Implementação das Bibliotecas Multithread do Sistema Operacional Mulplix

Márcio de Oliveira Barros
cuco@nce.ufrj.br

Júlio Salek Aude
salek@nce.ufrj.br

Paulo Alberto S. dos Santos
paulo@nce.ufrj.br

Núcleo de Computação Eletrônica
Universidade Federal do Rio de Janeiro
Cidade Universitária, RJ, Brasil
CEP: 20001-970 Caixa Postal: 2324

Resumo

Mulplix é um sistema operacional multithread, que está sendo desenvolvido para o computador paralelo **Multiplus**, a partir de uma extensão do sistema **Plurix**, anteriormente desenvolvido no **NCE/UFRJ**. Para facilitar o reuso dos programas originais do **Plurix** e o desenvolvimento de aplicações paralelas, foi criado, para o sistema **Mulplix**, uma biblioteca multithread de rotinas para a linguagem de programação **C**. Este artigo apresenta a implementação desta biblioteca, focalizando os problemas encontrados no processo de transformação de uma biblioteca originalmente desenvolvida para um sistema operacional tradicional, em uma biblioteca cujas rotinas possam ser executadas paralelamente.

Palavras Chave: paralelismo, multithread, bibliotecas de programação

Abstract

Mulplix is a multithread operating system, under development for the **Multiplus** parallel computer. **Mulplix** is based on an extension of the **Plurix** operating system, previously developed at **NCE/UFRJ**. In order to simplify the reuse of Plurix programs and the development of parallel applications, it was created within **Mulplix** a multithread library of functions for the **C** programming language. This paper presents the library implementation, discussing the problems that had to be faced on the process for transforming a library that was developed for a traditional operating system into a library whose functions can work in a parallel environment.

Keywords: parallel processing, multithread, programming libraries

1. Introdução

Com o advento de sistemas operacionais multithread, que fatoram seus processos em *threads*, linhas de controle com troca de contexto leve, um mesmo programa pode ser executado de forma mais eficiente em diversos processadores simultaneamente. Para que um programa possa usufruir destes benefícios, seu código deve ser alterado. Estas alterações visam principalmente a fatoração do programa em *threads* e a sincronização da execução destas *threads*.

Entretanto, deve ser possível transportar os programas desenvolvidos nos sistemas operacionais tradicionais para os novos sistemas multithread. Esta migração deve ser facilitada, de forma que o volume de código escrito anteriormente possa ser reutilizado com um mínimo de esforço. Com o objetivo de facilitar esta migração, especialmente no que concerne a programas escritos na linguagem **C**, foi desenvolvida para o sistema operacional **Mulplix** uma biblioteca de rotinas multithread.

Este artigo apresenta a implementação das bibliotecas multithread do sistema operacional **Mulplix**, focalizando os problemas enfrentados na transformação de uma biblioteca originalmente desenvolvida para um sistema operacional tradicional, em uma biblioteca cujas rotinas possam ser executadas paralelamente.

2. O Sistema Operacional Mulplix

O sistema operacional **Mulplix** [AZEV93] está sendo desenvolvido para o computador paralelo **Multiplus** [AUDE95], um multiprocessador de alto desempenho com arquitetura modular de memória global fisicamente distribuída. A arquitetura do **Multiplus** suporta até 1024 nós de processamento, organizados em 128 clusters, contendo cada um deles um máximo de 8 nós de processamento. Cada nó de processamento contém um *chipset* **Sparc** e parte da memória global. Os clusters são interligados por uma rede de interconexão multiestágio e associado a cada cluster há um processador de entrada e saída.

O sistema **Mulplix** está sendo desenvolvido como uma extensão do **Plurix**, um sistema Unix-like, desenvolvido integralmente no **NCE/UFRJ**. As principais extensões ao **Plurix** incorporadas no sistema **Mulplix** são: a introdução do conceito de *threads*, sub-processos com troca de contexto leve; a disponibilização de primitivas de sincronização para o usuário; a adoção de políticas de escalonamento de *threads* e processos e de gerência e alocação de memória adequadas à arquitetura do **Multiplus**; a disponibilização de bibliotecas adequadas ao ambiente multithreaded do **Mulplix** e a provisão de suporte à gerência de arquivos distribuídos.

