TRIX, um Sistema Operacional Multiprocessado para Transputers

Marcelo Pazzini*
Philippe Navaux†

Resumo

Este artigo descreve a implementação de um sistema operacional para controlar diversos processadores do tipo transputer. Esse sistema, chamado TRIX, foi desenvolvido tomando por base os fontes do sistema operacional MINIX, que é compatível com UNIX.

O TRIX possui uma arquitetura em camadas, com um micro-*kernel* na camada mais interna implementando processos seqüenciais que se comunicam por mensagens. Chamadas ao sistema e operações de entrada e saída em dispositivos são desempenhadas por servidores, que executam como processos independentes.

Aqui está descrita a estrutura do sistema, o seu micro-*kernel* e as características distribuídas dos processos servidores.

Abstract

This paper describes the development of an operating system to manage several processors of transputer type. This system, called TRIX, was build based on the source code of MINIX operating system, which is compatible to UNIX.

TRIX has a layered structure, with a micro-kernel in its innermost portion which implements message passing sequential processes. System calls and input-output operations are done by servers, whose are independant processes.

Here is described the system structure, its micro-kernel and the distributed features of the server processes.

Aluno de mestrado do CPGCC/UFRGS, bolsista da CAPES/PICD, professor do DELC/UFSM. Áreas de interesse: sistemas operacionais de paralelos, distribuídos e de rede. Endereço: Departamento de Eletrônica e Computação (DELC), UFSM Campus, 97119-900, Santa Maria, RS, e-mail: pazzini@inf.UFRGS.br

†*Professor do CPGCC/UFRGS, Doutor em Informática pelo Instituto Nacional Politécnico de Grenoble (França, 1979). Áreas de interesse: arquitetura de computadores, processamento paralelo, avaliação de desempenho. Endereço: Instituto de Informática da UFRGS, Cx. Postal 15064, 91509-900, Porto Alegre, RS, e-mail:* navaux@inf.UFRGS.br

1 Introdução

Muito se trabalha hoje com novas tecnologias para microeletrônica com o intuito de aumentar o desempenho dos sistemas de computação. A cada dois ou três anos, a freqüência do relógio dos processadores mais populares é praticamente dobrada. Não obstante tal progresso, o desempenho disponível atualmente não é suficiente para atender algumas demandas de processamento mais pesadas.

Para se poder alcançar maiores níveis de desempenho, sem depender necessariamente de processadores mais rápidos, faz-se uso de processamento paralelo. O trabalho computacional é desmembrado em diversas tarefas, que são executadas de forma concorrente. Para este tipo de trabalho, há um processador que já se mostrou bastante apropriado devido suas características intrínsecas de suporte a multiprocessamento: o transputer[INM90]. Existem inclusive algumas instituições de pesquisa em processamento paralelo que já desenvolvem sistemas operacionais para transputers (nem todos compatíveis com UNIX), como [DIS88], [SMI89], [MEI91] e [POU91].

Os principal atrativo que um transputer oferece para sistemas de processamento paralelo é o seu mecanismo próprio de processos seqüenciais comunicantes. Escalonamento de processos e troca de mensagens entre eles são implementados pelo *hardware* do processador[SHE87]. O escalonador possui duas prioridades e a troca de mensagens é do tipo *rendez-vous*, feita através de canais, tudo implementadas por instruções de máquina. Externamente, um transputer tem quatro *links* para conexão de outros processadores, que sob o ponto de vista do *software* se comportam como canais comuns de comunicação. Assim, dois processos transputer sempre se comunicam via canais, não importando se estão no mesmo processador ou em processadores distintos.

O trabalho em torno do projeto TRIX visa desenvolver um sistema operacional multiprocessado experimental, para servir de base a futuros trabalhos de pesquisa em sistemas operacionais e processamento paralelo[PAZ91]. Como características essenciais do sistema têm-se simplicidade, desempenho e compatibilidade com UNIX. Com o sistema operando, pode-se fazer experiências de muitos tipos em processamento paralelo, como por exemplo, ensaios de balanceamento de carga, avaliações de desempenho, implementação de linguagens, bancos de dados, sistemas dedicados, etc.

