Implementação do Padrão Pthreads sobre o Sistema Operacional Mulplix

Márcio de Oliveira Barros*
cuco@nce.ufrj.br

Júlio Salek Aude**
salek@nce.ufrj.br

* **NCE/UFRJ**

** **IM/UFRJ e NCE/UFRJ**

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

Resumo

POSIX Threads, também conhecido como Pthreads, é um modelo de programação paralela padrão, baseado em multithreading. Este artigo trata da implementação do modelo Pthreads sobre as primitivas de programação paralela Mulplix. Mulplix é o sistema operacional do multiprocessador de memória compartilhada e distribuída Multiplus. Compatibilidade e soluções adotadas na implementação são discutidas. Resultados experimentais comparam a performance da implementação proposta com a implementação do modelo Pthreads para o sistema Solaris.

Abstract

POSIX Threads, also known as Pthreads, *is a standard parallel programming model, based on multithreading. This paper addresses the implementation of the Pthreads model on top of the Mulplix parallel programming primitives. Mulplix is the operating system under development for the Multiplus distributed shared-memory multiprocessor. Compatibility issues and adopted solutions for the implementation of the Pthreads thread and synchronization models are discussed. Experimental results compare the performance of the proposed implementation with the Pthreads implementation for the Solaris system.*

Palavras-chave: threads, programação paralela, sincronização, sistemas operacionais

1. Introdução

POSIX Threads [IEE94], também conhecido por **Pthreads**, é um modelo de programação paralela, definido pelo grupo de trabalho 1003.4a da POSIX. O modelo Pthreads está, atualmente, em sua décima versão.

Denominamos por modelo de programação paralela um conjunto de rotinas que atuam como interface entre uma aplicação e os recursos paralelos de um sistema operacional. São exemplos de modelos de programação paralela o Posix Threads, Parallel Virtual Machine (PVM) [GEI94], Message Passing Interface (MPI) [MPI95], Solaris Light Weigth Processes (LWP) [SUN95], Solaris Threads [GRA95] e as primitivas de programação paralela do sistema operacional Mulplix [AZE93].

O modelo Pthreads é baseado em multithreading, ou seja, um processo em execução é dividido em processos menores, chamados threads. Cada thread é uma linha de controle que pode ser executada em paralelo em processadores distintos. Cada processo possui pelo menos uma thread, conhecida com thread principal. Esta thread é responsável pela criação das demais threads componentes do processo.

O sistema operacional Mulplix [AUD96] define um modelo nativo de programação paralela, também baseado em threads. Este modelo é bastante simples, com o intuito de prover maior eficiência. Atualmente, ele serve como base para a implementação de modelos de programação paralela de mais alto nível, como o MPVM [SAN97] e o Pthreads, sob a forma de bibliotecas de rotinas para o compilador C.

A seção 2 apresenta o sistema operacional Mulplix e seu modelo nativo de programação paralela. A seção 3 apresenta as rotinas de manipulação de threads do modelo Pthreads e discute sua implementação sobre o Mulplix. A seção 4 apresenta os mecanismos de sincronização do modelo Pthreads e discute sua implementação a partir das primitivas de sincronização do modelo Mulplix. A seção 5 apresenta resultados experimentais da implementação do modelo Pthreads. Considerações finais e perspectivas futuras são traçadas na seção 6.

2. O Sistema Operacional Mulplix

Mulplix [AUD96] [AZE93] é um sistema operacional UNIX-like com suporte a paralelismo de granularidade média. Mulplix tem como objetivo prover um ambiente eficiente para a execução de aplicações paralelas no Multiplus, um multiprocessador de alto desempenho com arquitetura modular de memória global fisicamente distribuída. A arquitetura do Multiplus suporta até 1024 processadores, organizados em 128 clusters, contendo cada um deles um máximo de 8 processadores. Em sua versão inicial, Mulplix resultará de extensões ao sistema operacional Plurix, um sistema UNIX-like com suporte a multiprocessamento em arquiteturas SMP [FAL89].

Uma grande extensão do sistema Mulplix em relação ao Plurix é a definição do conceito de threads. No Mulplix, uma aplicação paralela consiste de um processo e um conjunto de threads. Assim, quando um processador troca de thread em um mesmo processo, somente o contexto do processador precisa ser salvo. Informações a respeito da gerência de memória ou alocação de recursos são únicas para o processo como um todo e permanecem inalteradas quando o processador troca de contexto de thread. Em relação à sincronização, Mulplix define

primitivas para manipulação de semáforos de exclusão mútua e semáforos de ordem parcial a nível de usuário.