3. A Biblioteca C

A biblioteca multithread do **Mulplix** foi desenvolvida a partir da biblioteca original do sistema **Plurix**. Esta biblioteca pode ser dividida nos seguintes conjuntos de rotinas:

- **Manipulação de Inteiros Longos**: contém rotinas para manipulação matemática de inteiros longos, como divisão, multiplicação e resto de divisão. Estas rotinas são chamadas pelo compilador **C**, quando o código gerado manipula números inteiros longos;

- **Memória**: contém rotinas para manipulação de regiões de memória. Também estão neste grupo as rotinas de alocação dinâmica de memória;
- **Usuários**: contém rotinas para manipulação das estruturas que representam os usuários do sistema e os grupos de usuários;
- **Identificadores**: contém rotinas para manipulação de identificadores, que são cadeias de caracteres de tamanho pré-determinado. As rotinas deste grupo permitem uma manipulação mais rápida de identificadores, sendo utilizadas por diversos programas do sistema, como o compilador, o linqueditor e o montador assembly;
- **Arquivos**: contém as rotinas de manipulação de arquivos;
- **Strings**: contém as rotinas de manipulação de cadeias de caracteres;
- **Sistema**: contém as rotinas que fazem chamadas diretas às primitivas do sistema operacional. Estas rotinas estão relacionadas com tratamento direto de arquivos, diretórios e volumes, tratamento de usuários e grupos de usuários a nível de sistema, processamento de sinais, entre outros;
- **Rotinas genéricas**: contém rotinas de medição de tempo, de ordenação, de geração de números randômicos, de manipulação de terminais, de busca em vetores, de percurso de diretórios, entre outras rotinas genéricas.

A biblioteca possui um módulo que define o símbolo de entrada dos programas C. O linqueditor utiliza este símbolo para marcar o ponto de início da execução dos programas. O código que segue este símbolo, denominado **prólogo**, inicializa o ambiente de execução do programa, chama a rotina ***main*** e, ao fim da execução desta rotina, termina o programa. Este módulo é fundamental para a implementação das bibliotecas multithread, como será visto adiante.

4. Problemas com a Biblioteca C

Quando um programa **C** que utiliza a biblioteca de rotinas é compilado, seu código objeto declara as chamadas a rotinas da biblioteca através de símbolos, inicialmente indefinidos. Estes símbolos serão resolvidos pelo linqueditor, que acoplará as rotinas da biblioteca necessárias ao código objeto do programa, gerando o executável final. Desta forma, cada executável conterá uma cópia do código e dos dados manipulados pelas rotinas da biblioteca.

No **Plurix**, cada programa em execução define um processo, que é uma linha de controle cuja execução não pode ser dividida. O código do processo é executado de forma estritamente seqüencial em um único processador.

No sistema operacional **Mulplix**, diversas *threads* do mesmo processo podem ser executadas paralelamente em processadores distintos. Como algumas destas *threads* podem conter chamadas a uma mesma rotina da biblioteca, diversos processadores podem executar esta rotina simultaneamente. Esta execução paralela gera problemas quando as rotinas possuem áreas de exclusão mútua, em que apenas uma execução é permitida em cada instante.

Como o **Plurix** não possui o conceito de *threads*, as rotinas de sua biblioteca não estão protegidas contra a execução paralela de suas áreas de exclusão mútua dentro do mesmo processo. Estes acessos devem ser serializados de forma a manter a consistência

do resultado da execução destas rotinas. Existem duas abordagens [JONE91] ortogonais para esta sincronização:

- **Sincronização externa**: nesta abordagem, a sincronização das chamadas paralelas a rotinas é responsabilidade do programador. As rotinas não possuem nenhum mecanismo de proteção, restando ao programador a tarefa de utilizar semáforos de exclusão mútua onde a sincronização é necessária. A principal vantagem desta abordagem é sua flexibilidade, que permite que o programador defina exatamente onde a sincronização é necessária, dispensando sincronizações nas regiões do programa onde não haverão chamadas paralelas à mesma rotina. Sua principal desvantagem é delegar a responsabilidade pela sincronização ao programador;
- **Sincronização interna**: nesta abordagem, a sincronização é feita no código de cada rotina. Sua principal vantagem é liberar o programador da tarefa de controlar a sincronização, que já foi definida a nível da biblioteca. Sua principal desvantagem reside no fato de que a sincronização é sempre utilizada, mesmo nos momentos em que o programador sabe que não ocorrerão chamadas paralelas.

Com o objetivo de facilitar o transporte dos programas de sistema do **Plurix** e de programas desenvolvidos para outros sistemas operacionais compatíveis com o Unix para o **Mulplix**, optamos pela sincronização interna das rotinas da biblioteca. É importante observar que, para que esta migração seja facilitada, os protótipos das rotinas devem sofrer o mínimo de alterações.