Ao construir o sistema TRIX, decidiu-se usar o código fonte de um sistema já pronto e funcionando, para encurtar o tempo de desenvolvimento. Optou-se pelo MINIX, que é um sistema bem estruturado, dividido em camadas de processos seqüenciais que se comunicam por troca de mensagens[TAN87]. Mais que isso, o MINIX é compatível com o UNIX versão 7 e o seu código fonte pode ser usado para fins acadêmicos sem ferir o seu *Copyright*. Foi necessário porém dotar o MINIX de características de distribuição, para o controle de uma arquitetura multiprocessada. A implementação destas características é o principal assunto deste artigo.

2 Estrutura do Sistema

No momento da escolha de um sistema base para o TRIX, foram levados em consideração alguns critérios: disponibilidade de programas fonte sem problemas de *Copyright*; modularidade do *kernel*; simplicidade; compatibilidade com UNIX. Sob todos estes aspectos o MINIX aparece como uma boa opção. Além disso, ele se adapta bem à arquitetura do transputer, por ser um sistema implementado como uma coleção de processos seqüenciais que se comunicam por mensagens. Será feito então um breve resumo sobre a estrutura do sistema operacional MINIX, seguido do detalhamento das novidades implementadas com o TRIX.

2.1 Estrutura do MINIX

O MINIX é dividido em quatro camadas, como mostra a figura 1, cada qual desempenhando uma atividade específica. A camada mais baixa, chamada *Process Management* é responsável por controlar as interrupções, o salvamento e a restauração de registradores. É esta camada também que implementa um mecanismo de troca de mensagens entre processos. Sua função é prover às camadas superiores um modelo de processos seqüenciais independentes, que se comunicam usando mensagens.

A segunda camada é formada pelos *drivers* dos dispositivos de entrada e saída (E/S). Cada dispositivo de E/S possui um processo de controle, denominado *task*. Todas as funções e processos das duas camadas inferiores são ligadas em um único módulo executável, denominado *kernel*. Embora as *tasks* estejam todas ligadas em um único módulo, compartilhando dados e parte do código, elas são escalonadas independentemente e se comunicam usando mensagens.

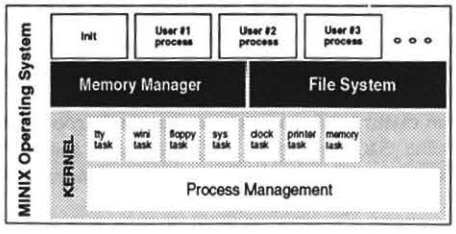


Figura 1: Estrutura do MINIX

A camada 3 é formada por dois processos: *Memory Manager* e *File System* (MM e FS). O primeiro interpreta todas as chamadas UNIX que dizem respeito o controle de memória e gerência de processos, do tipo `fork`, `exec` ou `brk`. O segundo, interpreta todas as chamadas referentes ao sistema de arquivos, como `open`, `read` ou `chdir`. O *File System* foi escrito para ser um servidor de arquivos, podendo ser deslocado para um processador remoto, com poucas alterações.

As três primeiras camadas implementam o sistema operacional propriamente dito. A quarta e última camada é formada pelos processos de usuário, como `shell`, editores, compiladores e os programas executáveis criados pelo usuário. O processo `init` é um processo de usuário, que através das primitivas UNIX inicia a árvore de processos dos usuários.

2.2 Estrutura do TRIX

O TRIX mantém esta mesma estrutura em quatro camadas, com algumas alterações. Passa-se a chamar de *kernel* (ou micro-*kernel*) somente a camada mais baixa do sistema, responsável pelo escalonamento e troca de mensagens. Na versão atual o código do micro-*kernel* do TRIX tem menos de 2500 bytes e sua área de dados tem em torno de 9000 bytes, incluindo a tabela de processos e *buffers* de mensagens. Assim, fica no *kernel* somente o código responsável pela troca de mensagens, já que o escalonamento é feito pelo *hardware*. Todas as *tasks* são compiladas, ligadas, carregadas e disparadas separadamente, não compartilhando nenhum dado ou código.

