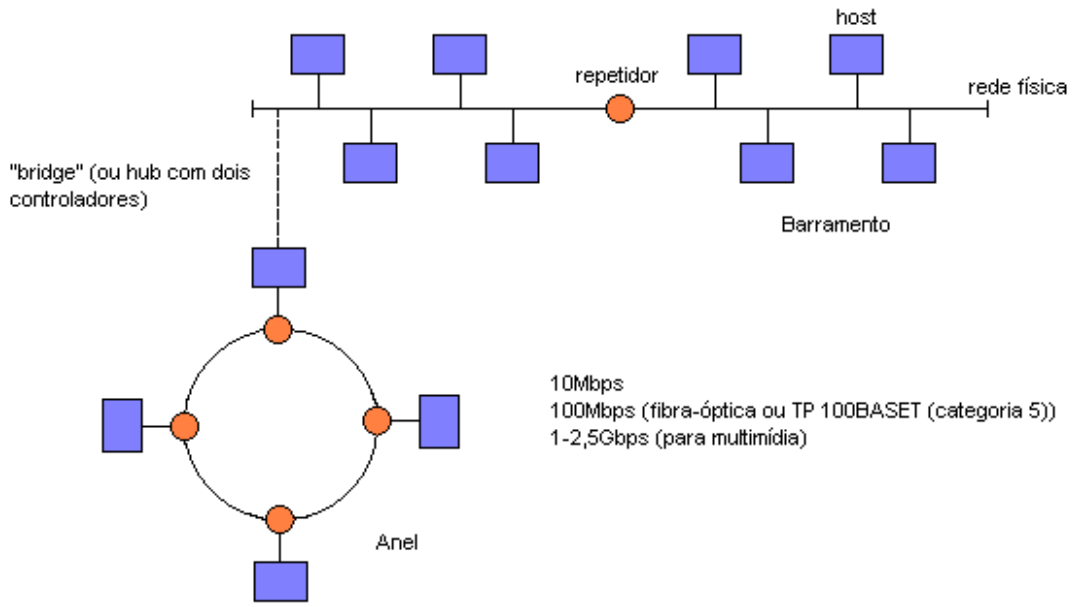


Figura 07 – Topologias de redes

Ethernet

XEROX, 1973
IEEE 802.3 CSMA/CD – "Carrier Sense multiple Access with Collision Detection"

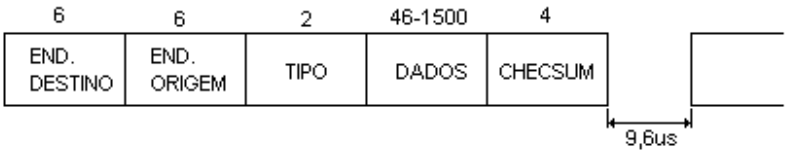


Figura 08 – Frame Ethernet

Um HOST fica permanentemente "escutando" por pacotes a ele endereçados. Para enviar um pacote:

- detecta ausência de "carrier"
- propaga o pacote
- compara o conteúdo do envio com a recepção. Se detectar colisão:
  . propaga um "jamming signal"
  . espera um tempo aleatório antes de tentar novo acesso.

Token Ring

IBM, 1969

IEEE 802.5

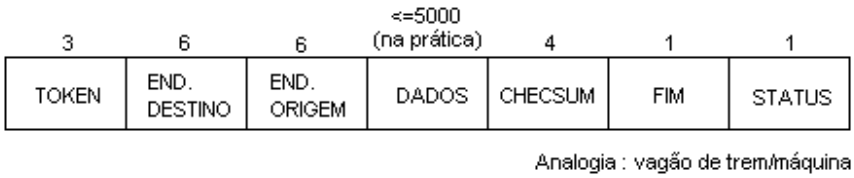


Figura 09 – Frame Token Ring

Envio

- espera token livre
- adiciona campos ao token - compõe o pacote
- torna o token ocupado
[muda status se quiser enviar]

Recepção

- compara endereço de destino com endereço local. Se diferente, passa o token (mensagem). Se igual, retira (cópia) os campos do token, tornando-o livre.

1.1.Comparação

- Para cargas leves :
  - Ethernet é mais eficiente para mensagens de comprimento longo (por causa dos repetidores do token ring) - capacidade de transmissão
  - Token ring é determinística

- Para cargas moderadas ou altas
  - Token ring é menos sensível em relação ao tempo médio de transmissão.

FDDI

É um padrão de rede que opera em 100Mbps, normalmente utiliza fibra. Emprega topologia em anel.

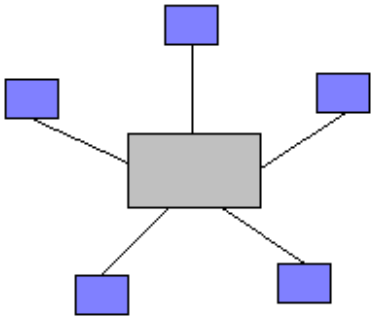


Figura 10 – Topolgia em anel utilizando concentrador

Mecanismos de comunicação

- Tipos de comunicação
  - Síncrona
  - Assíncrona

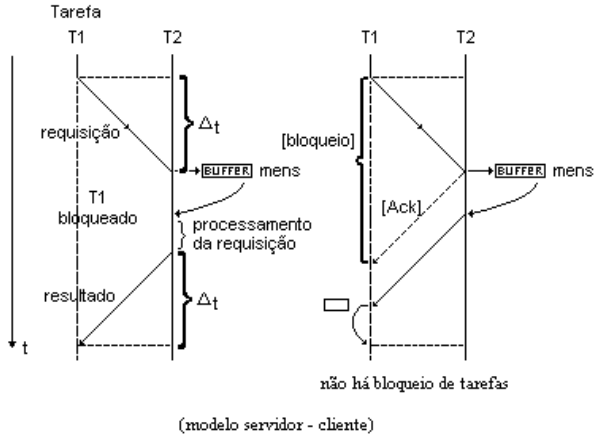


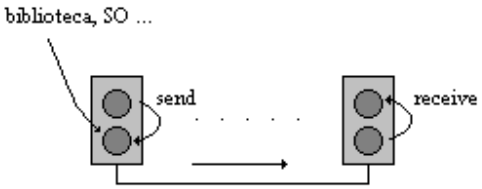
Figura 11 – Mecanismos de comunicação

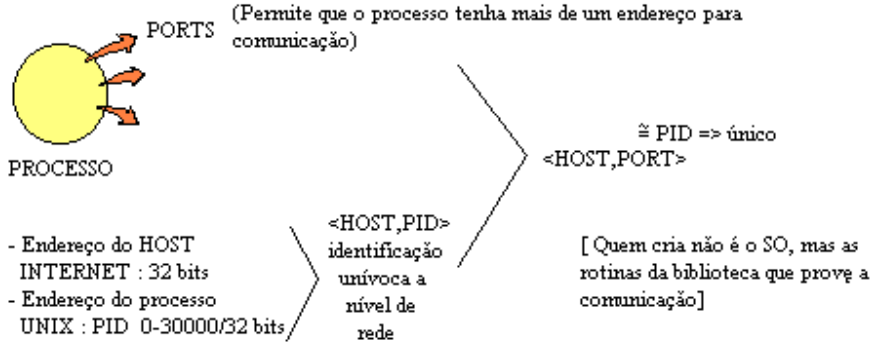
- Mecanismos
  - Troca de mensagens (síncrona/assíncrona).
  - Compartilhamento de dados (não é compartilhamento de memória!; pois trata-se de sistemas distribuídos).
  - Chamada de processamento remoto (síncrono, do tipo servidor-cliente) [rendez-vous-> ADA]

Comunicação interprocessos

Troca de mensagens

Supõe endereços (origem/destino)





PORT : um recurso utilizado para o envio e recepção de mensagens.

Mensagem : é uma cadeia de bytes trocada entre processos (normalmente via rede).

Estrutura : [Cabeça – informações gerais (quem emitiu, tamanho do corpo, tipo de mensagens...)
[Corpo – conteúdo.

Envio de mensagem : um processo pode enviar uma mensagem para : outro processo; grupo de processos (multicast); todos os processos em todos os hosts (broadcast);

Recepção de mensagem :

- seletiva (transmissor (remetente) especificado); [“ack”] [“confirmation”]
- não seletiva (recebe de qualquer remetente).

Pode ser síncrona ou assíncrona.

Codificação de mensagens :

- cópia de memória (nós homogêneos – mesma arquitetura (representação) – Tempo real)
- representação canônica (e.g. ASCII, EBCDIC; DSI:EDR; TCP/IP UNIX : XDR)

(para dados básicos)

P1
```
int i; char buff [20];
i = 3;
sprintf(buff,"%d",3);
envia buff;
```

P2
```
int j;
char buff[20];
recebe b;
sscanf(b,"%d",&j);
```

Troca de mensagens no UNIX

- Domínio (família)
  - UNIX (entre 2 processos da mesma máquina). Superado!
  - INET (Internet).
- Protocolos
  - UDP (User Data Protocol) (não confiável (SIC), entrega de datagramas, baseado no IP);
  - TCP/IP (confiável, baseado em "stream").
- Endereço de Host
  - 32 bits (unsigned long) – endereço físico.
  - nome simbólico (strings) eg. "alias", "Leblon", ..., "Iemanja",...., "Leblon.Def.FEE.Unicamp.br" – endereço lógico.
  - Nome lógico – SYS CALL – Nome físico.
- Soquete ("Socket").

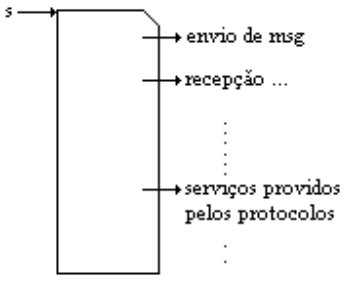
  É uma generalização do conceito de port. É um recurso utilizado também para o envio de mensagens. (usar como descritor de arquivos – pode usar as mesmas sys-calls).

Local ao processo

```
int s;          domínio      datag./stream          default_Tcp/Ip
s = socket (    AF_INET ,    SOCK_DGRAM  ,  0 )
                             SOCK_STREAM
```

Global na máquina

- Atribuição/Associação ("Binding") de nomes.
- Número de ports são inteiros
- BIND (s, ...)



O número do port pode ser atribuído pela aplicação (soquete passivo) ou pelo S.O. (soquete ativo).

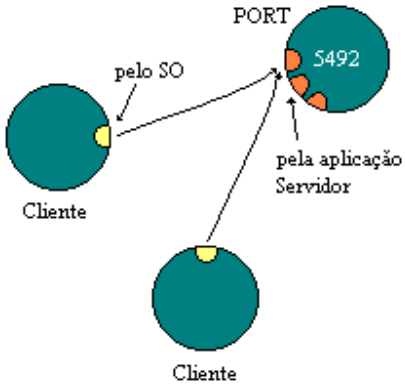


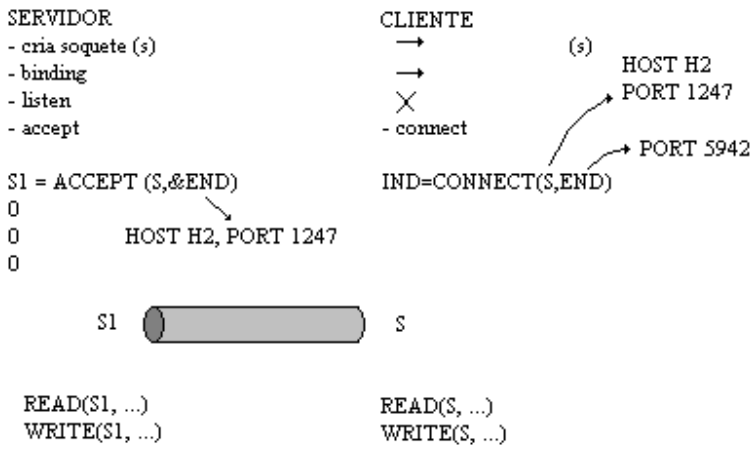
Figura 12 – Atribuição do port.

Processamentos para comunicação para datagramas :

- cria socket	- cria socket
- bind	- bind
- receive	- send
{ processa mensagem	} aguarda resultado
- send (resultado)	- receive
- close socket	- close socket
Servidor	Cliente

Conexão

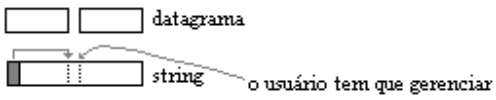
listen (systemcall) torna um soquete passivo.
- soquetes passivos aguardam conexões.
- Soquetes ativos iniciam conexões.



- Liberação de um soquete

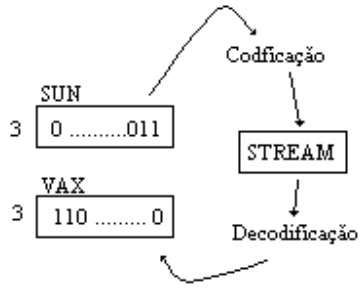
Close (s).

[UDP até 8 kbytes por mensagem : 64k garante que funciona]



Representação canônica de dados

OSI (ISO) – EDR (External Data Representation)
TCP/IP – XDR



<u>XDR</u> :

- conceito de filtro (utilizado na codificação ou decodificação).

[cra-se] stream : buffer utilizado para codificar ou decodificar.
Filtro é uma função utilizada tanto na codificação quanto na decodificação de dados.

int Filtro (Stream,&dado);
p/ inteiro no XDR:

```
int buff[128];                                          |
int i; float F;                                         |
XDR *str;                                               |
                                                        |
str = xdrmem_create(&str,buff,128,XDR_Encode);          |
i = 3;                                                  |        read (s,buff,128);
xdr_int(&str,&i);                                       |
F=3.14;                                                 |
Idr_float(&str,&F);                                     |
/* abre socket */                                       |
write(s,buff,128);                                      |
                                                      peer
```

Mapeador de Portas (Portmapper)

[Texto resumido, extraído de uma especificação realizada por G. Nakamiti e K. Souza]

Funcionalidade

Um mapeador de portas é um processo cuja função principal é prestar informações acerca de serviços disponíveis em um ou mais *hosts*, no que diz respeito ao *host* que oferece o serviço e à porta do processo correspondente. Para que cumpra o seu objetivo, o mapeador possui dois tipos de funções:

- Funções de Registro: fazem a inserção ou deleção de um serviço na tabela de controle do mapeador. Estas são requisitadas pelos servidores quando desejam se cadastrar/descadastrar;

- Funções de Pesquisa: procuram na tabela de controle do mapeador o serviço desejado por um cliente, informando o par <host,port> correspondente, caso o serviço exista, ou uma indicação de que este não está disponível.

Projeto e Implementação

Na fase de projeto de um mapeador de portas, deve-se tomar decisões quanto: à forma de comunicação entre os processos, à política de acesso seguida pelo mapeador quando um serviço é oferecido por vários servidores, e quanto ao tratamento de inconsistências causadas pela morte de um servidor.

Forma de Comunicação entre Processos

O mapeador de portas será implementado utilizando a suite Internet de protocolos. Nesta suite pode-se escolher o protocolo a ser utilizado para a camada de transporte de informações (TCP ou UDP), ou seja, se a comunicação com o mapeador será feita por meio de uma conexão ou através do envio de datagramas.

Na forma conectada, a comunicaçãoé confiável, livre de erros e sem contornos, e o protocolo que a implementa se encarrega da retransmissão de mensagens recebidas com erros, porém, esta forma se justifica melhor pelo seu uso frequente.

No envio de datagramas, cada mensagem é endereçada individualmente e a sua entrega não é garantida. é utilizado principalmente em requisições que requerem resposta do destinatário em um temporazoável, antes que seja tentada uma retransmissão.

Para a implementação do presente trabalho foi escolhida a forma de datagramas pela seguinte razão:

A comunicação entre os servidores/clientes e o mapeador consiste apenas de uma requisição e de uma confirmação/resposta. Desta forma, caso uma conexão fosse estabelecida, esta seria bastante sub-utilizada.

Política de Acesso

Quando um serviço é oferecido por vários servidores, o mapeador deve escolher um deles para enviar como resposta ao cliente. Para garantir uma melhor distribuição de carga entre os servidores decidiu-se pela escolha através de uma distribuição uniforme entre eles.

Tratamento de Inconsistências

Quando um dos servidores morre devido a um *crash* no sistema, ou a um problema interno ao próprio processo, deve-se tomar alguma providência no sentido de não informar ao cliente que determinado serviço está disponível, quando na verdade ele não está. Neste sentido, várias soluções são possíveis, com maior ou menor confiabilidade:

- Cadastro Periódico dos Servidores: onde cada servidor faz o seu recadastramento periódico, indicando que o serviço ainda está disponível. Esta possui o incoveniente de o servidor morrer logo após um recadastramento e o mapeador continuar a informar que o serviço oferecido por ele ainda está disponível, porém, este é um indeterminismo que existe em maior ou menor grau em qualquer das soluções encontradas para o problema. Esta solução, além de causar sobrecarga na rede, também implica em um aumento da complexidade de cada servidor cadastrado;

- Verificação de Servidor Ativo: procura reduzir a complexidade mencionada no item anterior, levando-a para o mapeador, que verifica periodicamente, ou quando um serviço é solicitado, a existência de um dado servidor. Neste caso, deve-se fazer uma estatística de utilização de cada servidor para escolher a melhor política: a verificação periódica é mais indicada para os servidores de maior utilização, enquanto que a feita no momento em que um deles é selecionado é mais indicada no caso oposto. Esta última opção, apesar de reduzir a sobrecarga na rede, implica em um aumento no tempo de resposta do mapeador;

- Informação do Cliente: ao receber uma informação inválida (servidor que não mais existe), o cliente envia uma mensagem ao mapeador indicando que aquele servidor deve ser descadastrado. Esta solução não apresenta atrasos nem sobrecarga na rede, entretanto, causa um aumento na complexidade dos clientes que interagem com o mapeador;

- Ativação do Servidor pelo *portmapper*: o proprio mapeador, ao receber um pedido do cliente, ativa o servidor e retorna o seu par <host, port>correspondente. Neste caso, o mapeador se utilizaria de RPCs (*Remote Procedure Calls*) para fazer a ativação do servidor.