O modelo de programação paralela nativo do Mulplix está disponível a aplicações através de chamadas ao sistema [AZE93]. A chamada ao sistema *thr_spawn* permite a criação de grupos de threads. O número de threads que serão criadas, o nome da rotina a ser executada nestas threads e um argumento comum, que será passado para esta rotina, são os parâmetros da chamada ao sistema. Um parâmetro opcional, que permite a definição da unidade de processamento preferencial para a execução da thread será adicionado em futuras versões do sistema operacional.

Uma segunda versão da chamada ao sistema anterior, chamada *thr_spawns*, permite a criação de threads em modo síncrono. No disparo síncrono, a thread que dispara as novas threads suspenderá sua execução, aguardando que as novas threads terminem sua execução.

Outras três primitivas para controle de threads estão disponíveis no Mulplix. A primeira, *thr_id*, retorna o identificador numérico da thread, único no processo a que a thread pertence. A segunda, *thr_kill*, permite o encerramento de uma thread por outra thread do mesmo processo. Todas as threads disparadas pela thread encerrada também são encerradas. A última primitiva, *thr_term*, conclui a execução da thread atual.

Primitivas de alocação de memória permitem a criação de regiões de memória compartilhadas ou privativas. Para memória compartilhada, a primitiva *me_salloc* oferece duas opções: espaço de alocação de memória concentrado ou distribuído. No primeiro caso, espera-se que a maior parte dos acessos seriam realizados na região de memória fisicamente local ao processador preferencial de execução da thread. A alocação distribuída deve ser utilizada quando todas as threads tiverem um padrão de acesso uniformemente distribuído. A primitiva *me_palloc* cria uma região de memória privativa da thread.

Mulplix implementa dois mecanismos de sincronização distintos. O primeiro é utilizado em relações de exclusão mútua e o segundo é utilizado em relações de ordem parcial. Para a manipulação de semáforos de exclusão mútua, estão disponíveis primitivas para criação (*mx_create*), captura (*mx_lock*), liberação (*mx_free*) e destruição (*mx_delete*) de semáforos. Além das primitivas anteriores, *mx_test* permite que uma thread capture um semáforo, se este estiver livre, sem que a thread espere por sua liberação. Mulplix implementa semáforos de exclusão mútua simples e múltiplos. Nos semáforos de exclusão mútua múltiplos, o número máximo de threads simultaneamente na região crítica é indicado no momento da criação do semáforo.

Nos semáforos de ordem parcial, que implementam sincronização de barreira, estão disponíveis primitivas para criação (*ev_create*), sinalização assíncrona (*ev_signal*), espera pela ocorrência do evento (*ev_wait*), sinalização síncrona (*ev_swait*) e destruição (*ev_delete*) de semáforos. As primitivas *ev_set* e *ev_unset* também foram implementadas para forçar a sinalização ou inicialização incondicional de um semáforo.

3.Threads no Modelo Pthreads

As rotinas definidas pelo modelo Pthreads podem ser divididas em dois grandes grupos, segundo os elementos que elas manipulam: rotinas relacionadas com threads e rotinas relacionadas com instrumentos de sincronização. Nesta seção, discutiremos as rotinas de

manipulação de threads do modelo Pthreads e sua implementação sobre as primitivas de manipulação de threads do modelo nativo Mulplix.

3.1 Disparo de Threads

O disparo de uma thread adiciona uma nova linha de execução ao processo corrente. A nova thread compartilha a área de memória global de seu processo, sendo executada em paralelo com as outras threads de seu processo e dos outros processos ativos.

Uma thread executa uma rotina do programa em que ela foi disparada. Quando a rotina executada na thread for concluída, a thread também é encerrada. No modelo Pthreads, qualquer rotina que execute em uma thread deve receber um único argumento e retornar um valor.

Pthreads define dois tipos de threads, considerando o comportamento da thread após sua conclusão: threads *detached* e threads *joinable*. Quando a rotina executada em uma thread *detached* conclui sua execução, a thread libera todos os recursos associados a ela e termina. Quando a rotina executada em uma thread *joinable* conclui sua execução, a thread aguarda até que outra thread faça um pedido pelo seu valor de retorno, liberando os recursos associados a ela e terminando somente neste momento. Como nenhuma thread pode requisitar o valor de retorno de uma thread detached, o valor retornado é ignorado.