A forma mais comum de se encontrar transputers é em placas conectadas ao barramento de um computador completo, chamado de **hospedeiro**. A figura 2 exemplifica uma arquitetura típica do TRIX. O hospedeiro, que está conectado a todos os dispositivos de entrada e saída raramente é um transputer. Atualmente é um processador 80x86, rodando sob MS-DOS um programa emulador

de micro-*kernel* e de *tasks*. Dependendo da sofisticação deste sistema operacional, pode-se ter um ou mais processos concorrentes.

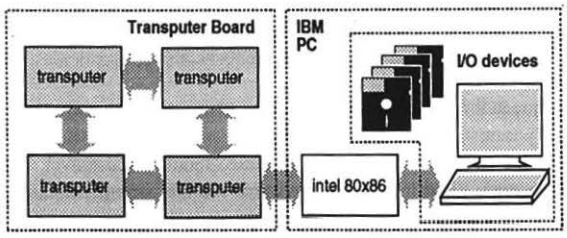


Figura 2: Arquitetura do equipamento usado no TRIX

Como nem todos os processadores possuem dispositivos acoplados, não é sempre necessário haver *tasks* para todos tipos de dispositivos em todos os processadores. Existem porém alguns processos que **precisam** existir localmente por executarem funções relativas aos processos e à memória locais do sistema. São eles: *system task*, *clock task* e *Memory Manager*. Além disso, todo processador também carrega uma *link task*, pois todos transputers somente se comunicam por *links*. Isto pode ser visto na figura 5 (a).

O *File System* somente é executado em um dos processadores da rede. Isto porque o código do *File System* do MINIX é feito de tal forma que seria muito complicado adaptá-lo para um sistema de arquivos distribuído — seria necessário reescrevê-lo. Esta centralização porém não causa uma redução no desempenho atualmente, pois somente um processador possui dispositivos de entrada e saída: o hospedeiro. No futuro, prevê-se um sistema somente com transputers, com dispositivos distribuídos pelos diversos processadores, o que certamente motivará o desenvolvimento de um sistema de arquivos distribuído.

3 O Micro-kernel

Embora o *hardware* do transputer seja capaz de escalonar processos e enviar mensagens, para se implementar um sistema operacional é necessário alguns mecanismos adicionais. Isto é mais marcante ainda quando se tem o compromisso de compatibilidade com UNIX. Embora fosse possível montar o sistema TRIX **sem** *kernel*, o *hardware* do transputer não valida uma série de detalhes como a direção, o tamanho, o endereço, a origem e o destino das mensagens, além de não contabilizar o tempo de CPU **por processo**.

Para tomar conta destes detalhes, foi usada a camada mais baixa do MINIX para implementar um micro-*kernel*. Este programa é executado após o *boot* do TRIX, e se encarrega de disparar todos os processos básicos do sistema. Após isso, o micro-*kernel* entra em um laço infinito, à espera de pedidos de envio ou recepção de mensagens dos processos em execução. Os pedidos podem ser `send` (envio de mensagem), `receive` (espera mensagem) ou `send_rec` (envia e espera resposta). *Tasks* MM e FS são disparados com prioridade transputer URGENT e processos de usuário como NOT_URGENT. Enquanto aqueles podem fazer qualquer tipo de pedido, estes somente podem solicitar `send_rec` ao *kernel*.