Aumento de Desempenho do Mapeador

Nas seções anteriores ressaltamos o atraso na consulta/envio da resposta que existe em qualquer forma de implementação do mapeador, além de sobrecarga na rede se o mapeador estiver numa única máquina. Para amenizar estes problemas propõem-se as seguintes medidas:

- Armazenamento em cache local, onde se faria a consulta ao mapeador apenas quando a informação não estivesse disponível localmante. A vantagem principal deste está na rapidez da consulta quando o serviço é solicitado frequentemente, e a sua desvantagem está na manutenção desta estrutura de cache, principalmente quanto a problemas de consistência;

- Replicação dos mapeadores em todas as máquinas, cujas vantagens seriam a rapidez na resposta e diminuição na carga imposta à rede. Suas principais desvantagens são: o aumento na carga imposta aos vários processadores pela adição de mais um processo por máquina, e o problema de se manter uma consistência global;

- A transformação do mapeador em um servidor de RPC, no qual, ao ser recebida uma solicitação de serviço, uma mensagem é enviada pelo mapeador ao próprio provedor do serviço, que dispara um processo que irá atendê-lo, enviando de volta uma resposta.

Chamadas de procedimentos remotos (RPC)

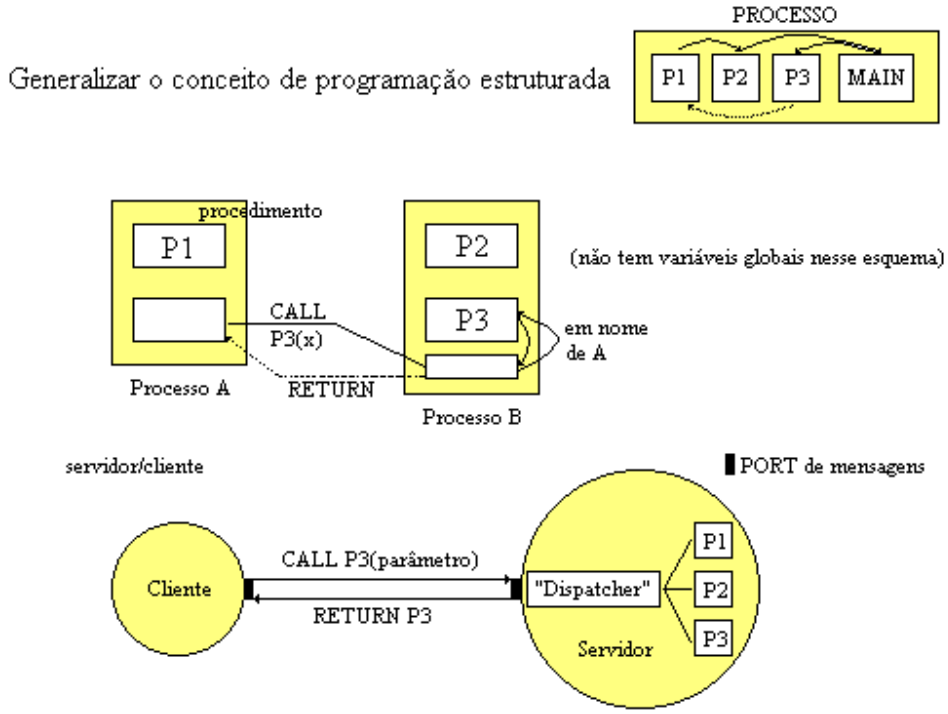


Figura 13 – Chamada de procedimento remoto (RPC)

Transparência das chamadas

- transparência total
  - RPC é embutido na linguagem de programação [não existe]; ou
  - Usar pré-processador (STUB) das chamadas, que identifica se chamada é local/remota, extrai os tipos de parâmetros, codifica, localiza, abre ports, comunica, recebe (depende do compilador, versão – é difícil manter).
- transparência limitada.

Estrutura do servidor

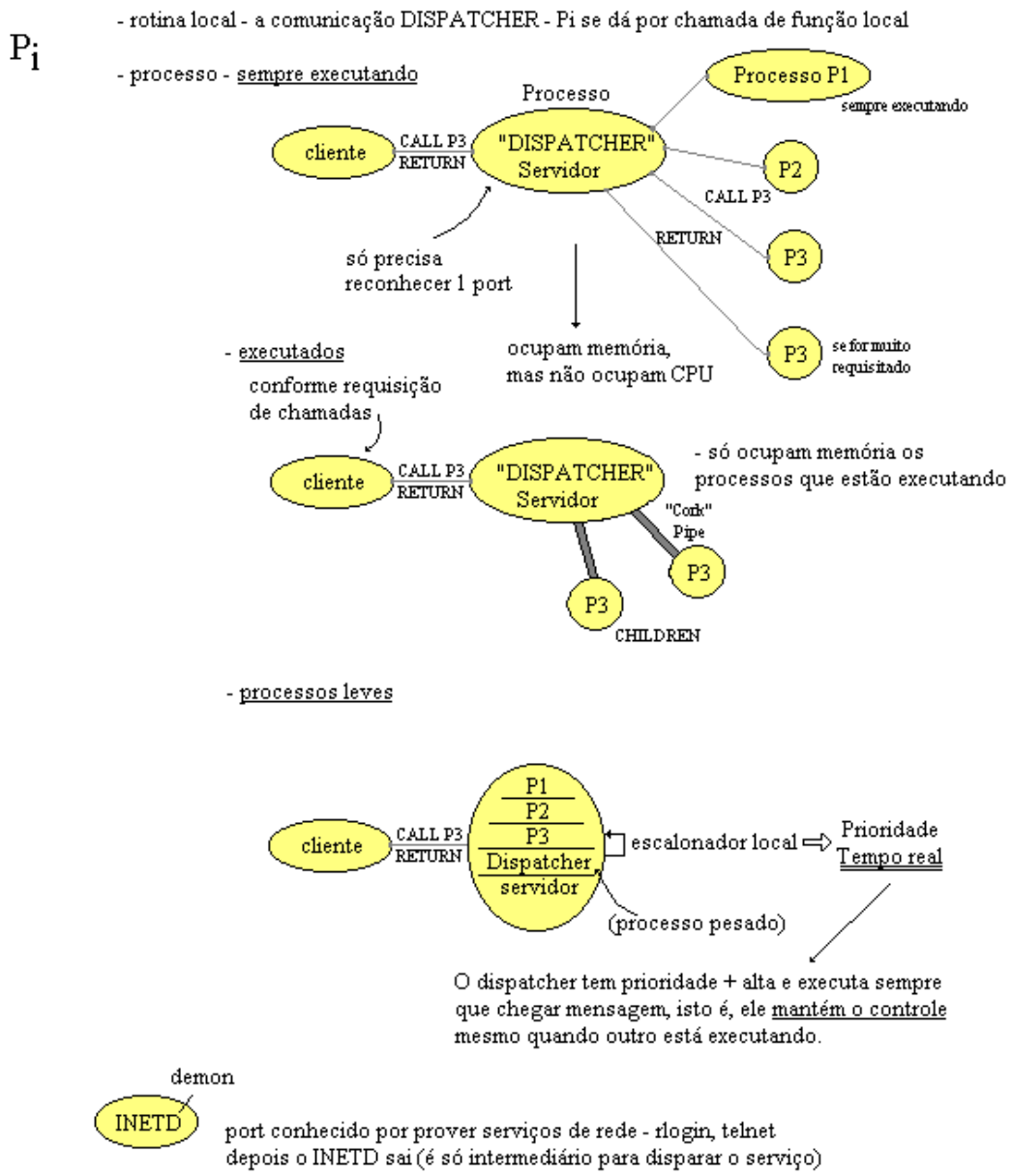


Figura 14 – Estrutura do servidor

Passagem de parâmetros (para transparência limitada)

- passagem sempre por valor;
- os parâmetros são passados numa forma canônica (e.g. XD2);
- restrições na quantidade de parâmetros passados/retornados (normalmente 1/1).

Registro de procedures **(serviços)**

1. localização do servidor (HOST,PORT);
2. localização da procedure (normalmente Nome simbólico).

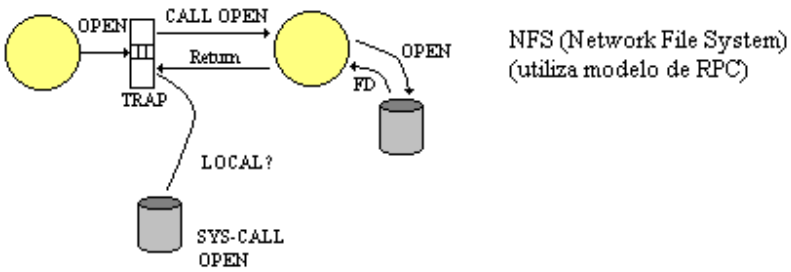


Figura 15 – Registro de procedures

SUN RPC

- total indeterminismo (não garante tempo para atendimento de chamdas)
- velocidade baixa (software de alto nível implementado com RPC – competição com serviços normais de rede).
- Identificação de procedures
  - identificação do servidor (ID);
  - versão do servidor;
  - ID da procedure.

Exemplo : um serviço de números globais.

glob.h :

```
#define NGSERVER 14972
#define NGVER 0
#define NGPROC 1

int ng = 0;
int nglob(void); /*função*/

{
ng++;
return(ng);
}
```

servidor :

```
main();
{
registerrpc(NGSERVER,NGVER,NGPROC,NGLOB,xdr_void,xdr_int);

SVC_RUN(); /*DISPATCHER - "CASE" LOOP infinito */
}
```

protótipo da procedure; identificação do serviço; endereço da função que provê o serviço; filtros de E/S (ponteiros p/ funções)

cliente :

```
main();
{
int gl;
callrpc("leblon",NGSERVER,NGVER,NGPROC,rdr_void,0,rdr_int,&gl);
/* gl é um número global na red */
printf("%d",gl);
}
```

!! tem que dar o endereço

MAILBOXES

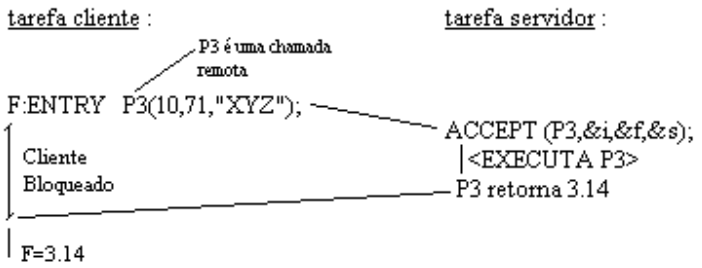
Idêntico a troca de mensagens, exceto no conceito de port.
MAILBOX (MB) é uma extensão do conceito de port. É mais abstrato, de mais alto nível.
É identificado por u nome simbólico "XYZ".
Eventualmente "tipado", i.é., um MB do tipo t recebe mensagens apenas do tipo t.
Eventualmente possui um grupo de "proprietários". Dois esquemas : 1 lê ou todos lêem.
Possui limites na criação (e.g. tamanhos diferentes de mensagens pendentes).

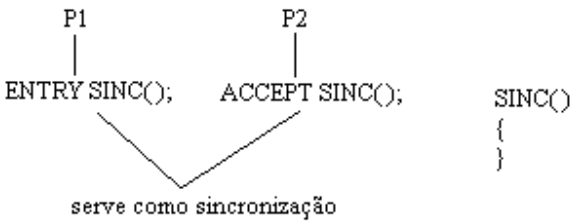
Primitivas básicas :

[grupos de]
cria_mb("XYZ",tipo,limites,usuários);
ch=acessa_mb("XYZ");
port_mb(ch,msg);
get_mb(ch,&msg);
destrói_mb(ch);

RENDEZVOUS (ADA)

Para comunicação entre tarefas.
Normalmente é embutido na linguagem de programação.
Similar à RPC, mas com aceite da execução da procedure explicitado pelo servidor.





Memória distribuída com replicação de dados

MRSW

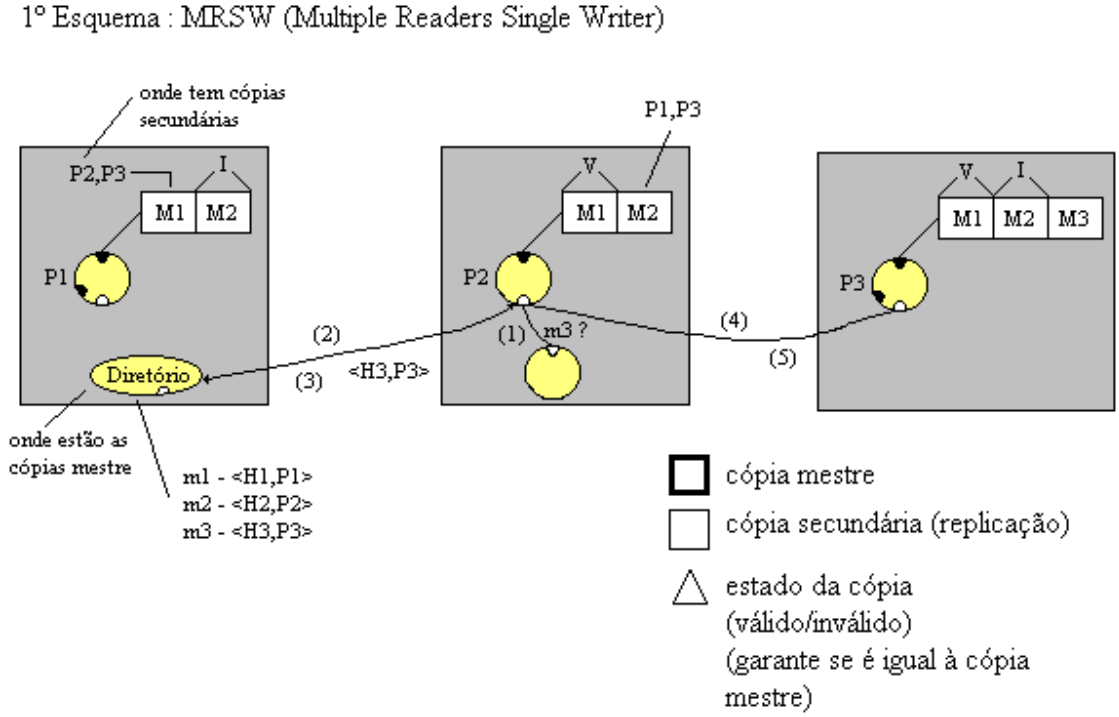


Figura 16 – Memória distribuída com replicação de dados

Procedimento para leitura de m_i :

a) existe cópia local de m_i
   a1) Válida – lê
   a2) Invalida – obtém cópia válida (diretório, instalando-a localmente, atualizando a lista de onde está as cópias secundárias); lê.
b) não existe cópia local de m_i
   obtém cópia válida (diretório), instalando-a localmente; lê.

Procedimento para gravação (atualização)

Dois esquemas são possíveis

a) gravações sempre se realizam no host onde se localiza a cópia mestre.
   a1) invalida todas as cópias secundárias (recebendo ack).
   a2) grava.

b) gravações são sempre locais (a cópia mestre muda de host).
   b1) invalida todas as cópias secundárias [ack].

b2) migra cópia mestre.
b3) grava.

Vantagens :

- tolerância parcial a falhas;
- eficiente quando a taxa de leitura sobre gravações (L/G) é alta;
- paralelismo total para leitura; por partições (parcial) para gravações.

Desvantagens :

- complexidade na manutenção da consistência.

MRMW

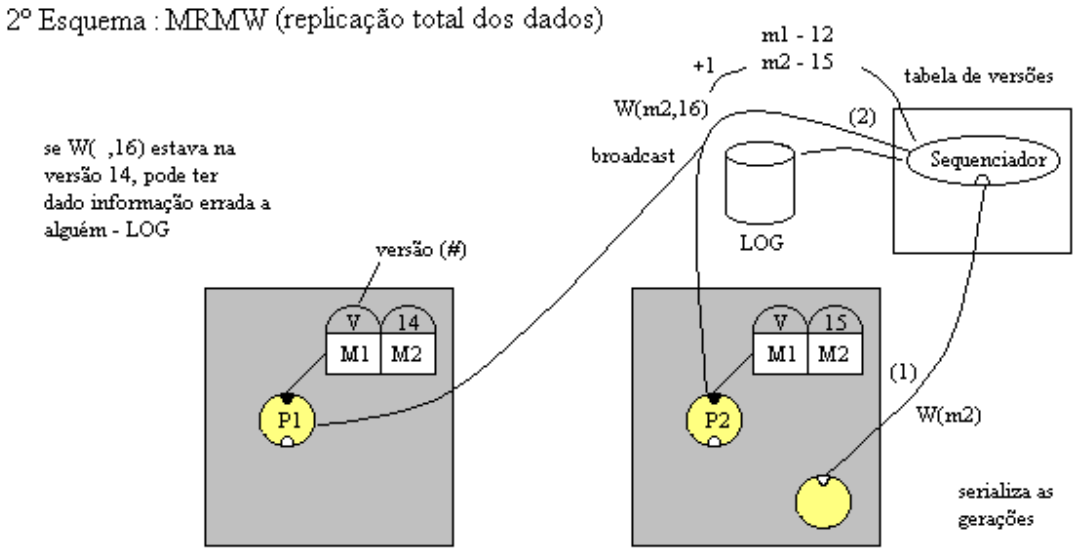


Figura 17 Memória distribuída com replicação de dados

Problema :

O broadcast entrega aos hosts em instantes diferentes, pois em SDs não existe clock central.