Os problemas encontrados na sincronização interna das rotinas da biblioteca podem ser classificados em dois grupos: rotinas com interface não reentrante e rotinas com implementação não reentrante.

4.1. Rotinas com Interface não Reentrante

Rotinas com interface não reentrante são rotinas que utilizam variáveis globais para compor seus valores de retorno. Quando uma rotina com interface não reentrante é executada paralelamente em mais de um processador, os valores das variáveis globais utilizadas por ela são sobrepostos, gerando inconsistência no resultado da rotina.

Um caso especial de rotinas com interface não reentrante são as rotinas que utilizam variáveis locais estáticas para armazenar seus valores de retorno. Uma variável local estática é uma variável local que mantém seu valor entre duas chamadas da função. Exceto por seu contexto estar limitado a função que a declara, uma variável local estática pode ser vista como uma variável global.

Uma solução para o problema das rotinas com interface não reentrante é a alteração dos protótipos destas rotinas, criando-se um novo parâmetro, que substituirá a variável global utilizada. Este parâmetro é utilizado para o cálculo e retorno do resultado da rotina. Execuções paralelas atuarão sobre regiões de memória distintas, não interferindo entre si.

Um exemplo de rotina com interface não reentrante é a rotina ***localtime***. Esta rotina originalmente utilizava uma variável global, onde depositava seu resultado, retornando um ponteiro para esta variável global. O antigo protótipo da rotina é apresentado a seguir:

TM *localtime (time_t *timer)

Este protótipo foi alterado, de forma que a rotina receba um segundo parâmetro, onde depositará seu resultado. A nova implementação retorna um ponteiro para este parâmetro.

TM *localtime (time_t *timer, TM *tp)

4.2. Rotinas com Implementação não Reentrante

Rotinas com implementação não reentrante são rotinas que utilizam variáveis globais ou variáveis locais estáticas para armazenar seu estado entre chamadas ou dentro de uma mesma chamada. Chamadas paralelas a estas rotinas geram inconsistência nos valores destas variáveis, afetando o resultado das rotinas.

O acesso a variáveis utilizadas para cálculos intermediários ou para armazenar estado entre chamadas define áreas de exclusão mútua dentro do código das rotinas. Quando uma *thread* entra na área de exclusão mútua de uma rotina, nenhuma outra *thread* deve entrar nesta área. Uma segunda *thread* deve aguardar que a primeira termine a execução da área crítica, antes de entrar nesta área de código.

Desta forma, deve haver uma serialização das chamadas a estas rotinas, de forma a controlar o acesso às áreas de exclusão mútua. Esta serialização é provida por um semáforo de exclusão mútua. Este semáforo é capturado pela primeira *thread* que executa a área crítica. Cada nova *thread* que chega no início desta área fica bloqueada até que o semáforo seja liberado. Esta liberação ocorre quando a *thread* que capturou o semáforo sai da área crítica, liberando a execução deste código para outra *thread*.

Um exemplo de rotina com implementação não reentrante é o gerador de números aleatórios, representado pela rotina **rand**. Esta rotina trabalha sobre o valor de uma semente geradora de números aleatórios, que é armazenada em uma variável global. Na implementação multithread desta rotina, os acessos a esta variável são serializados com um semáforo de exclusão mútua.

A utilização de semáforos nas rotinas com implementação não reentrante gera um problema: os semáforos são recursos limitados do sistema, devendo ser requisitados ao sistema antes de ser capturados pelas *threads*, e liberados para o sistema, quando não forem mais necessários. Assim, estes semáforos devem ser inicializados antes da primeira chamada a rotina que os utiliza e devem ser destruídos apenas quando não houver nenhuma outra chamada a esta rotina.

Para criarmos e destruirmos os semáforos utilizados pela biblioteca sem que o programador tenha que chamar, explicitamente, uma rotina para a inicialização de semáforos e uma rotina para a destruição destes semáforos, utilizamos o prólogo de início do programa. Este prólogo foi alterado de forma a chamar uma rotina de inicialização de semáforos antes de chamar a função ***main***. Da mesma forma, o prólogo chama uma rotina de destruição dos semáforos, quando a função ***main*** termina sua execução.

4.2.1. A Variável *errno*

A variável global ***errno*** é utilizada por diversas rotinas da biblioteca e rotinas que acessam às primitivas do sistema operacional. Esta variável contém o código do último erro que tenha ocorrido na chamada de uma destas rotinas.