3.1 Mensagens

Cada processo disparado no sistema recebe um **canal** para se comunicar com o *kernel*. É através deste canal que o processo envia uma estrutura `sendrec_hdr` (figura 3), que contém os dados relativos ao pedido de envio/recepção de mensagem. O *kernel* ao encontrar duas solicitações que combinem entre si, efetua a cópia da mensagem e dispara os dois processos novamente. O bloqueio e o disparo de processos é feito usando-se as instruções apropriadas do transputer, deixando-se o escalonamento por conta do *hardware*.

```
typedef struct {
  long src_dst;          /* to/from who send the message */
  long function;         /* send, receive or both */
  long size;             /* total size of the message */
  message *m_ptr;        /* pointer to the message */
} sendrec_hdr;
```

Figura 3: Cabeçalho para mensagens

Duas otimizações foram inseridas no mecanismo de mensagens em relação ao MINIX. O tamanho das mensagens passou a ser variável e o `send` não mais bloqueia o processo. A primeira mudança foi inserida criando-se as funções `vsend`, `vreceive` e `vsendrec`, que implementam a troca de mensagens de tamanho variável. O *kernel* passou a levar em conta que a estrutura da mensagem foi alterada (ver figura 4) para conter o tamanho da mensagem. Todos os programas do MINIX à medida que são portados são alterados para usar estas novas funções. Por segurança na integridade dos dados do receptor, ao copiar uma mensagem o *kernel* usa o **menor** tamanho entre os dois especificados. Como é bem determinado o protocolo de comunicação entre os processos, em geral os tamanhos coincidem, não havendo perda de informação.

```
typedef struct {
  long m_source;                    /* who sent the message */
  long m_dest;                      /* to whom send it */
  long m_size;                      /* total size of message */
  long m_body[MESSAGE_BODY_WORDS]; /* message data */
} message;
```

Figura 4: Estrutura de uma mensagem

Ao inserir o paradigma de distribuição, fica bastante complicado manter o mecanismo de *rendez-vous* na troca de mensagens. Se um processo que emitir um `send` a um processo remoto ficar bloqueado, o *kernel* remoto teria que avisar o *kernel* local para que este desbloqueie o processo local após a entrega da mensagem. Além disso o *kernel* ao receber uma mensagem remota para um processo que não estivesse esperando teria que descartá-la ou armazená-la em algum local temporário. Por isso, foi montado um *buffer* para armazenar as mensagens emitidas remotamente que ainda não puderam ser copiadas ao destinatário (por este ainda não ter emitido um `receive`). Este conceito foi estendido às mensagens locais simplesmente porque o código

do *kernel* ficaria mais complicado se houvesse necessidade de manusear dois tipos de mensagens (remotas e locais).

3.2 Identificação global de processos remotos

Um processo se comunica com outro através do número de sua entrada na tabela de processos (como no MINIX), que funciona como uma porta. Os servidores atendem às portas 0 e 1 (MM e FS respectivamente) e as tasks atendem às portas negativas, cada uma associada a um tipo de dispositivo. No TRIX, o *kernel* troca os campos `m_source` e `m_dest` da mensagem para conter os endereços globais dos processos envolvidos. Caso `m_dest` seja referente a um processo remoto, o *kernel* ao receber a mensagem entrega-a para a *link task*. A *link task* (como será descrito na seção 4.2) se encarregará de levar a mensagem até o processador destinatário.

Para identificar processos remotos, foi incluído no identificador de processo (porta) a informações sobre sua localização. Além disso, é mantida pelo *kernel* uma tabela de processos remotos, convertendo um endereço local em global. Assim um processo que está se comunicando com um processo remoto não necessariamente precisa saber a localização deste. Como processos de usuário somente se comunicam com os servidores e os servidores somente com algumas *tasks*, esta tabela nunca é muito grande. Embora os servidores possam se comunicar com todos os processos de usuário, isso só ocorre para responder a um pedido. Como o *kernel* atualiza o campo `m_source` da mensagem com o endereço global do processo de origem, os servidores não precisam ter informação sobre o endereço local de todos os processos de usuário.