Solução :

Invalidar as cópias antes do broadcast – eficiência !! baixa.

Vantagens :

- equivalente a anterior, muito mais com replicação total sempre consistente – tolerante a falhas.

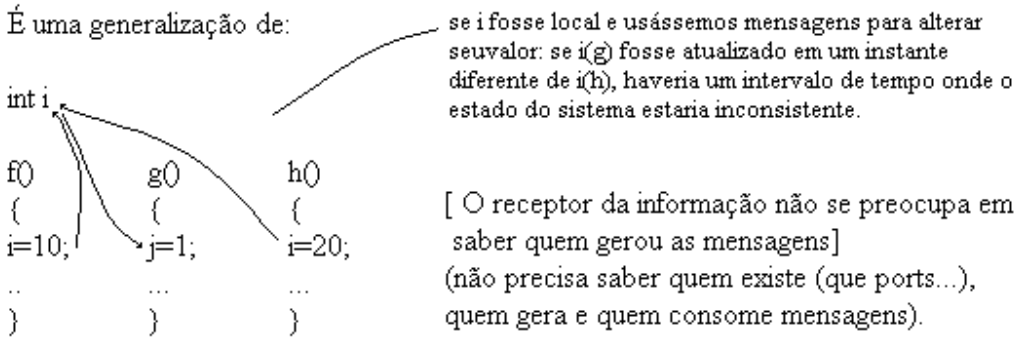
Desvantagens :

- equivalente a anterior, mais gargalo no sequenciador (melhora com 1 sequenciador para cada partição da memória).
- Maior necessidade de remontar versões ante a ocorrência de gaps de versões – uso do log.

Estruturação de memória

- buffer não interpretado no sentido de empacotamento;
- não ocorre empacotamento/desempacotamento de dados – *struct* conhecido por todos – só cópia de bits. Serve para tempo real. Só para redes homogêneas. (Normalmente cada mensagem m_k corresponde a um buffer B_k, que corresponde a um *struct* S_k).
- dados estruturados – depende da aplicação. Tem que ser interpretado.
  - n-uplas (e.g. "string",19,17-47)

Comunicação por compartilhamento de dados



Vantagens :

- assincronismo entre produtor/ consumidor de informação;
- modularidade/ extensibilidade;
- definição mais precisa de "estado";
- familiar ao programador (≠ IPC – sockets, __)
- não favorece nenhuma arquitetura particular de sistemas distribuídos.

Desvantagens :

- velocidade de comunicação;
- controle de acesso (e.g. problemas de "corrida"- ≈ "transação ").

Compartilhamento de dados em sistemas distribuídos

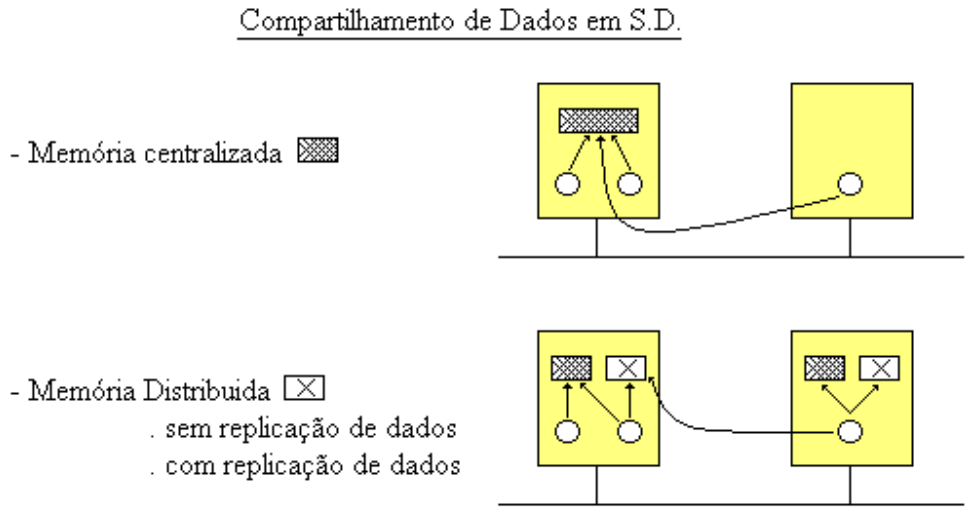


Figura 18 – Compartilhamento de dados em SD

Operações básicas

- Fetch (real) atômicas (*sic*)
- Store (write)

Implementação de "memória compartilhada" em SD

- através de processos

Figura 19 – Implementação de "memória compartilhada" através de processos

- memória centralizada

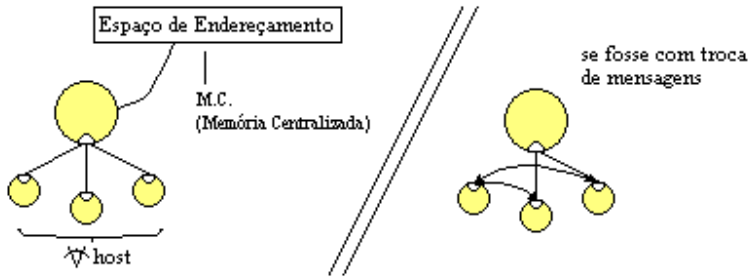


Figura 20 - Implementação de "memória compartilhada" através memória centralizada

Vantagens :

- a litura retorna sempre o valor da última escrita;
- consistência sempre garantida (se as operações forem atômicas);
- implemetação simples.

Desvantagens :

- criação de gargalo, com prejuízo da eficiência;
- confiabilidade.

Implementações alternativas para memória centralizada

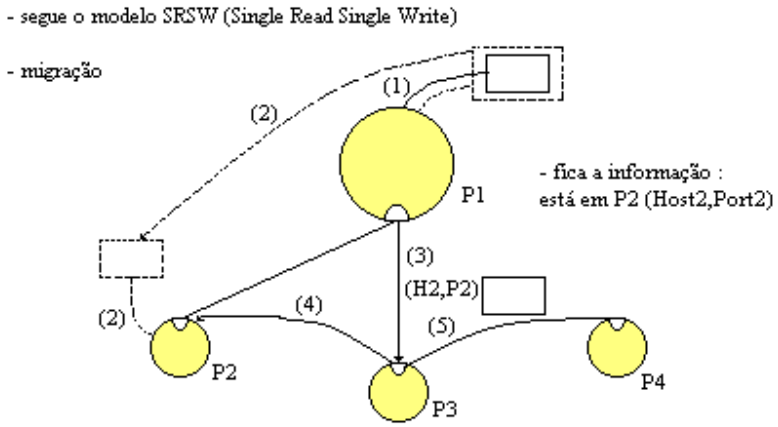


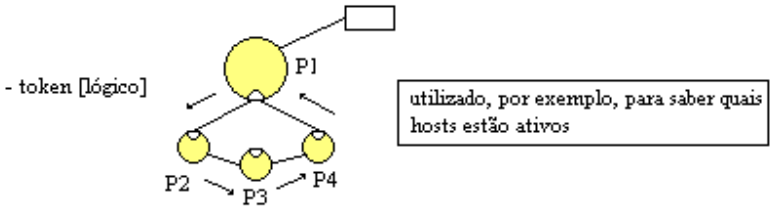
Figura 21 – Implementação alternativa para memória centralizada.

Vantagens e desvantagens similar a anterior.

Esquemas de localização de memória compartilhada

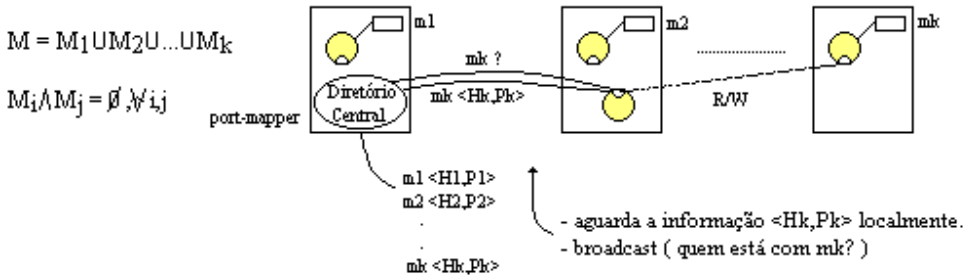
- cadeia de ponteiros;
- broadcast (com quem está?);

- diretório central (e.g. portmapper).
- token [lógico].



A memória circula (é o token - ou melhor, parte dele) - mensagens pequenas
- algoritmo já existente que precisa circular na rede.
- desv.: latência do token

Memória distribuída sem replicação



Obs.: o diretório central pode ser substituído por um esquema de broadcast.

Vantagens :

- consistência garantida;
- implementação simples;
- paralelismo de leitura/gravação (para regiões disjuntas – $m_i \cap m_j = \varnothing$).

Desvantagens :

- confiabilidade (morte súbita de processador/ processo) – (*bottleneck* pode ser resolvido pelo particionamento e alocação de memória).

Obs.: Esse esquema também permite imigração (por ex. para balanceamento dinâmico de carga, sobrevivência a paradas programadas).

- frames {classe [ATR,VALOR] ... [ATR, VALOR] }

Memória compartilhada (M.C.)

Controle de acesso – manter consistência :

Lock :

É uma chave de acesso para uma dada região da memória compartilhada (granularidade). Locks possuem certa abrangência (granularidade).

Tipos : - leitura.
- gravação.

A serialização implica numa política de obtenção de locks.

Locking em 2 fases

Teorema : Se um conjunto de operações (R/W) obedece as regras :

a) antes de ler ou gravar uma região de memória, obtenha o respectivo lock para aquela região;
b) depois de liberar um lock nenhum outro será requisitado para essa seqüência de operações;

Então qualquer conjunto de operações é realizável (preserva a consistência mas não a ordem temporal em sistemas distribuídos – deadlocks)

1) Locking por marcas de tempo (m.t.) (menos flexível para locks em regiões inter-dependentes).

Regiões de memória compartilhada têm a elas associadas marcas de tempo;

- "hora" da última leitura;
- "hora" da última gravação.

Antes de iniciar uma seqüência de operações, o processo obtém um "time-stamp" (T.S.).

Protocolo :

- uma leitura numa dada região de memória só se processa se a M.T.W. (marca de tempo de gravação) for menor que o time-stamp.
- Uma gravação numa dada região da M.C. só se processa se:
  $MT_W < TS$
  $MT_{READ} < TS$

(Se qualquer dessas operações falhar, o processo pede outra M.T. e tanta repetir a operação)

Locking por validação otimista

FASE A : As operações são executadas sem preocupação com serialização.
FASE B : Verifica se ocorreu violação com a serialização. Caso afirmativo, um conjunto de operações é desfeito.

Controle interprocessos

Mecanismos :

- instanciação (criação)
- e controle da execução (e.g. Suspensão, término)
- sincronização
- tratamento de exceções.

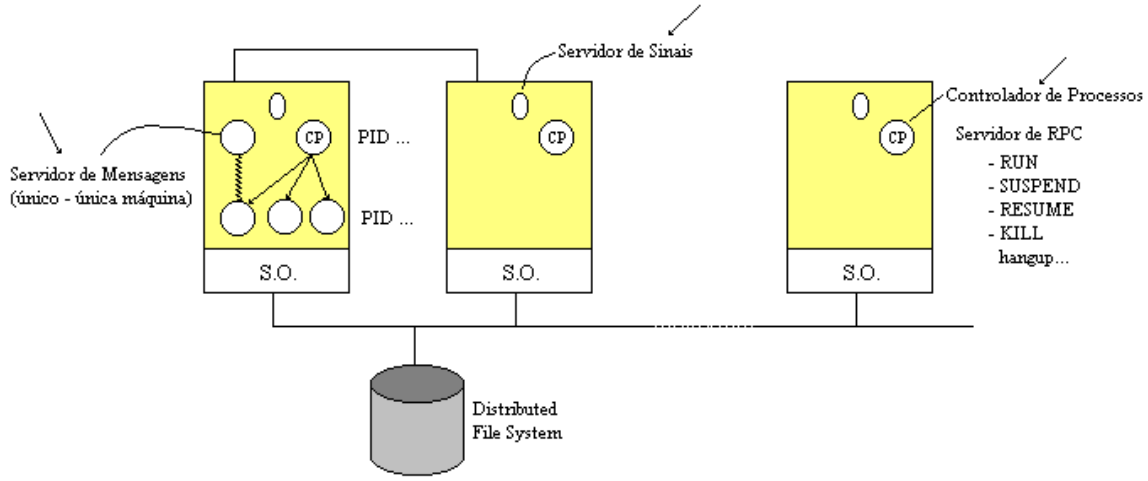


Figura 22 – Controle inter-processos

Deadlock

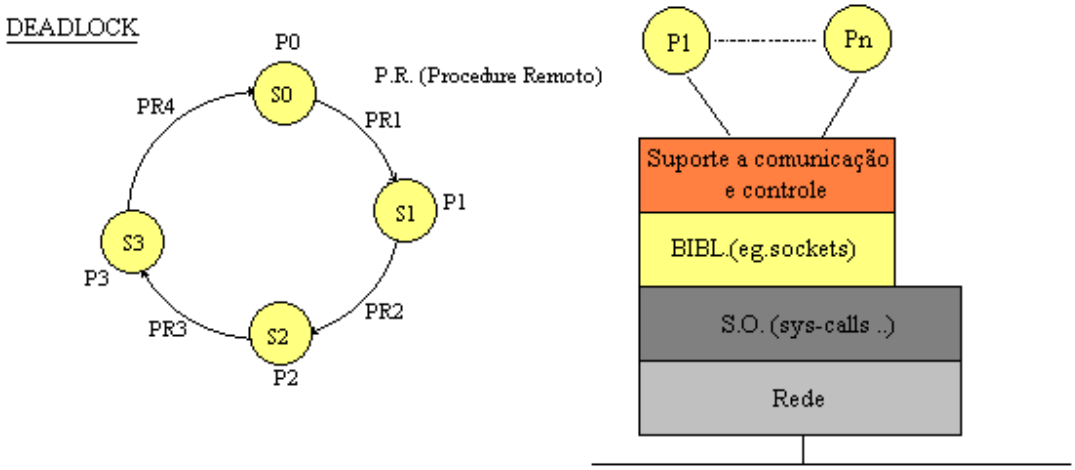


Figura 23 - Deadlock

Caracterização do deadlock

Sejam P_j processos aguardando eventos E_i; $P_j \in P$; $E_i \in E$; P_j (E_i) (P_j aguarda evento E_i), de sorte que $E_k \in E$ somente ocorre por ação de $P_e \in P$.

Condições necessárias para ocorrência de deadlocks

1) exclusão mútua;

- lógico (dados compartilhados);
- físicos (e.g. impressora) e de comunicação (e.g. chegada de mensagens).

2) "hold-and-wait"; (segura os recursos necessários que já obteve e espera a liberação de outros).
3) Inexistência de preempção;
4) Existe (ou pode existir) espera circular.

Obs.: Nem todas são necessárias ao mesmo tempo.

Grafo de alocação de recursos

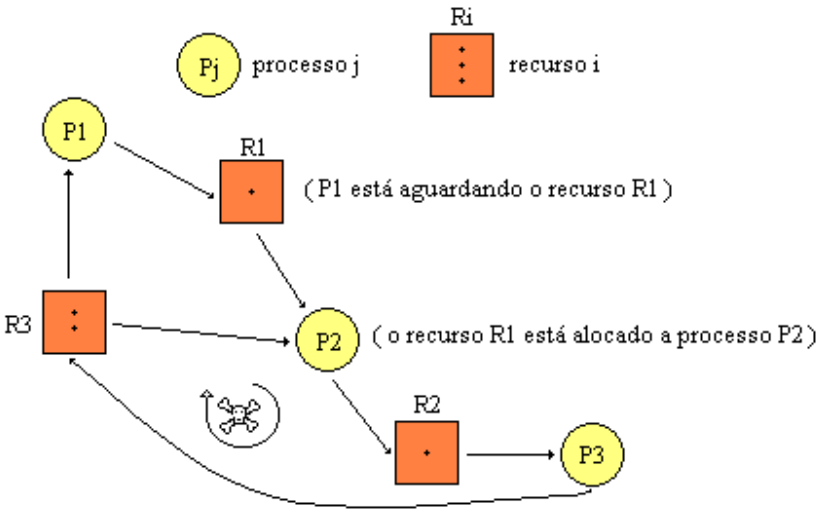


Figura 24 – Grafo de alocação de recursos

Tratamento de deadlocks

a) prevenção (cuidar para não ocorrer);
b) fuga (evitar a situação);
c) detecção (já ocorreu);
d) quebra (quebrar o ciclo).

Prevenção

1) adição de recursos (condição 1);
2) protocolo de requisição (condição 2);

- permitir a utilização de um único recurso por vez;
- obtenção de todos os recursos antes de sua utilização.
  - Ineficiбencia (fica com o recurso mais tempo que o necessário);
  - "starvation" (funciona como com prioridades).

3) admitir a preempção (condição 3);

Se um processo requisita um recurso não disponível, todos os recursos de posse do processo lhe são tomados.

4) evitar a espera circular (condição 4).

Enumera-se os recursos R_1, R_2, ... R_n e impõe-se que os processos requisitem recursos apenas em ordem crescente.

- ineficiência;
- "starvation".