As rotinas que acessam a variável ***errno*** são casos especiais de rotinas com implementação não reentrante. O conteúdo desta variável é alterado por chamadas

paralelas destas rotinas, obscurecendo erros que tenham ocorrido ou induzindo rotinas a tratarem erros que não ocorreram.

A Figura 1 apresenta esta situação. As setas indicam a ordem dos acessos a variável ***errno***. Supondo que o arquivo **arq2** exista e possa ser aberto para leitura, se o arquivo **arq1** não existir ou não puder ser lido, a segunda thread achará que o arquivo **arq2** não foi aberto.

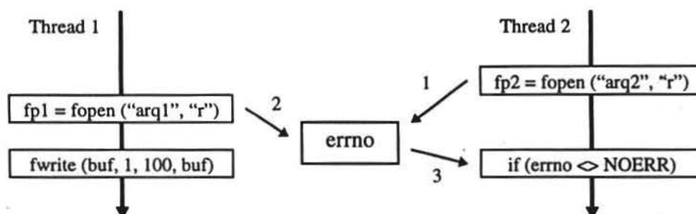

Figura 1 - Utilização da variável global ***errno*** por duas *threads*

A variável ***errno*** não pode ser mantida consistente por sincronização interna, uma vez que diversas rotinas distintas a manipulam separadamente, inclusive a *thread* em si.

Uma das soluções para a proteção da variável ***errno*** é sua replicação em uma área privativa de dados de cada *thread*. Assim, cada *thread* teria uma instância de ***errno***, indicando os erros ocorridos em chamadas de rotinas feitas pela *thread*. Entretanto, o sistema operacional **Mulplix** não provê mecanismos para memória estática privativa de *thread* e a utilização de memória dinâmica para a variável ***errno*** criaria um *overhead* que pode ser evitado.

Para solucionar o problema da ***errno***, a estrutura que representa uma *thread* dentro do núcleo do sistema operacional foi alterada, criando-se um novo campo que conterá os códigos de erro de execução. Duas novas primitivas foram adicionadas ao sistema, permitindo o acesso a este campo. A primitiva **set_*errno*** altera o valor deste campo, enquanto a primitiva **get_*errno*** retorna o conteúdo deste campo. A variável ***errno*** foi mantida a título de compatibilidade, sem nenhuma sincronização sobre o acesso a seu conteúdo.

4.2.2.Alocação Dinâmica de Memória

As rotinas de alocação dinâmica de memória são um segundo caso especial de rotinas com implementação não reentrante. As rotinas de alocação e liberação de blocos de memória alocados dinamicamente atuam sobre uma arena de alocação, delimitada por variáveis globais.

Os acessos a arena devem ser sincronizados de forma que apenas uma *thread* manipule a arena em um determinado instante. Para isto, um semáforo de exclusão mútua é utilizado pelas rotinas ***malloc*** e ***free***.

Entretanto, a sincronização do acesso a arena gera um *overhead* na alocação dinâmica de memória. Este *overhead* é indesejado e, em diversas situações, desnecessário. Para reduzir este *overhead*, a biblioteca multithread provê uma opção de alocação dinâmica rápida, sem proteção contra acessos paralelos.

As versões rápidas das rotinas ***malloc*** e ***free***, respectivamente ***fast_malloc*** e ***fast_free***, podem ser utilizadas quando o programador sabe que não ocorrerão chamadas paralelas a estas rotinas.

4.2.3. Acesso a Arquivos

Um terceiro caso especial de rotinas com implementação não reentrante são as rotinas de acesso a arquivos. Estas rotinas utilizam um conjunto pré-determinado de *buffers*, que são variáveis globais. Estas rotinas atuam também sobre a estrutura ***FILE***, que representa um arquivo. Um ponteiro para a estrutura ***FILE*** é recebido como parâmetro em cada uma destas rotinas.

Como diversas rotinas distintas podem estar atuando paralelamente sobre o mesmo arquivo, os acessos a estrutura ***FILE*** devem ser sincronizados por um semáforo de exclusão mútua. O mesmo deve ser dito sobre o acesso aos ***buffers*** globais.

Como cada arquivo deve ter uma sincronização própria, cada estrutura ***FILE*** deve conter um semáforo para sincronizar o acesso ao arquivo que ela representa. Este semáforo é utilizado em todas as rotinas de acesso a este arquivo.