3.3 Tempo de CPU por processo

A última característica do micro-*kernel* a apresentar é a contabilização do tempo de CPU de cada processo. Para isso o *kernel* mede o tempo entre um pedido de troca de mensagem e outro. Quando o *kernel* recebe um pedido de envio ou recepção de mensagem, se o processo que o efetuou for do tipo `URGENT` (Não perde a CPU), todo o tempo decorrido desde o último pedido foi gasto por ele. Senão, supõe-se que o tempo tenha sido gasto pelos processos de usuário que estão sendo escalonados em *round-robin* pelo *hardware*. O *kernel* divide então este tempo entre todos os processos de usuário (`NOT_URGENT`) que estiverem no estado `READY`. O tempo pode não ter sido gasto irmanamente por todos os processos, mas na cômputo geral o erro deve ser pequeno.

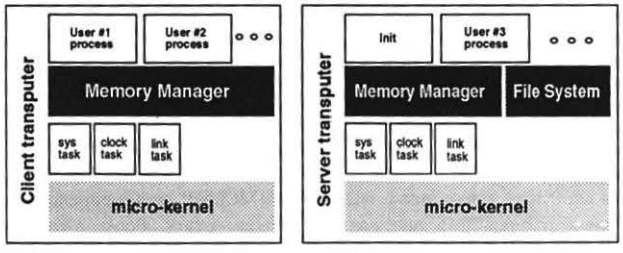
4 Processos do sistema

Ao ser projetado o TRIX decidiu-se que a carga dos programas do sistema operacional (servidores e *tasks*) seria feito de uma só vez, ao iniciar o sistema. Assim, cada processador recebe um conjunto de processos durante a carga inicial e os processos de usuário passam a ser distribuídos dinamicamente. Para distribuir os processos do sistema, é útil classificar os processadores como:

- **Controlador**, que possui dispositivos de entrada e saída;
- **Servidor**, que é o que vai executar o *File System*;
- **Cliente**, que não executa *tasks* de entrada e saída nem o *File System*.

Em todos os processadores executa-se a *system task*, a *clock task*, a *link task* e o *Memory Manager*. Embora não seja um *driver* de dispositivo, a *system task* é assim chamada porque controla os processos locais do sistema, fornecendo mecanismos para o *Memory Manager*. Executa-se a

clock task para controlar alarmes e o *shadowing*, como será visto mais adiante. Pelo menos mais uma *task* — para comunicação com os outros processadores — é necessária. Como transputers se comunicam por *links*, foi implementada a *link task*. Por fim, o *Memory Manager* é executado para controlar a criação e o término de processos de usuário. Este conjunto de processos é o que executa em um processador do tipo cliente, podendo ser observado na figura 5 (a).



(a) (b)

Figura 5: Processos dos transputers (a) cliente e (b) servidor

Nos processadores servidores (os mais complexos), executa-se todos os processos de um cliente, acrescidos do *File System*(figura 5 (a)). Na verdade, esta taxonomia aqui descrita foi criada por causa do FS centralizado. Com a implementação de um FS distribuído, isso não seria necessário.

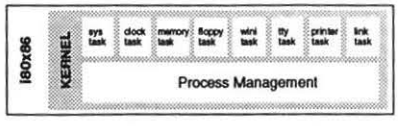


Figura 6: Processos do processador hospedeiro

Os controladores executam somente os *drivers* necessários. Se um dado processador possui disco rígido, executa a *winchester task*, se possui impressora, a *printer task* e assim por diante. Um processador do tipo controlador pode ser cliente ou servidor também, embora esta situação não seja usada, pois o único controlador é o hospedeiro. A figura 6 ilustra os processos executados no hospedeiro atual do TRIX.

4.1 System Task

A *system task* fornece os mecanismos para o *Memory Manager* criar e terminar processos, emitir sinais e fazer cópias de memória. Ela mantém a semântica MINIX, mas implementa alguns novos mecanismos que são necessários devido à arquitetura do transputer e do TRIX. À primeira vista, somente os processadores que executam processos de usuário necessitariam de uma *system task*. Ela é, porém, executada em **todos** os processadores do sistema, pois faz cópias de memória de um processo a outro, repassando para a *link task* as cópias remotas. Existe uma versão simplificada da *system task* para executar em processadores servidores ou controladores, onde não haja processos de usuário.