Fuga

Algoritmo de banqueiro (empresta quando disponível, seleciona quando escasso, mantém sempre uma situação "saudável")

Dados :

- DISP[j]=k; ∃ k instâncias do recurso j;
- MAX [i,j]=k; o processo P_i pode requisitar no máximo k instâncias do recurso j;
- ALOC [i,j]=k; o processo P_i tem a ele alocado k instâncias do recurso j;
- NEC [i,j] = MAX [i,j] – ALOC [i,j] = k; P_i pode vir a necessitar (solicitar) de k instâncias do recurso j;
- REQ [i,j] = k; o processo P_i está requisitando k instâncias do recurso j;
- REQ [i,j] ≤ NEC [i,j].

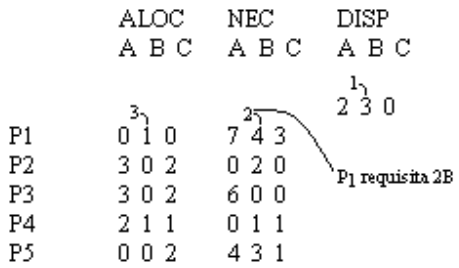
Algoritmo

1) Se REQ [i,j] ≤ NE [i,j] prossiga SENÃO reporte erro;
2) Se REQ [i,•] ≤ DISP [•] prossiga SENÃO processo deve esperar;
3) Compute DISP[•] = DISP[•] – REQ [i, •];
   ALOC [i, •] = ALOC [i, •] + REQ [i, •];
   NEC [i, •] = NEC [i, •] – REQ [i, •];
4) Para este novo estado, verifica a sua "segurança"
   Seja W [k] = DISP [k], k=1, __ , m;
   F [k] = FALSO, k=1, __, m;

   a) escolha k tal que :
      F [k] = FALSO;
      NEC [i,k] ≤ DISP [k];
      Se ∄k vá para c
   b) W [k] = W [k] + ALOC [i,k];
      F [k] = VERD;
   c) Se F [k] = VERD ∀k então o estado é seguro, caso contrário é inseguro.

5) Se o estado é seguro, conceda os recursos solicitados, caso contrário desfaça o passo 3 e não conceda.

Exemplo



Detecção

1) W [k] = DISP [k]
   Se ALOC [i,k] ≠ 0
      F [i] = FALSO. SENÃO F [i] = VERD

2) escolha i tal que
      F [i] = FALSO
      REQ [i,k] ≤ W [k]
      Se ∄ i vá para 4
3) W [k] = W [k] + ALOC [i,k]
   F [i] = VERD
   Vá para 2
4) Se F[i] = FALSO para um dado i, então o sistema está em deadlock e o processo $P_i \in$ P.

Processamento de transações

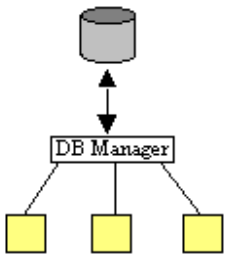


Figura 25 – Processamento de transações - Centralizado

Definição : Uma transação é um conjunto de operações delimitadas por um "BeginTransaction" e um "EndTransaction" , que satisfazem as seguintes propriedades :

1. Serialização (inter-transações) – para objetos comuns.
2. Atomicidade (em relação a falhas) – (já era obtido com "locks" , o sistema não fica inconsistente).
3. Persistência (em relação à queda de processadores).

Operações : locais ou remotas

- leitura (R);
- Gravação (W).

Processamento de transações (do ponto de vista do usuário).

```
1) t-id = Begin_Transaction ();                 /* abertura */
2) if (lock (t_id, object, modo R/W))           /* aquisição – obtém os locks */
       { operações                              /* operações */
         ...
       }
   else Abort_Transaction (t_id);
3) End_Transaction (t_id).                      /* finalização */
```

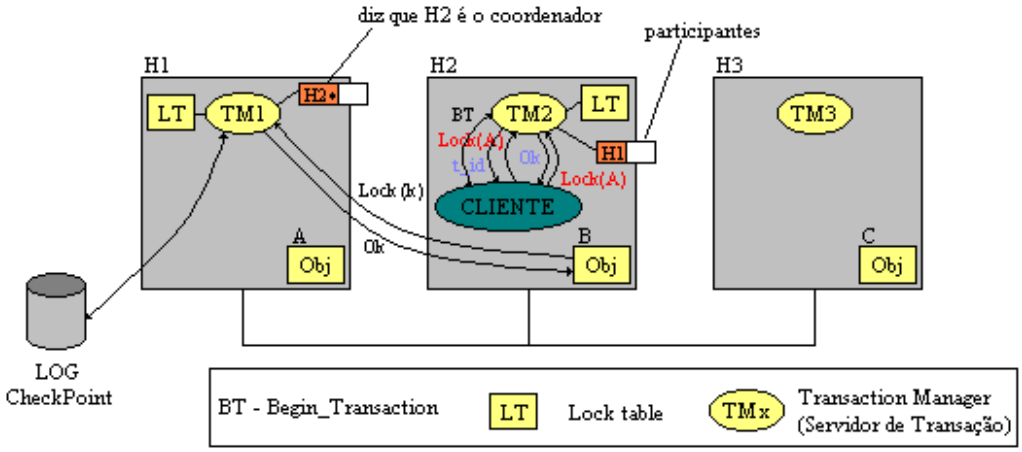


Figura 26 – Processamento de transação

Quebra (recuperação)

1) terminação;
2) preempção

- seleção da vítima;
- tomada dos recursos;
- roll_back (volta para algum estado anterior consistente).

Obs.: custoso (geralmente evita-se usando locks com time-outs) ; garante que não há deadlocks, mas não garante que haja progresso.

Deadlock em Sistema Distribuído

Estratégias de prevenção, fuga, detecção e recuperação.

estratégia centralizada

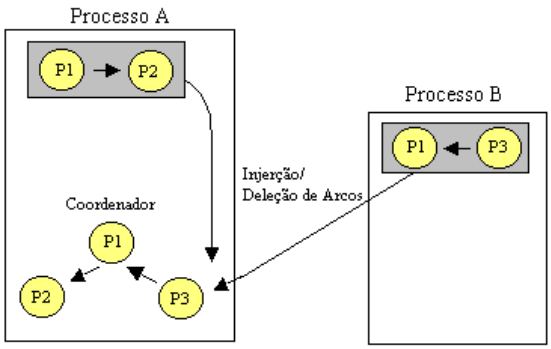


Figura 27 – Deadlock – Estratégia centralizada

Desvantagens :

- perda de mensagens (estado inconsistente);
- chegada de mensagens em ordem temporal trocada;
- toda a informação converge para um único processo.

estratégia hierárquica

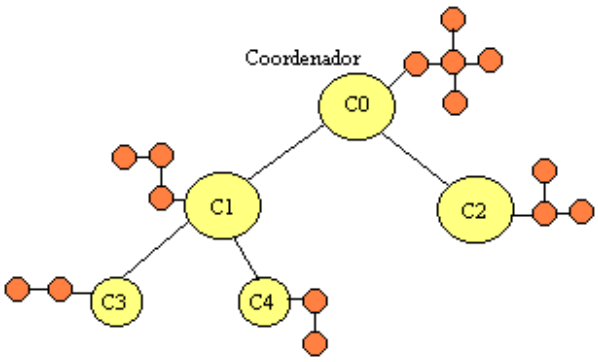


Figura 28 – DeadLock – Estratégia hierárquica

estratégia distribuída

- aguardando mensagens (recursos de comunicação); time-out – envia mensagem multicast, se voltar ele próprio está em deadlock e (se mata, mata filhos, ...)
- caso geral :

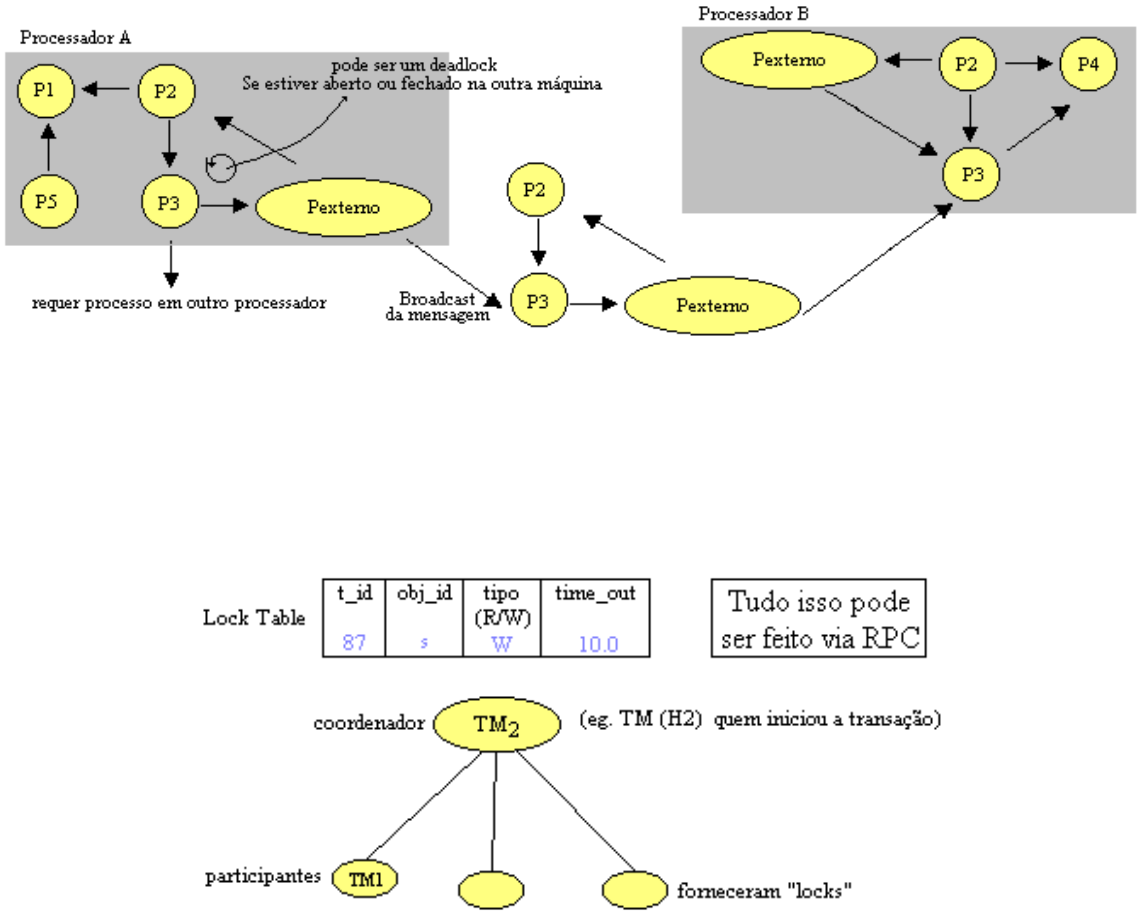


Figura 29 – DeadLock – Estratégia distribuída

Protocolo de obtenção de locks

- Locking em 2 fases – mais simples, mais usado (nesse todos os locks só são liberados no final da transação, pois senão poderia ser lido um valor inconsistente alterado por uma transação que ainda iria ser abortada (para garantir atomicidade).
  Solução: não prover primitivas para liberar locks – só através do End_Transction ou Abort_Transaction.
- Timestamp – necessita de marcas de tempo globais.

Locking em 2 fases

Atomicidade : restaurar estado anterior:
- cópia de objetos antes das modificações.
- "undo" log (guarda operação inversa – ocupa menos espaço em disco, mas é mais difícil de implementar).

Persistência

- supõe a existência de uma média estável (geralmente disco).

- Tem que, após falhas, deixar no estado após a última transação (de escrita) terminada com sucesso.
- "checkpoint" global ao final das transações.
- "checkpoint" parcial + log.

Protocolo de finalização

- "Commit" em 2 fases

1º fase

pré-commit (o coordenador pergunta aos participantes se estão prontos para terminar)

se alguém responder não – abort para todos e recuperando estado dos objetos, senão:

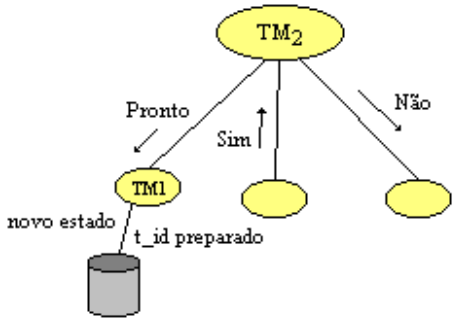


Figura 30 – Protocolo de finalização – 1º fase

2º fase

commit (o coordenador manda uma mensagem para terminar a transação – os participantes mudam o estado provisório para o estado novo e liberam os locks)

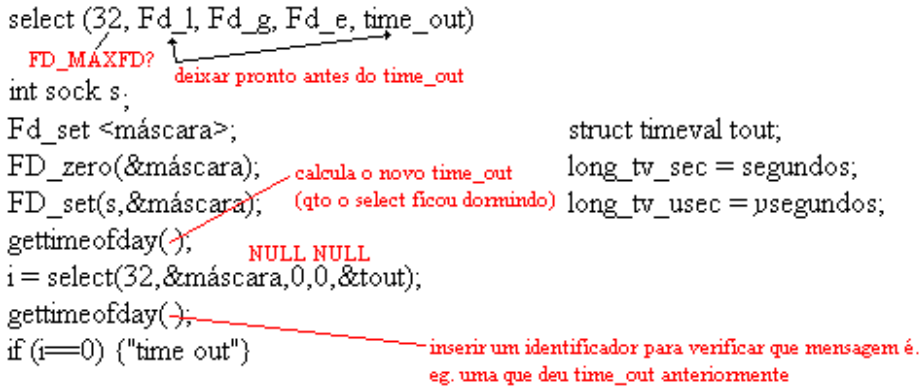
Tempo real

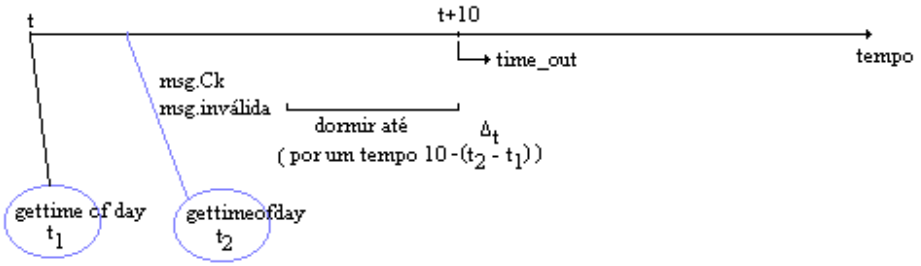
- time-outs (para locks e todas as transações);
- priorização das transações;
- relaxamento da persistência (usando eg. Memória RAM, EPROM).

RPC - time-outs
- priorização (processos leves).

SELECT

Verifica o estado dos descritores para leitura, gravação e "conduções especiais" .

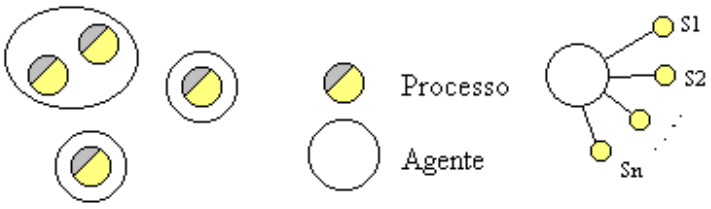

select (32, Fd_l, Fd_g, Fd_e, time_out)
FD_MAXFD?
deixar pronto antes do time_out
int sock s;
Fd_set <máscara>;
FD_zero(&máscara);
FD_set(s,&máscara);
gettimeofday( );
i = select(32,&máscara,0,0,&tout);
gettimeofday( );
if (i==0) {"time out"}
struct timeval tout;
long_tv_sec = segundos;
long_tv_usec = µsegundos;
calcula o novo time_out
(qto o select ficou dormindo)
NULL NULL
inserir um identificador para verificar que mensagem é.
eg. uma que deu time_out anteriormente


t
t+10
time_out
tempo
msg.Ck
msg.inválida
dormir até
Δt
(por um tempo 10 -(t2 - t1))
gettime of day
t1
gettimeofday
t2

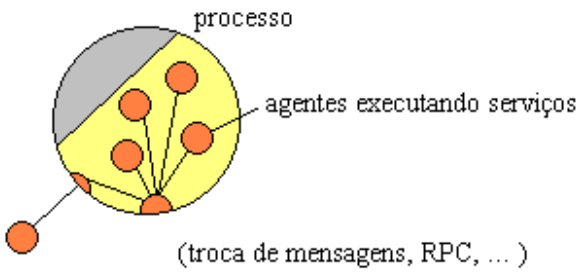
Arquiteturas distribuídas

Arquiteturas

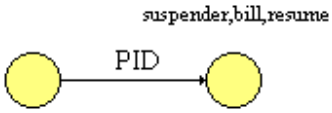
- conjunto de agentes (módulos) : os componentes da arquitetura



- vias de comunicação entre os agentes



- vias de controle entre os agentes
  - controle de execução de agentes.
  - escalonamento de serviços (tarefas – trabalho, serviço – implementação através de processo ou função)



relacionado à atividade de resolução do problema.

Arquiteturas (níveis)

- estrutural
  - definição dos agentes;
  - vias de comunicação;
  - vias de controle.
- plano de controle/comunicação (nível de detalhamento – como as ações de controle são tomadas, gerenciadas; e.g. como sair de um deadlock)

Nível estrutural

hierarquias – vias de controle e comunicação formam uma árvore e um agente no nível i : * controla N agentes no nível i+1; * comunica-se com n agentes no nível i+1 e P agentes no nível i-1.