O semáforo utilizado nas rotinas de acesso a arquivos também deve ser requisitado ao sistema antes de utilizado, e liberado para o sistema quando não for mais necessário. O problema do pedido e liberação destes semáforos foi resolvido com as rotinas de abertura e fechamento de arquivos. Como um arquivo somente pode ser utilizado depois de aberto, o semáforo da estrutura ***FILE*** que o representa é requisitado ao sistema na rotina de abertura do arquivo - ***fopen***. Analogamente, este semáforo é liberado para o sistema na rotina de fechamento do arquivo - ***fclose***.

O acesso paralelo de diversas *threads* a um mesmo arquivo pode gerar inconsistências no conteúdo deste arquivo. Suponhamos, por exemplo, que uma *thread* deseja escrever o seguinte texto, sem interrupção, no arquivo saída padrão:

printf ("Este texto não ");
printf ("pode ser dividido.");

Se uma segunda *thread* estiver escrevendo no arquivo de saída padrão, as duas execuções da rotina *printf* podem ser intercaladas por uma escrita na saída padrão pela segunda *thread*. A Figura 2 representa esta situação.

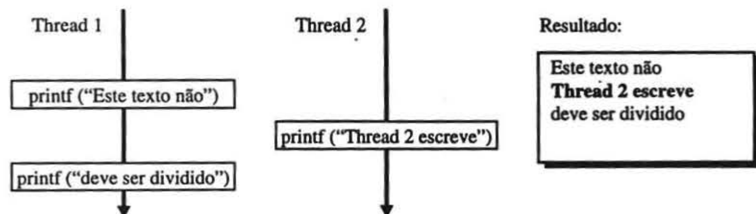

Figura 2 - Acesso a arquivo intercalado por duas threads

Para permitir que uma *thread* tenha acesso exclusivo a um arquivo, três rotinas foram acrescentadas a biblioteca do compilador C. A rotina ***flockfile*** recebe como parâmetro um ponteiro para uma estrutura **FILE** e o identificador de uma *thread*,

bloqueando o acesso a este arquivo para esta *thread*. Se alguma outra *thread* tiver acesso exclusivo ao arquivo desejado, a rotina aguarda até que este acesso seja liberado. O código anterior pode ser redefinido para:

```
flockfile (stdout, thr_id());
printf ("Este texto não ");
printf ("pode ser dividido.");
funlockfile (stdout);
```

Assim, a função ***flockfile*** faz com que o acesso ao arquivo de saída padrão seja exclusivo da *thread* atual (a função ***thr_id()*** retorna o identificador da *thread* atual). As duas chamadas a rotina *printf* não serão interrompidas por nenhuma outra rotina que escreva neste arquivo. A Figura 3 representa esta situação.

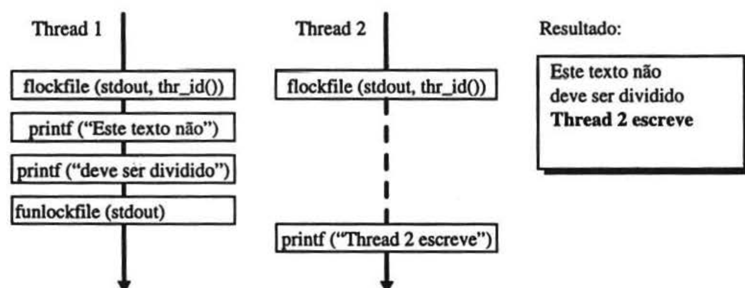

Figura 3 - Acessos a arquivos sincronizados por flockfile

A rotina ***funlockfile*** recebe como parâmetro um ponteiro para uma estrutura **FILE**, liberando o acesso a este arquivo para todas as *threads*. Somente a *thread* que anteriormente pediu acesso exclusivo ao arquivo pode liberar este acesso.

A rotina ***ftrylockkfile*** recebe os mesmos parâmetros que a rotina ***flockfile***, verificando se o arquivo tem acesso exclusivo para outra *thread*. Caso o arquivo não tenha acesso exclusivo para outra *thread*, captura o acesso exclusivo para a *thread* cujo identificador é passado como parâmetro.

Como nas rotinas de alocação dinâmica de memória, a sincronização gera um *overhead* nas rotinas de leitura e escrita de arquivos. Versões rápidas das rotinas ***putc*** e ***getc*** foram implementadas na forma de duas macros: ***fast_putc*** e ***fast_getc***. Estas macros provêem um método rápido de acesso a arquivos, sem sincronização.

A sincronização do acesso aos *buffers* globais é uma sincronização simples. Um único semáforo de exclusão mútua é utilizado para serializar o acesso a estes *buffers*.