Como o transputer faz o escalonamento de processos por *hardware*, a *system task* recebeu funções adicionais para interromper a execução de um processo sem que este esteja bloqueado, semelhante ao que é feito no DISTRIX[SMI89]. Isto é implementado varrendo a lista de processos ativos do sistema e extraindo o processo em questão, para que possa ser trocado o *shadow* do processo ou para que possa receber um sinal.

Para implementar a primitiva `fork` do UNIX em um processador **sem** relocação de memória como o transputer /citearqtransputer existem duas alternativas: inserir no programa (através do compilador) indireções em todos os acessos aos dados do programa, ou fazer *shadowing*[TAN87], que consiste em deixar somente uma das duas cópias do processo após o `fork` executando por vez na posição original de memória. Após um certo tempo, bloqueia-se um do processos, copia-se ele para uma área chamada *shadow*, copiando-se simultaneamente o processo que estava na *shadow* para a posição original.

A primeira alternativa insere uma perda de 30 %[MCC90] no desempenho do sistema, e a segunda, de algo próximo a 70 %, pelo que se observou no TRIX. Porém a primeira alternativa atrasa **todos** os processos do sistema durante **todo** o tempo. Já a segunda, atrasa somente os processos que executaram `fork` e ainda não executaram `exec` em um novo programa. Segundo comentários de diversos autores no grupo *comp.os.minix* da rede *Usenet*, mais de 90 % das vezes que um processo executa `fork`, imediatamente este executa um `exec`, executando um novo programa, podendo-se parar o *shadowing*. Por isso se optou pela segunda alternativa, assim como no MINIX para processadores Motorola sem unidade de gerência de memória.

4.2 Link Task

A link task implementa a comunicação via o *link* do transputer. Foi feita seguindo o padrão das *tasks* do MINIX, que recebem mensagens, processam seu conteúdo e enviam uma resposta de volta. Como o *kernel* entrega à *link task* todas as mensagens remotas, esta testa o campo `m_dest` para ver se a mensagem deve ser enviada a outro processador ou se a mensagem é para a própria *task*. A figura 7 mostra o trecho de código da *link task* que faz isso.

A *link task* possui as funções enviar e receber mensagens e cópias de memória através do *link*. Como foi descrito na seção 3, o *kernel* ao perceber que uma mensagem tem como destino um processo remoto, entrega esta à *link task*, que através de um *link* a envia para outro processador, onde é recebida por outra *link task*. O mesmo ocorre com a *system task* com relação à cópias de memória: repassa à *link task* os pedidos remotos.

A *link task* tem cinco fluxos de execução independentes (paralelos), sendo um para atender às mensagens do *kernel* e da *system task* e outros quatro para atender aos *links*. O procedimento do *link* espera receber uma mensagem ou uma cópia. Quando alguma chega (como foi descrito acima), se esta for uma mensagem local, é entregue ao *kernel*. Se for uma cópia para um processo local, os dados são recebidos diretamente no espaço de endereçamento do processo destinatário. Se for uma cópia ou uma mensagem para um processo não local, é passado adiante por outro *link*.

Existe uma função que mapeia todos os processadores do sistema em um *link*, que é chamada para se decidir qual *link* usar para enviar uma mensagem ou uma cópia remota. Atualmente, como a configuração da ligação dos transputers é estática, esta função é avaliada através de uma tabela. Mas o sistema foi montado de tal maneira que se possa implementar um roteador mais inteligente, que por razões de falha ou desempenho, seja capaz de calcular dinamicamente as rotas.

```
void main() {
/* Main program of link task. It receives a message and
 * dispatch the procedure to treat it.
 */
  int opcode, res;

  init_link();       /* initialize link tables & globals */
  while (TRUE) {
    receive(ANY, &mc);  /* waits for a message */
    /* Test if message is in fact to the task, or to the link.
     * thistask contains the global address of this link task.
     */
    if (mc.m_dest != thistask)
      opcode = LINK_SEND;
    else
      opcode = mc.m_type;

    switch (opcode) {
      case LINK_SEND: res = do_linksend(&mc);  break;
      case SYS_COPY: res = do_linkcopy(&mc);  break;
      default: res = EINVAL;  break;
    }
  }
}
```