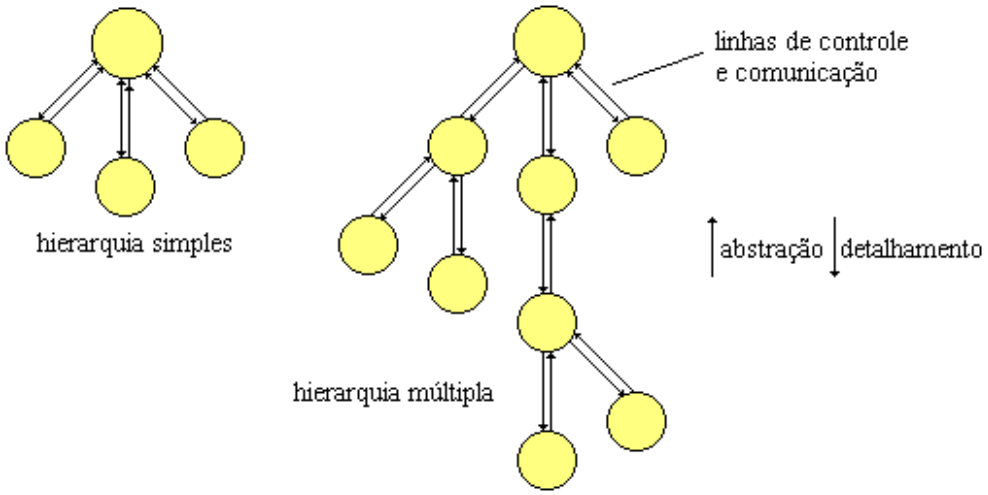


Figura 31 - Hierarquias

heterarquias (não hierarquias) – vias de controle e comunicação são indefinidas ou dinâmicas.

Exemplo de hierarquia

(não existe hierarquia pura em caso de emergência, pode-se controlar/comunicar com níveis diferentes).

O plano de controle/comunicação normalmente é descrito em determinado nível.

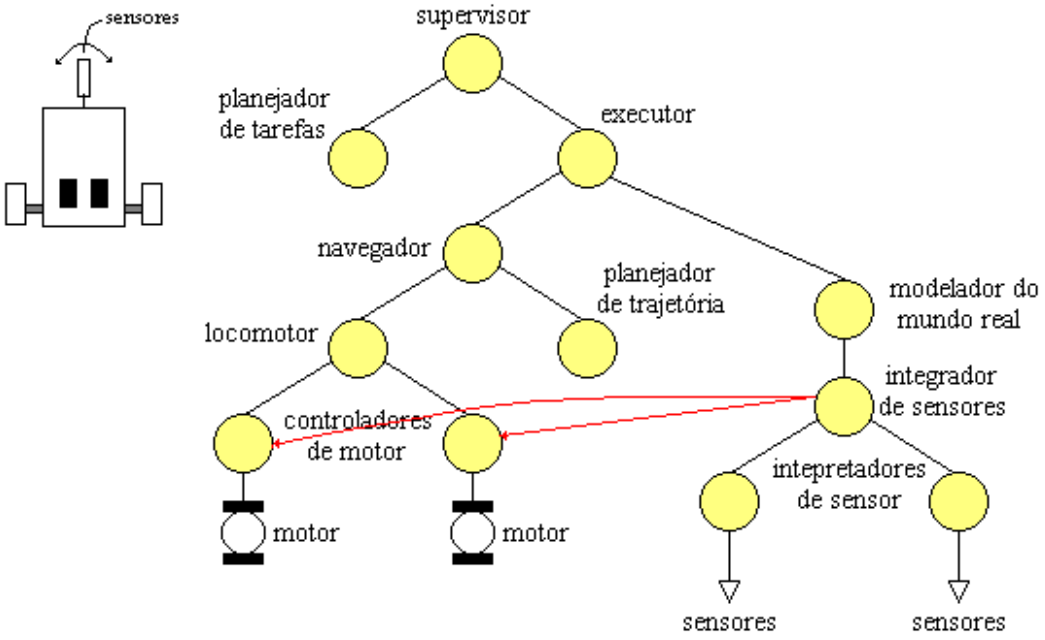


Figura 32 – Heterarquia

Exemplo de heterarquia

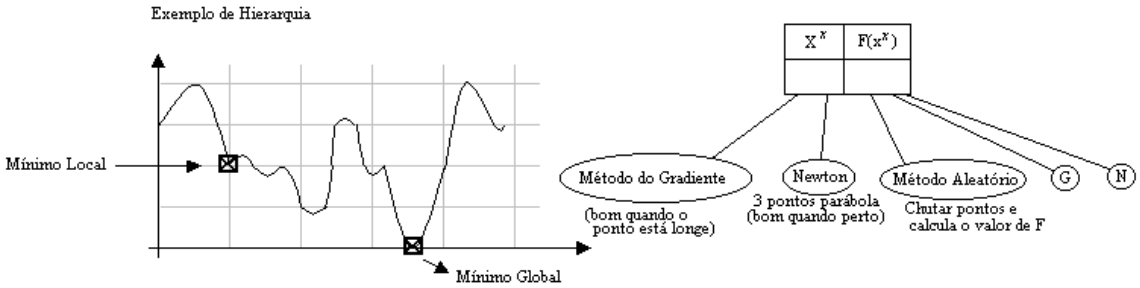


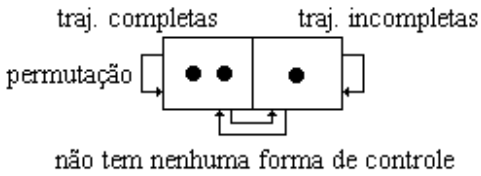
Figura 33 – Exemplo de heterarquia

Plano de controle/comunicação

BLACKBOARD (pool de dados) monitor; fila de escalonamento é uma hierarquia(*sic*) data oriented; memória passada com 2 BBs.

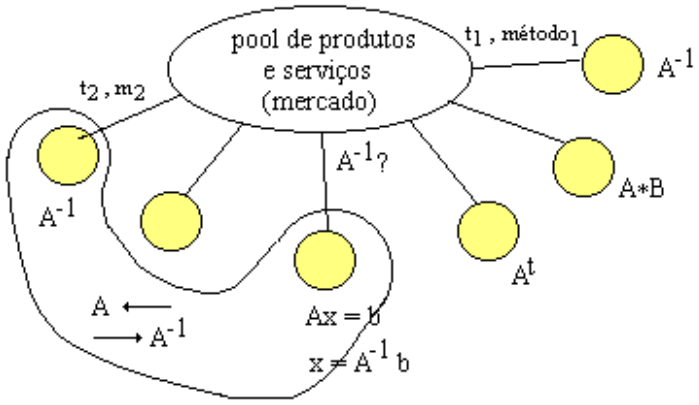
TIME ASSÍNCRONOS

eg. a) minimizar funções;
b) caixeiro viajante.



Quanto mais agentes, menos tempo para chegar a uma solução

"CONTRACT NETS" (MERCADOS/SERVIÇOS)



via RPC, por exemplo.

3 fases :
- anúncio do serviço;
- recebe propostas;
- contrata (pode haver relação mais direta).

Modelo de Objetos

- a nível de linguagem de programação;
- a nível de sistema (distribuído).

Utilização de modelo de objetos

Dispor de:

- gerenciador de objetos (Camelot);
- linguagem de programação concorrente ou distribuída (Avalon);
- sistema operacional (com linguagem de programação embutida) (Art, Amoeba).

Modelos de objetos

- classificação/instanciação;

- herança;
- polimorfismo. eg. Print(...) /* p/quadrado, circulo, etc. */

Objetos

São instâncias de classes

Classe

É um conjunto de atributos (que irão definir o "estado" do objeto) mais um conjunto de operações (métodos) envolvendo os atributos.

```
Classe
{
        char rd[16];  /* identificação do sensor */
        double leitura;         /* conjunto de atributos (pode haver dados públicos */
        float frequência;       /* não acessados só por operações) */
        int estado;
        ....
        int le_sensor();        /* conjunto de operações (métodos) */
        double valor();
        ...
} sensor;

sensor S1;
S1.le_sensor();
```

Estrutura de objetos

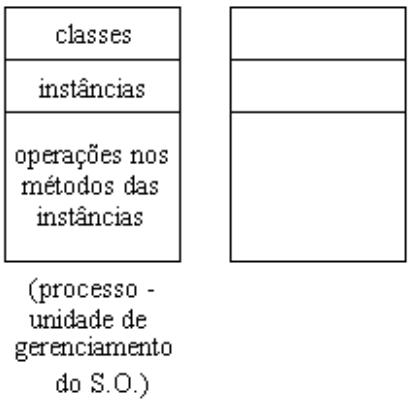


Figura 34 – Estrutura de objetos

Granularidade dos objetos

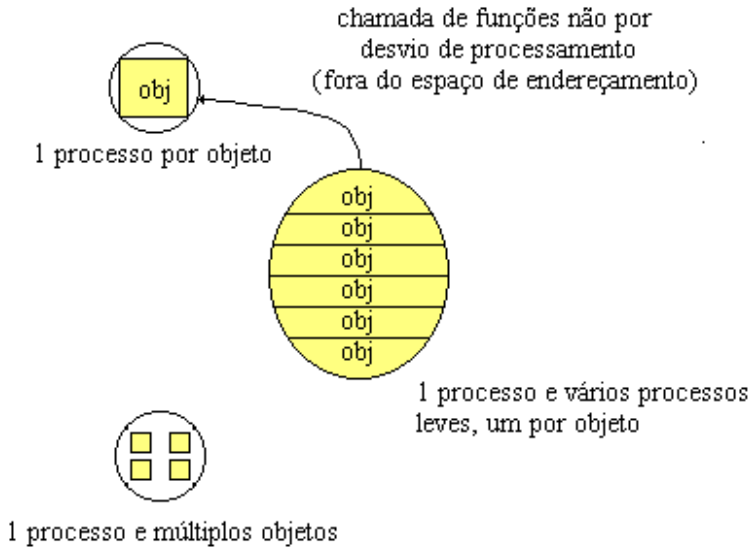


Figura 35 – Granularidade dos objetos

Modelo de objeto para Sistema Distribuído

- objetos passivos
  não define métodos – são apenas dados – não têm funções.

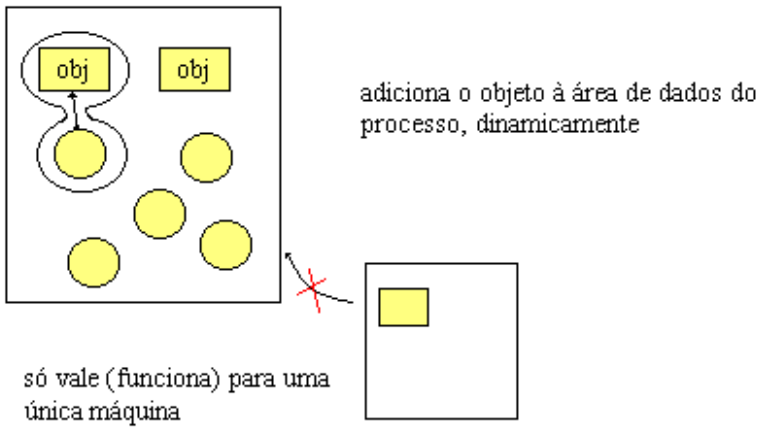


Figura 36 – Modelos de objetos (objetos passivos)

- objetos ativos
  os métodos possuem execução autônoma

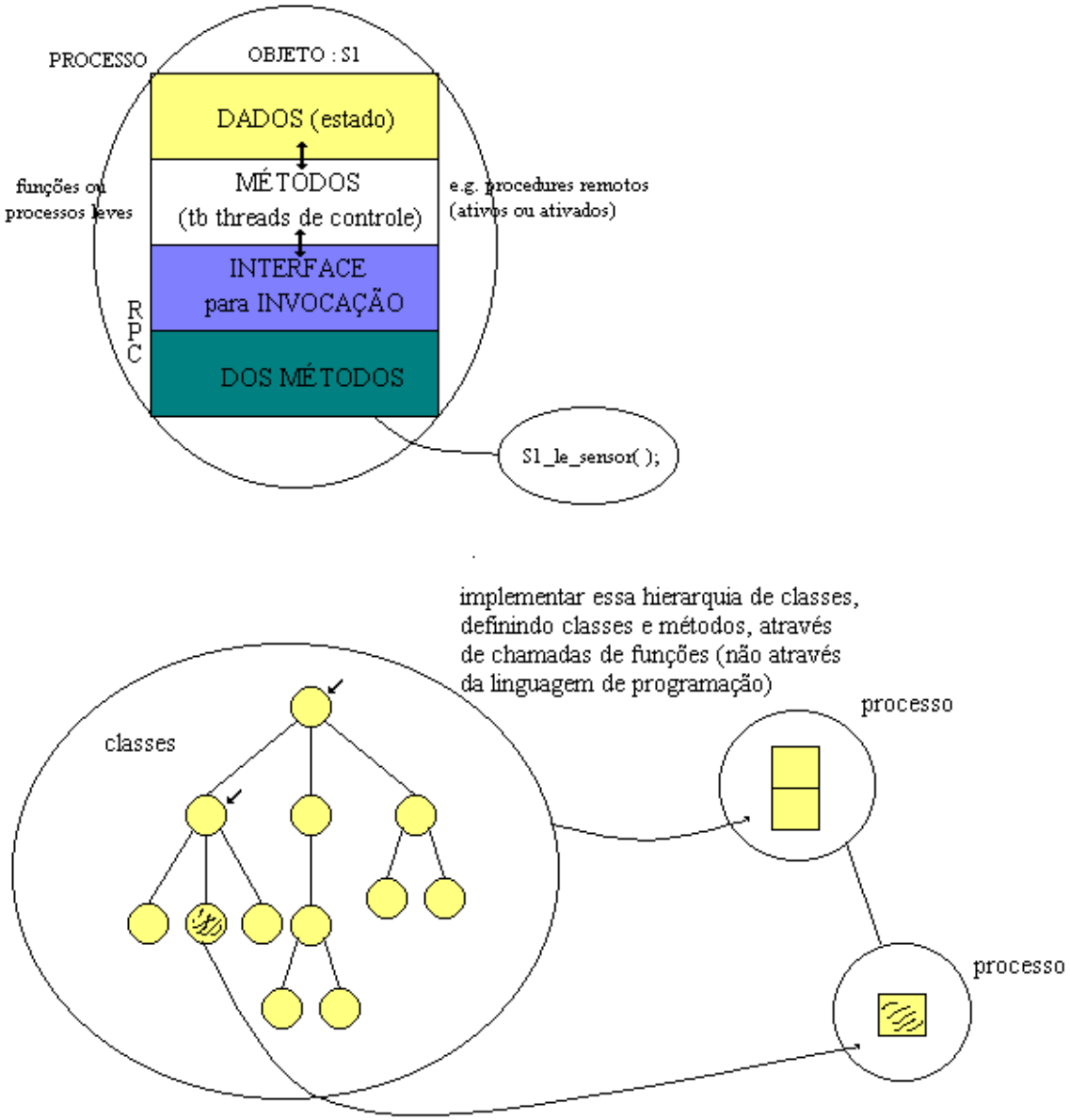


Figura 37 – Modelo de objetos (objetos ativos)

- servidor de classes (XYZ – atributos, derivados das classes wt ...)
  deve ser capaz de fornecer todos os atributos (métodos) da classe para quando for criar instâncias.
  Quando forem métodos, passar ponteiros para o arquivo executável (se fosse Lisp, passaria o próprio código, pois é difícil o tratamento de erros, por ex. com erros de compilação em C++).

Problemas

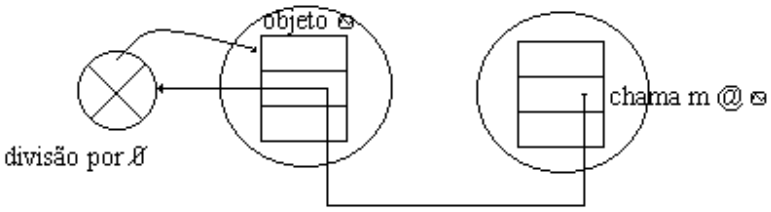
Chamada de métodos em paralelo – manter a consistência dos dados (≅ memória compartilhada – 1 método por vez, semáforo, lock por regiões, ...)

Interação inter-objetos

- Localização do objeto
  - Identificador de objeto (Servidor de nomes, host + nº do port que recebe RPC) <ID Global, Host, PORT>
  - nameserver (cache)
  - broadcast (cache) n últimas sequências.

Problema

Gerenciamento de falhas (ocorre nas 2 extremidades – quem chama e quem serve)



- tme-out
- probing (a cada intervalo de tempo, verifica se está ok).

Gerenciamento de recursos

- limitar o número de métodos concorrentes por processo (eg. Definido na classe);
- manter objetos em disco
  - gerenciador cria processo para executar o objeto quando for referenciado e não estiver executando.
- escalonamento de objetos
  - onde criar objetos (em que host?)
    - onde for referenciado;
    - no host especificado na criação;
    - no host menos sobrecarregado (+difícil de implementar).
- migração (geralmente só se migra dados e não processos)
  - no outro lado, cria-se um processo para abrigar o objeto, ou através de um gerenciador de objetos.