5. Trabalhos Relacionados

Pthreads é o padrão **POSIX** para programação multithread. Este padrão consiste de um conjunto de rotinas, fornecidas por uma biblioteca que permite o desenvolvimento de aplicações multithread. Estas rotinas permitem a criação e o término de *threads*, a manipulação de semáforos de exclusão mútua e de semáforos condicionais e a definição

de memória privativa de *threads*. Outra característica da biblioteca é o provimento de rotinas para manipulação de arquivos, protegidas contra chamadas paralelas.

Grande parte da funcionalidade da biblioteca **Pthreads** é provida pelo próprio sistema operacional **Mulplix**, restando à biblioteca apenas a definição das chamadas as primitivas do sistema que implementam estas funções. As rotinas de acesso a arquivos foram protegidas contra chamadas paralelas a nível da biblioteca.

Embora o sistema **Mulplix** não tenha suporte para memória privativa de *thread*, esta funcionalidade pode ser reproduzida pela utilização de variáveis locais na função que representa a *thread*. Outra solução para este problema reside na utilização de blocos de memória privados alocados dinamicamente dentro de cada *thread*.

6. Considerações Finais

Além de transformar as rotinas originais do **Plurix** em rotinas multithread, a biblioteca do **Mulplix** implementa rotinas para acessar as novas primitivas do sistema operacional. Fundamentalmente, as novas primitivas estão relacionadas com programação paralela [AZEV93]. Um novo grupo de rotinas foi criado para agrupar estas rotinas, implementadas para prover:

- criação, destruição, travamento e liberação de semáforos de exclusão mútua;
- criação, destruição, sinalização e espera para semáforos de barreira;
- disparo e término da execução de *threads*.

Além dos semáforos utilizados nas rotinas com implementação não reentrante, a biblioteca utiliza alguns semáforos para sua sincronização interna. Como o semáforo é um recurso limitado do sistema, a implementação da biblioteca multithread levou em consideração o fator custo/benefício da proteção de algumas rotinas contra chamadas paralelas. Desta forma, rotinas pouco utilizadas (como ***atexit***) ou que não fazem sentido dentro de *threads* (como ***getopt***) foram deixadas sem sincronização, reduzindo o número de semáforos utilizados por processo.

Um aspecto ainda não abordado com relação a biblioteca multithread do **Mulplix** são as rotinas de processamento de sinais. O processamento de sinais em ambientes multithread é um problema complexo, que gera diversas questões, como a definição da *thread* que deverá tratar um determinado sinal. Uma possível solução para esta questão é fazer com que a rotina de tratamento de sinais dispare uma *thread* especificamente para tratar o sinal. Mecanismos de sinalização entre *threads* do mesmo processo também estão sendo analisados.

Para facilitar a portabilidade de programas desenvolvidos em outras plataformas compatíveis com o **UNIX** para o **Mulplix**, desenvolvemos um folheto que apresenta as modificações realizadas na biblioteca original do **Plurix**. Este folheto pode ser utilizado como uma "receita de bolo", indicando as modificações que um programador deve efetuar em seu código para adaptá-lo à nova biblioteca.

Diversos testes foram realizados com as rotinas da biblioteca multithread, verificando se estas rotinas estavam realmente protegidas contra chamadas paralelas. Em um primeiro momento, estes testes atuaram a nível de unidade, ou seja, cada rotina foi testada separadamente. Em um segundo momento, a biblioteca multithread foi testada em conjunto com a implementação do **MPVM**, uma biblioteca de rotinas para programação paralela, compatível com o padrão **PVM**, que utiliza memória

compartilhada ao invés de troca de mensagens. Diversos experimentos foram realizados unindo as duas implementações, com uso intensivo da biblioteca, especialmente das rotinas de alocação dinâmica e acesso a arquivo.

Bibliografia

[AUDE95] Aude, Júlio S. et al, "Implementation of the Multiplus/Mulplix Parallel Processing Environment", Anais do VII Simpósio Brasileiro de Arquitetura de Computadores - Processamento de Alto Desempenho, Agosto 1995, pp. 621-635

[AZEV93] Azevedo, Rafael P. "MULPLIX: Um Sistema Operacional tipo UNIX para Programação Paralela", Tese de Mestrado, COPPE/UFRJ, Departamento de Engenharia de Sistemas e Computação, Março de 1993

[JONE91] Jones, Michael B. "Bringing the C Libraries With Us into a Multi-Threaded Future", USENIX, Dallas, Winter 1991