Figura 7: Programa principal da *link task*

4.3 Memory Manager

O *Memory Manager* implementa as primitivas UNIX relacionadas com processos e memória. Fazendo uso de chamadas baixo nível à *system task*, ele controla toda a alocação de memória de cada processo, cria e destrói processos, envia sinais, etc. Todas as funções originais do MINIX são mantidas, quando se referem à memória e aos procesos locais. As funções de criação e término de processos foram reescritas, incluindo um controle **distribuído** da árvore de processos.

Para implementar uma árvore de processos como a do UNIX de forma distribuída, cada entrada na tabela de processos do *Memory Manager* passou a contar com os seguintes campos adicionais:

- `mp_child`, endereço global apontando para o último processo filho criado por este;
- `mp_brother`, endereço global apontando para o processo filho do mesmo pai, que foi disparado antes deste;
- `mp_parent`, já existia no *Memory Manager* original para apontar para o processo pai, passa a conter um endereço global de processo.

A figura 8 mostra um exemplo de árvore de processos do TRIX, que graficamente explica o uso dos campos descritos acima. Nela, há dois processadores, um com três e outro com quatro

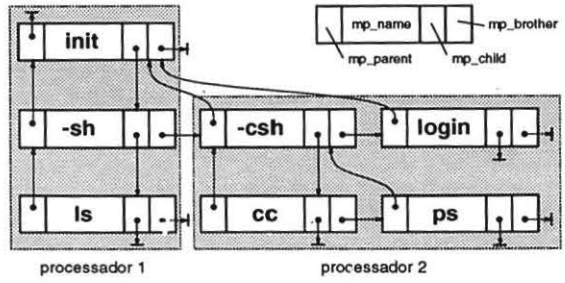


Figura 8: Um exemplo de árvore de processos

processos. O processo `init` possui três filhos: `-sh`, `-csh` e `login`. O `-sh` por sua vez criou o filho `ls`, o `-csh` criou `cc` e `ps` e o `login` não possui filhos. O controle desta árvore é feito através de mensagens, entre os *Memory Managers* envolvidos, enviadas na criação e término de cada processo. São elas:

- `RFORK`: pede ao *Memory Manager* remoto para executar um processo;
- `RYFORK`: processo executado remotamente (`RFORK` obteve sucesso);
- `RNFORK`: processo não executado remotamente (`RFORK` falhou);
- `ORPHAN`: processo pai terminou, novo pai passa a ser o `init`;
- `REXIT`: filho terminou, envia codigo de retorno ao pai;
- `RWAIT`: pai executou `wait`, para esperar filhos terminarem;
- `INHERIT`: algum processo com filhos morreu, `init` deve adotá-los;
- `FRESLT`: pai recebeu código de `exit`, pode liberar entrada na tabela.

O *Memory Manager* implementa a distribuição dos processos de usuários pelo sistema através de um balanceamento de carga. Este balanceamento é implementado em um módulo independente do *Memory Manager*, podendo ser alterado para experimentar diferentes modelos. O modelo atual é inspirado (simplificado) no modelo publicado em [LEI91], já estudado e implementado no CPGCC da UFRGS na dissertação [BEL92]. Tal mecanismo é mecanismo é totalmente distribuído, onde cada processador mantém o endereço do processador menos ocupado mais próximo.