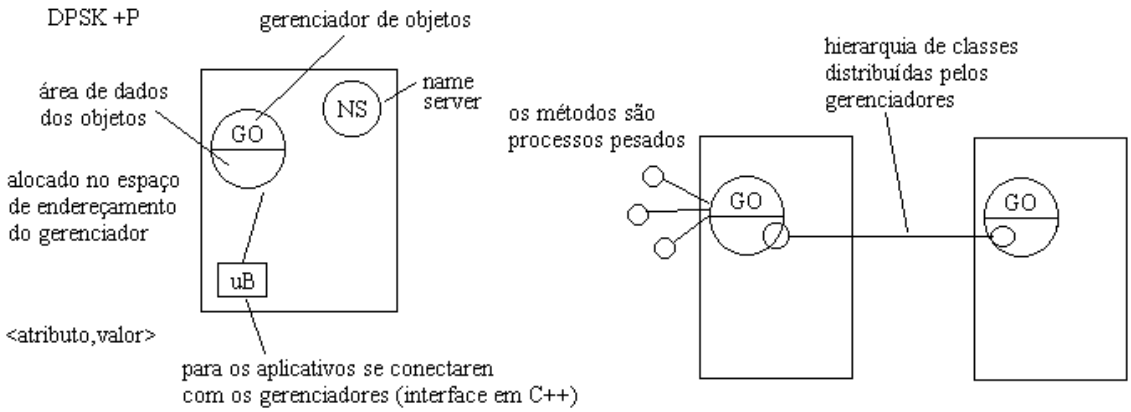


Figura 38 – Gerenciamento de recursos

Modelo de objetos para tempo real

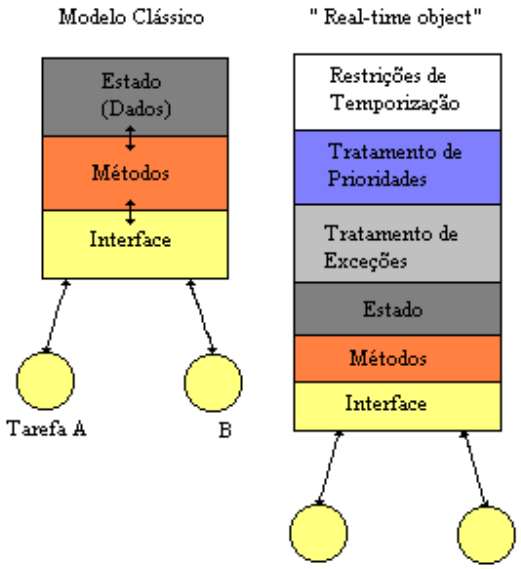


Figura 39 – Modelo de objetos para tempo real

Temporização

Problema : S.O. orientado para time-sharing (e.g. UNIX)

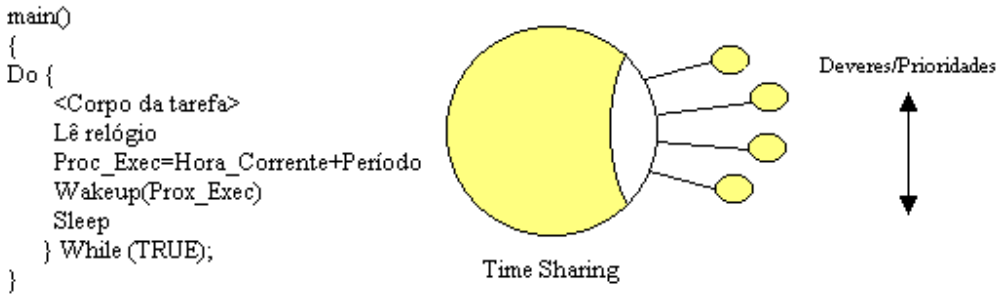


Figura 40 – Ilustração de processo de temporização

Escalonamento baseado em prioridades (não ser "prempted")

M_i : <Duração máxima>
M_i : <Início, Fim, período, deadline> métodos periódicos

Tratamento de prioridades

- associadas a clientes

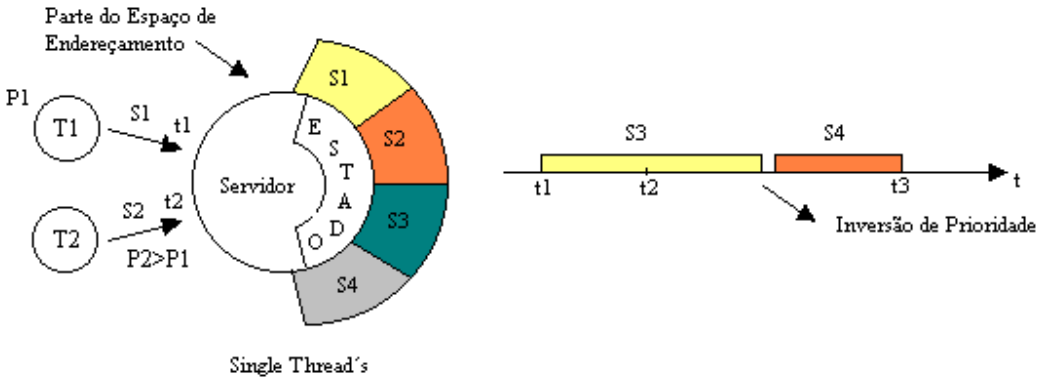


Figura 41 – Tratamento de prioridade associado a cliente

Estratégias para evitar a inversão de prioridade

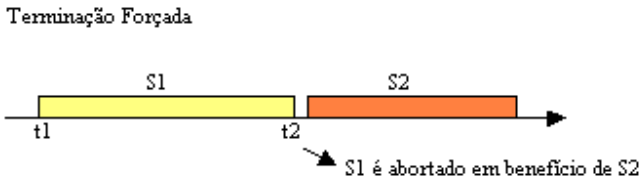


Figura 42 – Terminação forçada

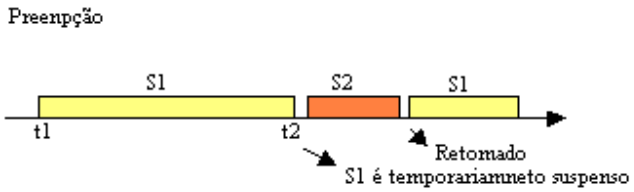


Figura 43 - Preempção

Problemas de terminação forçada e preempção : consistência do estado do objeto

Soluções possíveis : - objetos sem estado
- transação

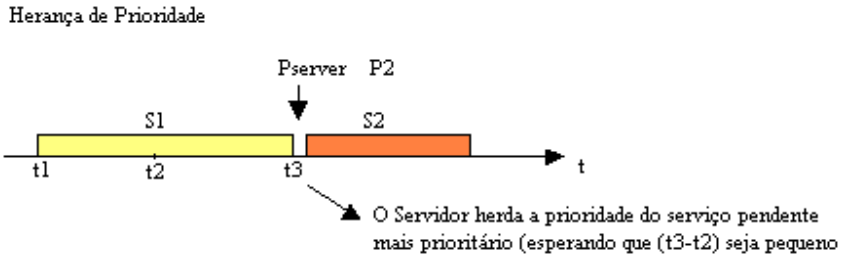


Figura 44 – Herança de prioridade

Não existe problemas com a consistência do estado, mas deve-se Ter meios de alterar a prioridade de processos em *run-time*.

"Multi-Threaded Objects"

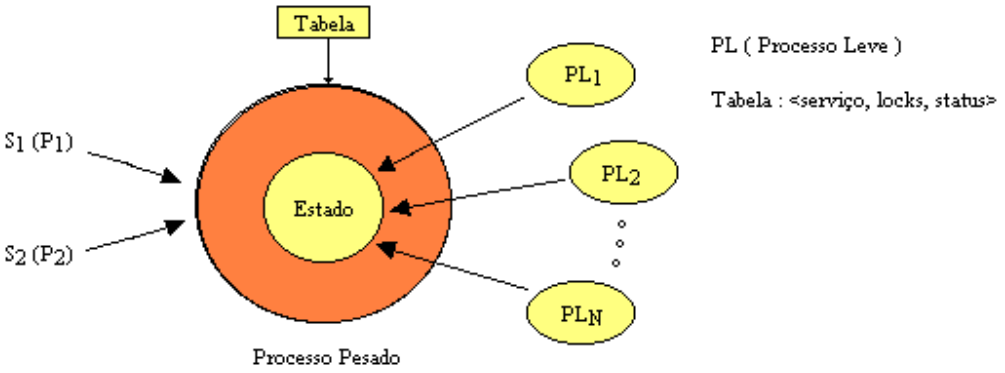


Figura 45 – "Multi-Threades Objects"

Tratamento de exceções

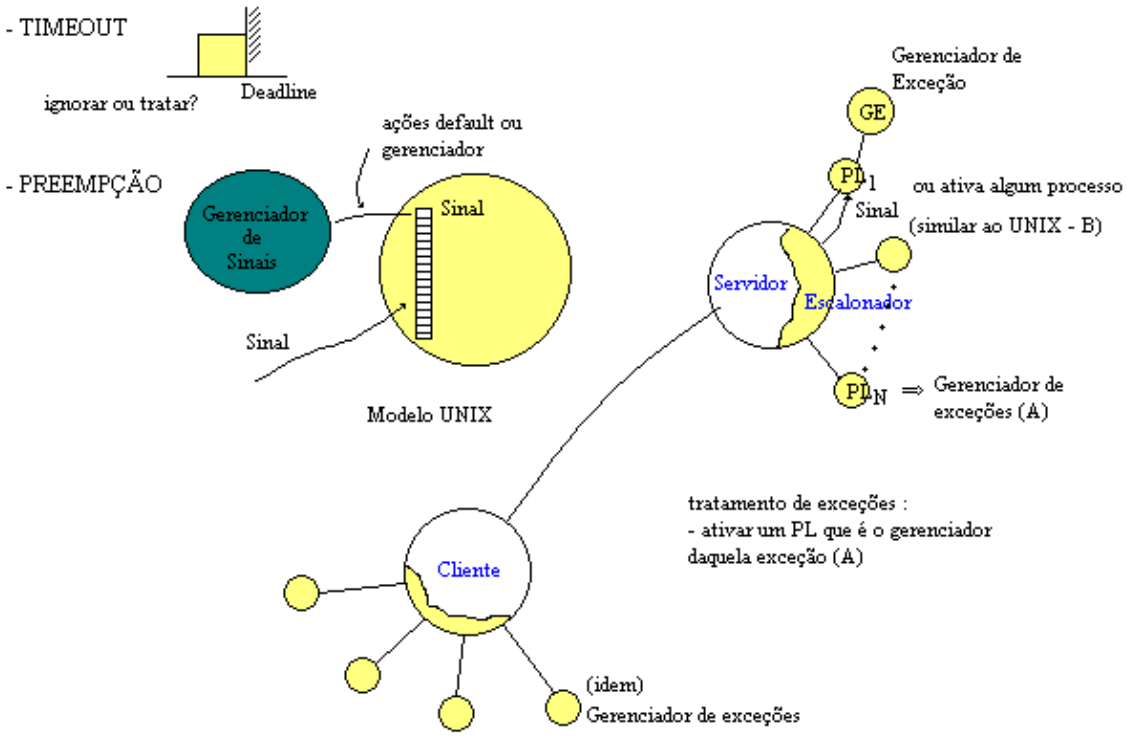


Figura 46 – Tratamento de exceções

Inversão de prioridade no subsistema de comunicação

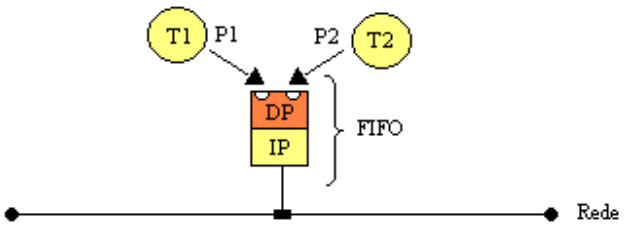


Figura 47 – Inversão de prioridade no subsistema de comunicação

Minimização do problema :

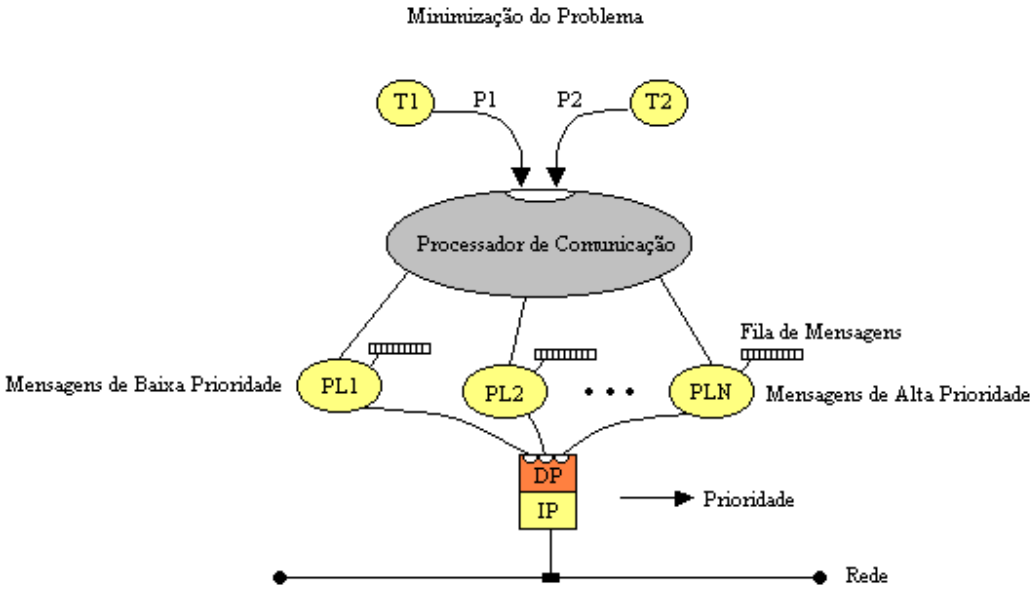


Figura 48 – Minimização do problema da inversão de prioridade no subsistema de comunicação

Servidores com processos leves

```
main() /*server*/
{
 <abre um socket DGRAM>
 LOOP
     recvfrom( _msg...);   ← bloqueia, mas quero que LP continue executando
     _LP(msg);               ⇒ biblioteca não bloqueante (LIBNBIO.A)
                             LLWP (biblioteca de processos leves)
}
```

Monitoramento de Sistemas Distribuídos

Dificuldade : estabelecimento de relação causa-efeito.
Duas fases :

- especificação (análise estática do S.D.)
  ex.: linguagem Estelle (ISO) – como um processo reage a determinados eventos causados por outros processos.
- monitoramento da execução (verificação dinâmica)
  técnica : introdução de "eventos" no S.D. (mensagens, acesso a dados compartilhados, mudança de estado dos processos, etc).
  eventos : - comunicação
  - controle
  deve prover aos eventos : - apresentação (eg. "display", print)
  - ordenação.

A WEB COMO SISTEMA DISTRIBUÍDO

Padrão iniciado pelo CERN (*Centre/Organisation Européenne pour la Recherche Nucléaire*), sede em Genebra, o mesmo do Colisor de Hádrons (Tim Berners-Lee em 1989 e aperfeiçoado por Robert Cailliau em 1990).

Desde 1994, a responsabilidade passou ao W3C (*World Wide Web Consortium*), uma colaboração entre o CERN e o MIT., e depois (2007) juntaram-se o INRIA (*Institut National de Recherche en Informatique et Automatique*) e a Universidade de Keio, do Japão.

ARQUITETURA E PROCESSOS

Lado **cliente**:
- browser;
- proxy (inicialmente para permitir ao browser executar protocolos de aplicação diferentes do HTTP. Hoje, usado para filtrar requisições e respostas, entrar em sistemas, comprimir arquivos, e armazenar). Problemas com proxies (RFC 3143);

Lado **servidor**:
- Presença grande de replicação: clusters com front end ("dispatcher") para redirecionar requisições de clientes a uma das réplicas. Três estratégias:
  - Distribuição de requisição por conteúdo: requisições para o mesmo documento ao mesmo servidor. Para não sobrecarregar o front end, um (outro) *dispatcher* repassa a conexão TCP ao servidor;
  - DNS de varredura cíclica, onde um único nome de domínio é associado a vários IPs. O Bind, servidor de DNS, por ex. move em círculo as entradas das listas de endereço ao enviar o RR – Registro de Recursos;
  - Broadcast: o front end repassa a requisição a todos, que decidem em conjunto qual servidor irá atender a requisição.

COMUNICAÇÃO

Baseada em HTTP, que por sua vez é baseado no TCP.

- HTTP 1.0: conexões não persistentes, i.e., cada requisição de cliente exigia o estabelecimento de uma conexão (RFC 1945);
- HTTP 1.1: conexões persistentes. Além disso, o cliente pode enviar várias requisições sem esperar resposta do servidor – pipeline (RFC 2616);

- HTTPS: autenticação do servidor e/ou cliente por criptografia assimétrica (RFC 2246, 3546);
- SOAP: para comunicação com serviços (Web Services), baseada em XML (RFC 3076).

SINCRONIZAÇÃO

Poucos trabalhos em sincronização.

- Autoria Distribuída de Documentos: feita através do protocolo WebDAV (Web Distributed Authoring and Versioning) (RFC 2291, 3744) :
  - lock de escrita exclusiva;
  - lock de escrita compartilhada (não há checagem de consistência. Nesse caso, espera-se que os autores modifiquem partes diferentes do documento);
  - não há necessidade de manter a conexão entre cliente e servidor. Isso quer dizer que se o cliente com o lock cair, o servidor deve retomar o lock. Mas esse procedimento não é especificado.

CACHING

Possibilidades (RFC 3040):

- cache compartilhada: resultados de consultas de um cliente repassados a outros clientes;
- cache hierárquica: seguindo a árvore de nomes (domínios). Alta latência por verificar em vários caches;
- cache distribuído: verifica nos proxies vizinhos ou repassa ao servidor Web caso não encontre;
- campo Time to Live do RR para endereços;
- get if-modified-since: dado pelo proxy ao servidor Web (get,post,delete);
- com replicação total: todo o documento é mantido em cache (útil se a taxa de atualização for baixa);
- com replicação parcial;
- cache ciente de conteúdo: manutenção de um BD local com as tabelas mais usadas (para servidores de borda);
- cache alheia ao conteúdo: o servidor de borda calcula um valor de *hash* para a consulta, usado para armazená-la;
- estudos para documentos dinâmicos.
- Estratégias LFU, LRU, função.

REPLICAÇÃO

Dados requisitos de desempenho, disponibilidade e custo, é importante estimar quantas réplicas são necessárias (muitas réplicas aumentam o desempenho, mas também o custo e a necessidade de banda) (RFC 3040).

A replicação soluciona problemas que não podem ser resolvidos unicamente através dos caches, pela dificuldade desses armazenarem conteúdo não estático, por exemplo. Assim, é necessário considerar:

- quantas réplicas serão instaladas;
- sua localização;
- quando instalar novas réplicas, dinamicamente.

Algumas métricas para estimar:

- métricas de latência / tempo de resposta: baseadas nos tempos para realizar operações, como recuperar documentos;
- métricas de banda: medição de largura de banda disponível entre dois nós;
- métricas espaciais: baseadas em distância, usualmente em número de hops;
- métricas de utilização de rede: largura de banda consumida;
- algoritmos de aprendizagem, que usualmente tomam por base resultados e decisões anteriores (Baentsch);
- algoritmos de otimização (Awerbuch).

Acesso às réplicas

- geração de réplicas por geração de processos, que podem ser criados e disparados local (fork) ou remotamente (através de front end) (Rabinovich);
- políticas de redirecionamento
  - DNS de varredura cíclica;
  - Políticas anteriores (ex. métrica baseada em distância);
- geração de réplicas por previsor de multidão instantânea (MI): uma MI é uma rajada de requisições a um determinado documento.O previsor dá tempo ao servidor Web para instalar novas cópias (usualmente baseados em métodos de extrapolação);
- replicação parcial de servidores/dados, tal como em cache, mas por prazos mais longos.

COORDENAÇÃO

Algumas abordagens:

- coordenação direta: quando os processos estão online e podem ser acessados diretamente (via mensagens, por exemplo);

- coordenação indireta: do tipo mailbox;
- coordenação por reunião: também síncrona, mas o acesso é feito indiretamente, através de uma estrutura (mesa de reuniões);
  - blackboard;
  - publicar/subscrever: os processos concordam em receber mensagens e as publicam. Podem permanecer anônimos (SD: Jini/ Sist.Coodenação: JavaSpaces)
  - mercado.
- comunicação geradora: dados compartilhados com equipamentos móveis. Os processos têm espaços de dados próprios, que podem ser compartilhados por proximidade. Também é possível trocas mensagens multicast e associa-los a um grupo.

TOLERÂNCIA A FALHAS

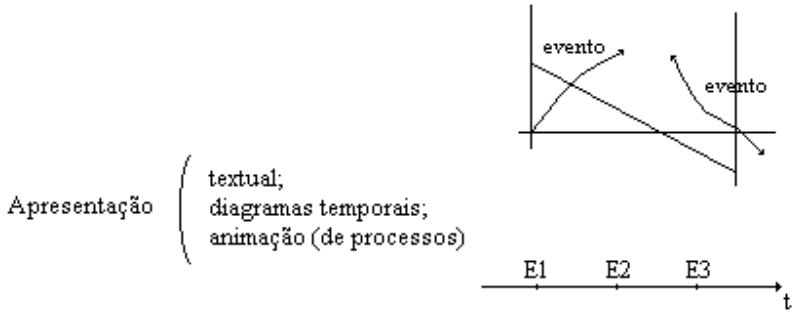
Algumas abordagens (nenhuma RFC !!!):

- usualmente obtida através de replicação de servidores e caches, isto é, por redundância;
- problemas com invocação em níveis, como nos Web Services, onde servidores podem requisitar serviços de terceiros. Uma solução é por replicação de chamadas, mas pode ser muito caro;
- tolerância falhas bizantina: chegar a um consenso (por votação e maioria) pode exigir muitas chamadas a servidores e ser muito cara;
- mensagens multicast enviadas aos vizinhos para reconstruir dados perdidos.

SEGURANÇA

A abordagem predominante é baseada em canais e transações seguras (RFC 2084):

- camada de sockets seguros (SSL) (RFC 3207);
- protocolo de segurança na camada de transporte (TLS) (RFC 4346, 5246), como HTTPS.



Relógio Lógico

É uma marca (número inteiro) associada a cada evento.
Cada processo mantém um único relógio lógico e adiciona o seu conteúdo aos eventos gerados pelo processo.

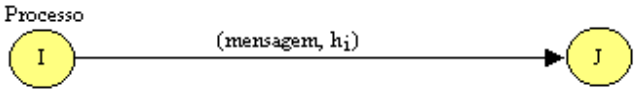
Notação :

$a \rightarrow b$	o evento a ocorreu antes (sentido físico) que o evento b
$a \parallel b$	os eventos a e b são independentes (concorrentes) (não quer dizer que um não influencie o outro, mas o protocolo não consegue identificar a causalidade (para fins de ordenação)) - não existe causalidade.

Protocolos para atualização dos relógios lógicos

Tempo linear (relógio de Lamport)

É feita uma ordenação total (ié, não se consegue determinar concorrência entre eventos)
h_i : relógio lógico do processo i



Protocolo:

R_1 : Na produção de um evento pelo processo : $h_i = H_i + d$ (1).
R_2 : Quando o processo i recebe uma mensagem, ele atualiza o relógio:
$h_i = \max(h_i, h_{mensagem})$
[e-executa R_1] se a recepção de mensagem também for um evento

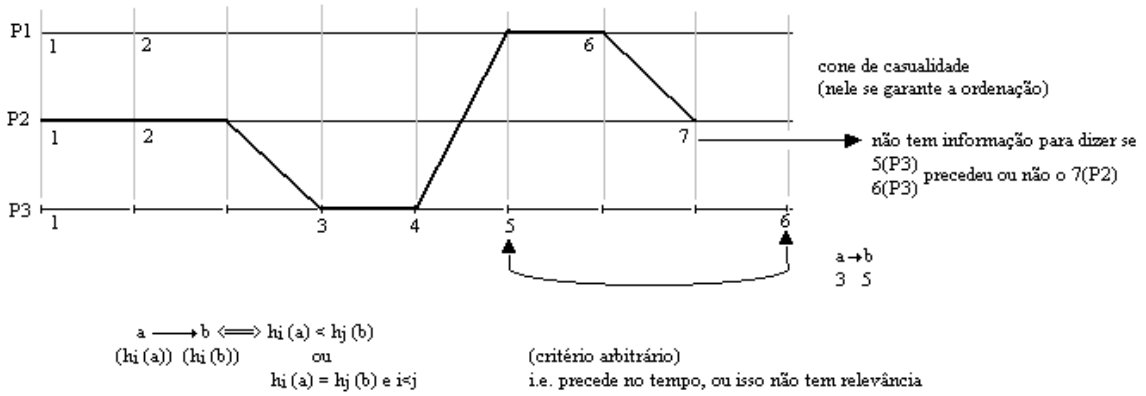


Figura 49 – Diagrama representativo do processo de atualização do relógio

Propriedade :

Se eventos não possuem relação causal, então sua ordenação é arbitrária (o protocolo não diz se a ordenação foi casual ou não, o que pode ser desejável).

Tempo matricial

$m_i[i,i]$: relógio local ao processo i
$m_i[k,l]$: visão do processo i sobre o conhecimento que o processo k possui sobre a noção de tempo no processo l.

R_1 : Na produção de um evento pelo processo i
$m_i[i,i] = m_i[i,i] + d$
R_2 : Quando o processo i recebe uma mensagem de j, o processo i atualiza o relógio:
$m_i[i,k] = \max(m_i[i,k], m_j[j,k])$,
$m_i[k,l] = \max(m_i[k,l],m_j[k,l])$, k,l=1,... N
[e executa R_1]

Propriedade :

Se $\min k (m_i[k,i]) = t$ então o processo i sabe que todos os processos estão informados que seu relógio atingiu t (pode descartar informações antigas).

Relógios físicos

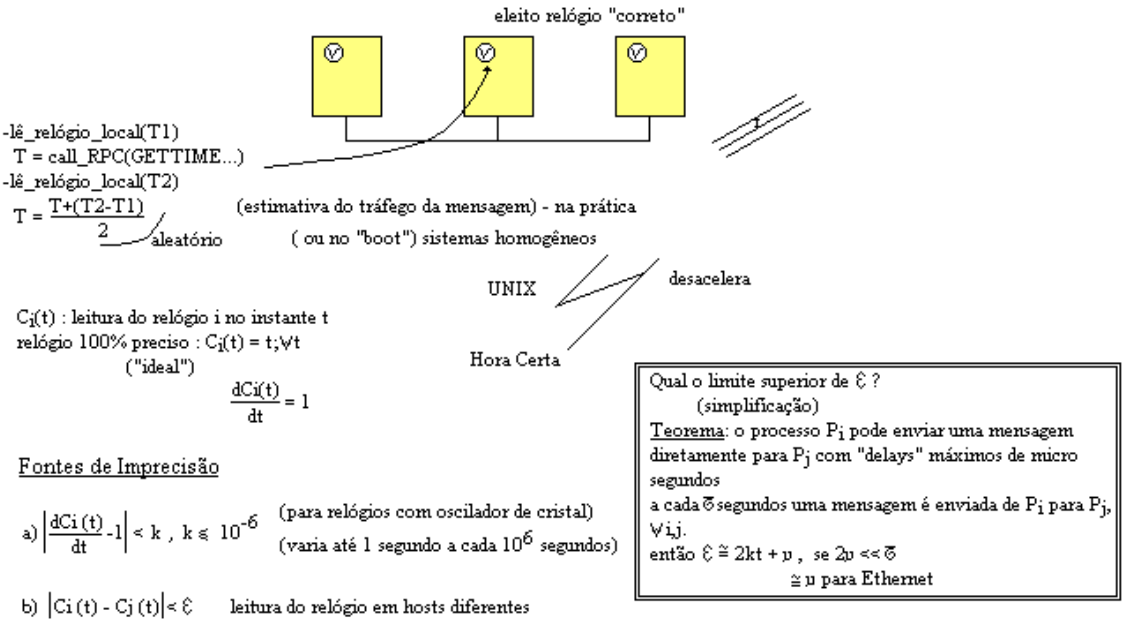


Figura 50 – Relógio físico

Núcleo de Aplicações de tempo real

Tendência de padronização do UNIX : IEEE POSIX

- interface (system calls)
- shell e ferramentas (eg. Correio eletrônico)
- procedimentos de testes (benchmarks)
- tempo real (em andamento)
- ADA
- Segurança
- Administração
- Networking

O que é recomendado pela IEEE:

- timers de alta resolução
- escalonamento por prioridades
- memória compartilhada
- “real time files” (não “buferizar” writes, por ex.)
- semáforos
- comunicação inter-processos
- notificação assíncrona de eventos
- fixação de processos em memória
- I/O síncrono e assíncrono (eg. Read não bloqueado)

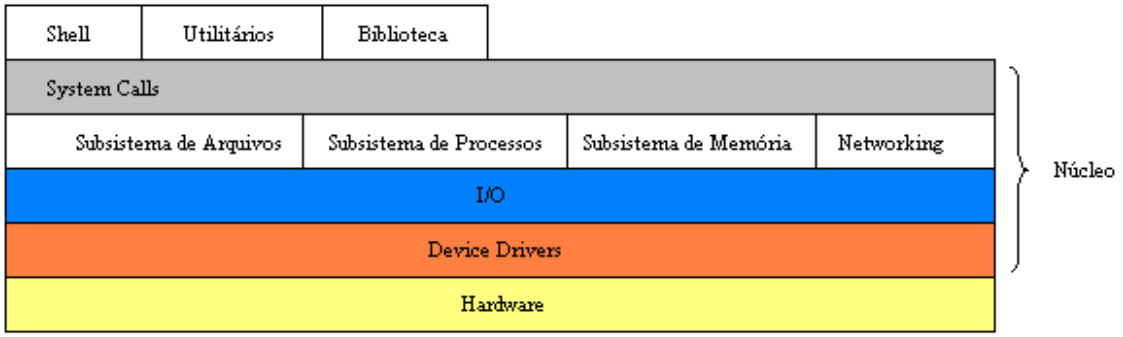


Figura 51 – Recomendação IEEE

[SYSTEM CALLS:]

O núcleo deve ser totalmente "preemptíve" (senão inversão de prioridades)

[Subsistema de Arquivos]

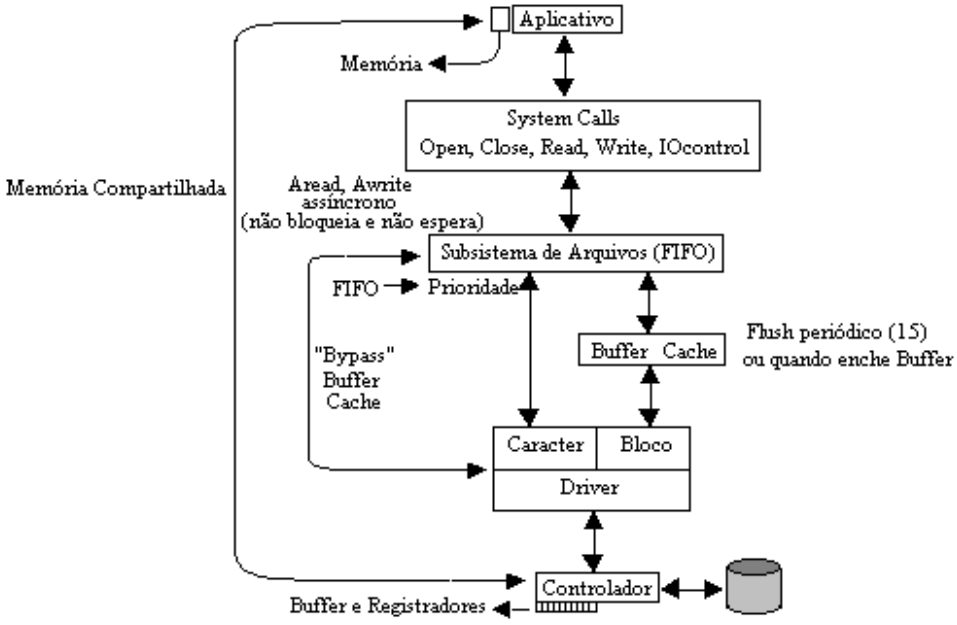


Figura 52 – Subsistema de arquivos

Representação interna de arquivos (por blocos) :

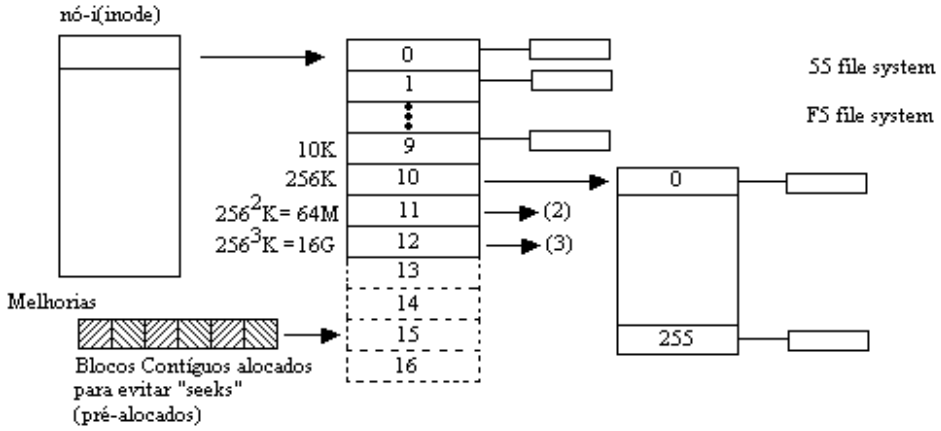


Figura 53 – Representação interna de arquivos

[Subsistema de Processos:]

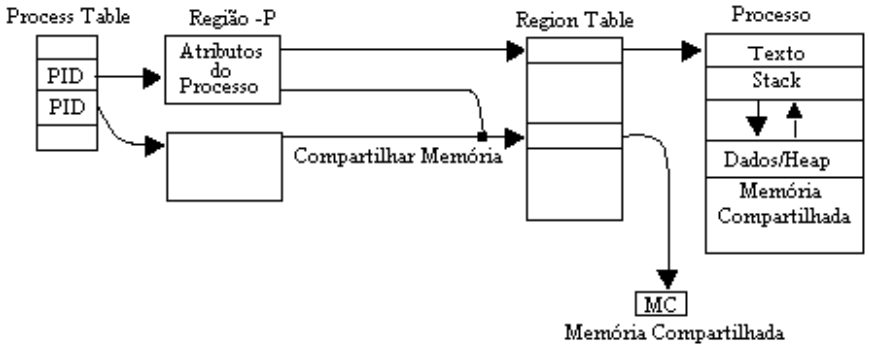


Figura 54 – Subsistema de processos

- Escalonamento
  - Time sharing
  - Prioridade : F(CPU_utilizada, recursos consumidos, prioridade "estática")

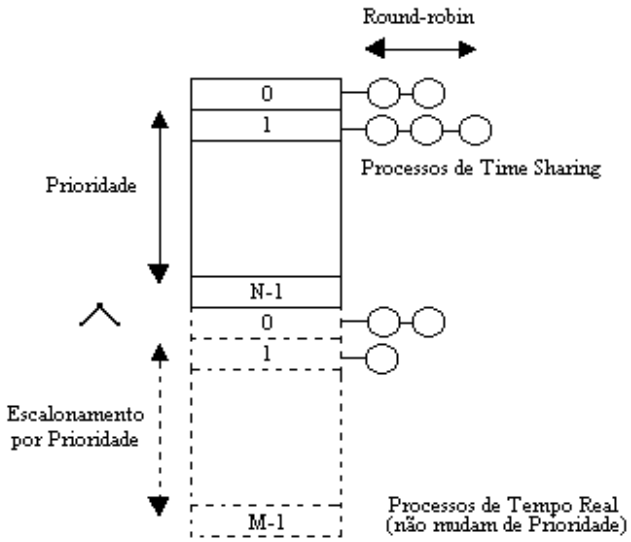


Figura 55 – Subsistema de processos

[Subsistema de Memória:]

- versões antigas (<system V; <4.2 BSD) : swapping

Figura 56 – Subsistema de memória (Versões antigas)

- versões atuais : paginação por demanda

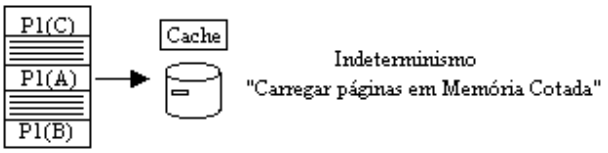


Figura 57 – Subsistema de memória (Paginação por demanda)

Melhoria :

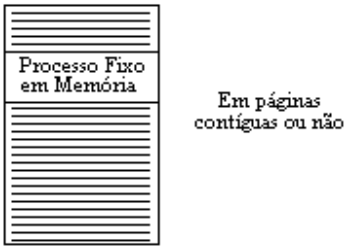


Figura 58 – Subsistema de memória (Melhoria)

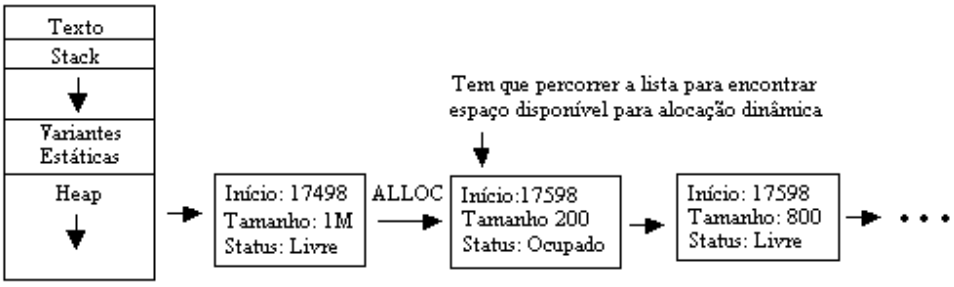


Figura 59 – Subsistema de memória

Char *buff
Buff = malloc (200)

Char buff [200]; evita busca linear na lista

Ao invés de 200, utilizar “500” – pior caso.