Resumidamente, o mecanismo é o seguinte: cada *Memory Manager* possui uma variável que diz **qual** o processador que está com menos carga. Quando for necessário disparar um novo processo, o *Memory Manager* local envia um RFORK para tal procesador. A cada intervalo de tempo a *clock task* envia para o *Memory Manager* uma taxa de ocupação média da CPU. Se esta taxa for menor que a taxa armazenada, ela substitui a antiga e o *Memory Manager* avisa todos os processadores vizinhos desta troca. Todos *Memory Managers* se comportam da mesma forma, acontecendo uma espécie de *broadcast* por difusão. Qualquer mensagem com uma taxa de ocupação maior que a atualmente conhecida como menor será descartada, parando de propagar.

Há também uma versão reduzida do *Memory Manager*, que pode ser opcionalmente executada em processadores servidores e controladores. Ele não cria processos de usuário no processador local, de forma a garantir um melhor desempenho, tanto para o FS e as *tasks* quanto para os processos de usuário.

5 Conclusão

Este artigo mostrou como foi feito o sistema operacional TRIX, como parte dos trabalhos do grupo de processamento paralelo do CPGCC da UFRGS. Tendo sido completada sua implementação, estudam-se agora as possibilidades de expansão do sistema, como por exemplo um sistema de arquivos distribuído.

O sistema apresentado é compatível com o UNIX versão 7 e, comparado com as soluções mais freqüentemente encontradas para o controle de transputers, ele tem a vantagem de ser executado pelos próprios transputers. Isto destaca tais processadores como sendo os mais significativos no sistema, posição que deveriam ocupar por seu alto desempenho. Em outros sistemas, esta posição era ocupada pelo hospedeiro.

O TRIX pode ser executado em um grupo de transputers com qualquer topologia, pois roteamento de mensagens entre processos é feito fora do *kernel* por um *driver* especializado, que pode ser reescrito para qualquer dispositivo. Além disso, o *kernel* provê aos processos do sistema uma transparência de localidade ao converter endereços locais em globais.

Efetua-se um balanccamento de carga simples e rápido, que não possui nenhum controle centralizado — a informação sobre a árvore de processos é totalmente distribuída. Seus ponto mais fraco está no mecanismo de *shadowing* que torna mais complexa a implementação do núcleo e diminui o desempenho quando está em ação. Este problema será solucionado por *hardware*, ao dotar-se os transputers de um dispositivo de gerência de memória.

Referências

[BEL92] BELMONTE FILHO, V. R., **Gerência de Processos em Sistemas Distribuídos Tolerantes a Falhas**. Dissertação de Mestrado, Porto Alegre, CPGCC/UFRGS. 1992.

[DIS88] DISTRIBUTED SOFTWARE, Ltd. **Helios-PC, Software development system for PC-hosted transputer cards**. Somerset, UK, 1988.

[INM90] INMOS, Corp. **Transputer Data Book**. 2. edição, 1990.

[LEI91] LEISS, E. L. & REDDY, H. N., **Distributed Load Balancing algorithms: design and performance analysis**. Research Computer Laboratory, University of Houston, 1991. (Research Report)

[MCC90] MC CULLAGH, P. & SMIT G. de V. Implementing UNIX in the INMOS Transputer. **South African Computer Journal**, n. 2, 1990.

[MEI91] MEIKO, Inc. **MeikOS Operating System**. manufacturer folder.

[PAZ91] PAZZINI, M., **Especificação do TRIX: Um Sistema Operacional Multiprocessado para Transputers**. Porto Alegre, CPGCC/UFRGS. Trabalho Individual, 1991.

[POU91] POUNTAIN, D. The Chorus of Approval. **BYTE**, v. 16, n. 4, p. 265-275, abril de 1991.

[SHE87] SHEPHERD, D. **Transputer Instruction Set: A Compiler Writer's Guide**. Prentice-Hall. Hertfordshire, UK, 1988.

[SMI89] SMIT, G. de V., HOFFMAN, P. K. & MC CULLAGH, P. J., DISTRIX, **A Multiprocessor UNIX Workbench**. Department of Computer Science, University of Cape Town, 1989. (Research Report)

[TAN87] TANENBAUM, A. S. **Operating Systems: Design and Implementation**. Prentice-Hall. Englewood-Cliffs, 1987.