[Mecanismo de Interrupção:]

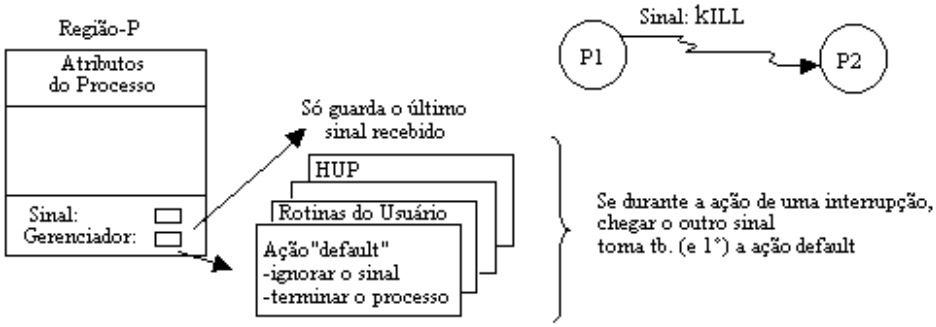


Figura 60 – Mecanismo de interrupção

Melhorias :

Mecanismo de eventos (podem ser definidos também pelo usuário)

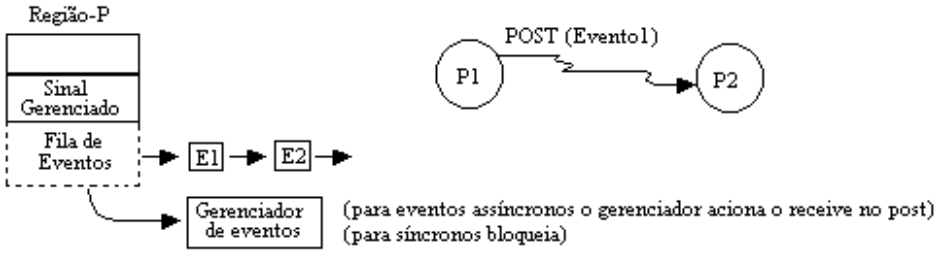


Figura 61 – Mecanismo de eventos

Threads

Uma ***thread*** conceitualmente é um tipo de dados que representa um fluxo de controle. Às *threads* deve ser associado um conjunto de operações de manipulação, incluindo formas de controlar seu escalonamento e comunicação. Um **processo leve** representa uma *thread* de controle não intimamente ligado a um espaço de endereçamento.

Threads normalmente podem executar mais rapidamente que processos normais. Isso porque sua comunicação se dá através de memória compartilhada ao invés de sistema de arquivos. A disponibilidade de uso de processos leves provê uma abstração adequada para a implementação de programas que rejam a eventos assíncronos (como servidores, por exemplo). São adequados também para a implementação de programas de simulação que modelem situações concorrentes.

Pode-se considerar a abstração de processos leves superior à de sinais. No sistema Unix, por exemplo, um sinal ocasiona uma espécie de troca de contexto, para uma nova instrução, por exemplo, e as regiões podem ser implementadas desabilitando interrupções. Com processos leves, a única forma de tratar eventos assíncronos é através de *threads*. Não existem exceções assíncronas em uma *thread*. Seções críticas são implementadas através de monitores, não havendo necessidade de "travar" interrupções, com a possibilidade de perda de informações enquanto a região crítica estiver sendo processada.

Funcionalidade

As bibliotecas referentes à processos leves usualmente provêem primitivas para manipulação de *threads* e controle de eventos em um processador. Várias delas fornecem suporte apenas para processos a nível de usuário. Nesses casos, a fatia de tempo alocada pelo sistema operacional a um processo deve ser compartilhada por todas as *threads* que pertencem àquele processo. Além disso, seus objetos não são acessíveis fora desse processo.

Exemplos de primitivas que devem ser suportadas[1]:

- Criação e destruição de *threads*, acesso a informações de *status,* gerência do escalonamento, suspensão e retomada da execução;
- Multiplexação do relógio (vários processos podem dormir concorrentemente);
- Mudança de contexto individualizada;
- Monitores e variáveis de condição para sincronização de *threads*;
- Extensão do conceito de *rendez-vous* (envio, recepção e resposta de mensagens) entre *threads*;
- Facilidades de manipulação de exceções (ex. notificação e escape);
- Formas de mapear interrupções em *rendez-vous*;
- Formas de mapear *traps* em exceções;
- Checagem de integridade de pilha, para ambientes onde não existam mecanismos sofisticados de gerenciamento de memória.

O escalonamento, por *default*, é baseado em prioridade e não preemptivo dentro da mesma prioridade. Quando um conjunto de *threads* estão executando, assume-se que eles todos estejam compartilhando memória.

Fundamentos de implementação

O mecanismo de processos leves permite que várias *threads* de controle compartilhem o mesmo espaço de endereçamento. Cada processo leve é representado por um procedimento que será convertido em uma *thread* através da primitiva **lwp_create()**. Quando uma *thread* é criada, ela passa a ser uma entidade independente com sua própria pilha. A primitiva *lwp_create()* aloca um contexto, inicializa uma pilha, e torna a *thread* pronta para ser chamada para execução. Uma coleção de *threads* executam em um único processo comum. Essa coleção é denominada ***pod***.

Os processos leves ou *threads* são escalonados por prioridade. Assim, a *thread* não bloqueda de mais alta prioridade é quem deve estar executando a cada momento. Elas pode bloquear em algumas circunstâncias, como para a chegada de uma mensagem ou a requisição de um *lock* de monitor. Quando possuírem a mesma prioridade, as *threads* executam com base na política FIFO (fila). Assim, se duas *threads* forem criadas com a mesma prioridade, elas devem executar na ordem de criação.

```
/* Programa que ilustra a criação de uma thread  simples.
Quando a thread main() termina, task() executa e termina
também. A biblioteca de processos leves identifica que  não
há mais threads, e o processo termina. */

#include <lwp/lwp.h>
```

[1] SunOS, em particular.

```
#include <lwp/stackdep.h>
#define MAXPRIO 10

main(argc, argv)
     int argc;
     char *argv;
{
     thread_t tid;
     int task();

     printf("main here\n");

     /* Transforma main() em um processo leve, e especifica a
prioridade máxima de escalonamento. main() executa com
prioridade 10, e prioridades 1..10 ficam disponíveis. Assim,
main() vai executar até que seja bloqueado ou passe o
controle a outra thread. */
     (void) pod_setmaxpri (MAXPRIO);

     /* Inicializa um cache de pilhas, a ser usado por
futuras chamadas lwp_newstk(). Cada uma dessa chamadas deverá
retornar uma pilha de pelo menos 1000 bytes, sendo que o
cache deverá conter 2 pilhas. Muitas pilhas requerem mais
memória, mas causam menos falhas de pilhas. No caso de falha,
um cache de mesmo tamanho será alocado. Quando uma thread
morrer, sua pilha é automaticamente liberada. */
     lwp_setstkcache (1000, 2);

     /* Cria uma thread que iniciará sua execução em task(),
com prioridade 10 de escalonamentoe e usar o cache de pilhas.
Sua identidade é retornada em tid. */
     lwp_create (&tid, task, MAXPRIO, 0, lwp_newstk(), 0);
}

task() {
     printf("hello world\n");
}
```

O comando para compilar o programa é o seguinte:

cc –o prog prog.c –llwp

Co-rotinas

É possível utilizar *threads* como co-rotinas puras, onde uma *thread* outorga explicitamente controle a outra. A primitiva **lwp_yield()** permite que uma *thread* aguarde uma *thread* específica de mesma prioridade, ou a próxima *thread* de mesma prioridade.

```
/* Programa que ilustra o uso de 3 co-rotinas: main(),
coroutine() e other(). O resultado é a impressão dos números
de 1 a 7, em sequência. */

#include <lwp/lwp.h>
#include <lwp/stackdep.h>

thread_t co1;   /* identificador da thread principal */
thread_t co2;   /* identificador da coroutine */
thread_t co3;   /* identificador de other */

main(argc, argv)
     int argc;
     char ** argv[];
{
     int coroutine(), other();

     lwp_self(&co1);

     lwp_setstkcache (1000,3);
     lwp_create (&co2, coroutine, MINPRIO, 0, lwp_newstk(),
0);
     lwp_create (&co3, other, MINPRIO, 0, lwp_newstk(), 0 );
     printf ("1\n");
     lwp_yield (THREADNULL);  /* yield to coroutine */
     printf ("4\n");
     lwp_yield (co3);  /* yield to other */
     printf ("6\n");
     exit(0);
}

coroutine()
     {
     printf ("2\n");
     if (lwp_yield(THREADNULL) < 0) {
          lwp_perror ("bad yield");
          return;
          }
     printf ("7\n");
     }

other()
```

```
     {
     printf ("3\n");
     lwp_yield (THREADNULL);
     printf ("5\n");
     }
```

Escalonadores customizados

Há três formas de prover controle de escalonamento ao cliente. A primeira é não fazer nada, e prover ao cliente simplesmente um ponteiro para um contexto de *thread* que pode ser escalonada à vontade dele. O grande empecilho deste método é que a maioria dos clientes não desejam construir seus próprios escalonadores.

A segunda forma consiste em prover uma única política de escalonamento, onde o cliente possui muito pouco controle sobre o que será executado na seqüência. Um exemplo dessa política é a sistema operacional Unix. Essa abordagem é a mais simples do ponto de vista do cliente, mas torna difícil implementar políticas que levem em conta os diferentes tempos de resposta das *threads* clientes.

A terceira forma consiste em prover primitivas que podem interferir na política de escalonamento padrão, permitindo a construção de diferentes políticas de escalonamento.

Exemplos de tais primitivas são **lwp_suspend(), lwp_yield(), lwp_resume(), lwp_setpri()** e **lwp_resched()**. A primitiva *lwp_suspend()* pode também ser utilizada em processos de correção de erros (*debugging*), para assegurar que uma *thread* tenha sido interrompida, antes de analisá-la.

```
/* Exemplo de um programa que constrói um escalonador round-
robin, time-sliced. Uma thread de mais alta prioridade atua
como escalonador e dorme pelo quantum de tempo desejado. */

#include <lwp/lwp.h>
#include <lwp/stackdep.h>
#define MAXPRIO 10

main(argc, argv)
     int argc;
     char **argv;
{
     int scheduler(), task(), i;
     (void) pod_setmaxpri(MAXPRIO);
     lwp_setstkcache (1000,5);
```

```
     (void) lwp_create((thread_t *)0, scheduler, MAXPRIO, 0,
lwp_newstk(), 0);
     for (i=1; i<3; i++)
          (void) lwp_create((thread_t *)0, task, MINPRIO, 0,
lwp_setstk(), 1 , i);
     exit(0);
}

scheduler()
     {
     struct timeval quantum;

     quantum.tv_sec = 0;
     quantum.tv_usec = 10000;
     for(;;) {
          lwp_sleep(&quantum);
     lwp_resched(MINPRIO); }
     }

/* Tarefas escalonadas round-robin, preempted */
task(arg)
     {
     for(;;)
          printf("task %d\n", arg);
     }
```

Trocas de contexto especiais

Uma *thread* pode fingir estar executando sozinha em uma máquina, mesmo que várias *threads* estejam executando. Isso pode ser realizado através de primitivas da biblioteca *lwp.h*. Essa biblioteca provê também **trocas de contexto** entre *threads*, que fazem com que recursos voláteis de máquina sejam multiplexados, de forma que cada *thread* opere com seu próprio conjunto de recursos de máquina. Em muitos casos, uma troca de contexto somente requer que os registradores e a pilha sejam multiplexados. A biblioteca permite que as *threads* possuam quantidades diferentes de estados para que processos com necessidades diferentes de recursos possam coexistir.

Uma *thread* possui também possui estados que podem ser modificados por outras primitivas. Esse estado privativo de cada *thread* inclui trocas de informações na forma de mensagens. O espaço disponível para armazenamento desses estados é bastante limitado, e somente as *threads* que necessitem de estados adicionais devem utilizá-los. Assim, não estão disponíveis informações como estados dos sinais, informação de *accounting* ou descritores de arquivos nesse contexto.

A biblioteca de *threads* aloca um novo *buffer* de contexto para cada novo contexto inicializado com uma *thread*, e disponibiliza um ponteiro para esse contexto para salvar e restaurar rotinas definidas nesse contexto.

Para utilizar esse mecanismo de contexto, é necessário primeiro definir um contexto especial, através da primitiva **lwp_ctxset()**. Isso requer que sejam definidas as formas de salvar e restaurar os estados do contexto, através de procedimentos para isso.

Uma vez que o contexto tenha sido definido, pode-se inicializar *threads* que utilizem o recurso multiplexado pelo contexto especial, usando a primitiva **lwp_ctxinit**(). A inicialização de uma *thread* que utilize um contexto especial pode ser feita diretamente ou, se os recursos permitirem, através de uma *trap*, quando o recurso for usado por uma *thread* pela primeira vez.

```
/* Programa que exemplifica a multiplexação e uso de recursos
em threads. */

typedef struct libc_ctxt_t {
     int libc_errno;
     } libc_ctxt_t;

static int LibcCtx;

/* Permite contextos especiais em libc */
libcenable()
{
     extern void libc_save();
     extern void libcrestore();

     LibcCtx = lwp_ctxset (_libc_save, libcrestore, sizeof
(libc_ctxt_t), TRUE);
}

/* Faz com que uma thread tenha o contexto libc */
lwp_libcset(tid)
     thread_t tid;
{
     (void) lwp_ctxinit(tid, LibcCtx);
}

/* Rotinas para salvar/restaurar dados */
```

```
void _libc_save (cntxt, old, new)
     caddr_t cntxt;
     thread_t old;
     thread_t new;
{
     extern int errno;

#ifdef lint
     old=old;
     new=new;
#endif lint

     ((libc_ctxt_t *)cntxt)-> libc_errno = errno;
}

void _libc_restore (cntxt, old, new)
     caddr_t cntxt;
     thread_t old;
     thread_t new;
{
     extern int errno;

#ifdef lint
     old = old;
     new = new;
#endif lint

     errno = ((libc_ctxt_t *) cntxt) -> libc_errno;
}
```

Referências bibliográficas

1. "Networking Programming"- Manual da SUN.

2. BALL,H.E., STEINER,J.G., TANEMBAUM,A.S. "Programming Languages for Distributed Computing Systems", ACM Computing Servers, vol 21 nº 3, Set. 1989.

3. COULOUNS,G.F., DOLLEMORE, J. "Distributed Systems Concepts and Design", 1988.

4. FURTH, B. et al, KLUWER. "Real Time Unix Systens", Academia Publisher, 1991.

5. J.E.B. MOSS. "Nested Transactions Na Approach to Realiable Distributed Computing", MIT Press, 1989.

6. LAMPORT, L. "Time. Clocks and the Ordering of Events in a Distributed System", Communication os the ACM, Vol. 21, nº 7, Julho 1978.

7. NITZBERG,B.& LO,V. "Distributed Shared Memory. A servey and Algorithms", IEEE Computer, Agosto 1991.

8. SILBERSCHATZ, A. PETERSON, J. GALVANI, P. “Operating Systems Concepts”, 3º ed, Addison-Westey, 1991.

9. STUM,M.& ZHOW,S. “Algorithims Implementing Distributed Shared Memory”, IEEE Computer, Maio 1990.

10. TANEMBAUM, Peterson. “Livros de SO”.