Cada thread possui um identificador numérico único dentro de seu processo. Este identificador é utilizado em diversas operações sobre a thread. Cada thread possui também um conjunto de propriedades, chamadas de atributos da thread, que definem seu comportamento. Os atributos de uma thread, apresentados na Tabela 1, são definidos antes de seu disparo.

Atributo	Objetivo
Escopo	Indica se a nova thread será conectada a uma thread de sistema ou compartilhará uma thread do sistema com outras threads do mesmo processo. O valor *default* deste atributo indica que a nova thread compartilhará uma thread de sistema com outras threads do processo.
Tipo	Indica se a thread é *detached* ou *joinable*. O valor *default* deste atributo indica threads *joinable*.
Tamanho da pilha	Indica o tamanho, em bytes, da pilha da thread
Endereço da pilha	Indica o endereço de memória do início da pilha da thread
Política de escalonamento	Indica a política de escalonamento das threads. Três políticas são definidas no modelo Pthreads: *round-robin*, *fifo* e escalonamento *default* do sistema.
Herança de política de escalonamento	Indica se a nova thread herdará a política de escalonamento da thread que a disparou. Se este atributo mantiver seu valor *default*, a nova thread herdará a política de escalonamento da thread que a disparou.
Máscara de sinais	Indica os sinais do sistema que a thread pode processar. Uma nova thread herda a máscara de sinais da thread que a disparou.

Tabela 1 - Os atributos de uma thread Pthreads

Os atributos de uma thread são representados pelo tipo estruturado de dados *pthread_attr_t*. Estes atributos são definidos antes do disparo da thread, e somente alguns deles podem ser alterados depois que a thread estiver em execução.

A rotina *pthread_attr_init* inicializa cada atributo de thread com seu valor default. Após o disparo da thread, a estrutura de dados que representa os atributos da thread pode ser liberada, através da rotina *pthread_attr_destroy*. As rotinas apresentadas na Tabela 2 manipulam cada atributo de thread independentemente.

A política de escalonamento pode ser alterada depois que a thread for disparada. A rotina *pthread_detach* altera o tipo de uma thread para *detached*. Outros atributos da thread estão relacionados com cancelamento e serão citados posteriormente.

Rotinas	Objetivo
pthread_attr_setscope *pthread_attr_getscope*	Manipulam o escopo da thread
pthread_attr_setdetachstate *pthread_attr_getdetachstate*	Manipulam o tipo da thread
pthread_attr_setstacksize pthread_attr_getstacksize	Manipulam o tamanho da pilha da thread
pthread_attr_setstackaddr *pthread_attr_getstackaddr*	Manipulam o endereço da pilha da thread
pthread_attr_setschedpolicy *pthread_attr_getschedpolicy*	Manipulam a política de escalonamento da thread
pthread_attr_setinheritsched *pthread_attr_getinheritsched*	Manipulam o atributo de herança da política de escalonamento da thread
pthread_sigmask	Manipula a máscara de sinais da thread

Tabela 2 - Rotinas que manipulam os atributos de threads

No modelo Pthreads, o disparo de uma nova thread é executado pela rotina *pthread_create*, a partir dos atributos da nova thread, da rotina que será executada na thread e um argumento que será passado para esta rotina. Após a criação da thread, a rotina *pthread_create* retorna indicando o identificador numérico da nova thread.

A rotina *pthread_self* retorna o identificador da thread atual. A rotina *pthread_equal* recebe dois identificadores de threads e verifica se os dois são iguais.

3.2 Implementação do Disparo de Threads

Três dificuldades foram encontradas na implementação das threads do modelo Pthreads sobre o modelo Mulplix. A primeira se refere ao tipo de thread disparada. As threads do modelo Mulplix são sempre *detached*, sendo disparadas em grupos. As threads Pthreads são disparadas separadamente, mas podem ser *detached* ou *joinable*.

A segunda dificuldade reside na compatibilidade entre as interfaces esperadas pelas rotinas executadas no interior de threads dos diferentes modelos. No modelo Mulplix, a rotina disparada em uma thread recebe, como parâmetros, um argumento definido pelo usuário e seu número de ordem dentro do grupo em que foi disparada. Rotinas executadas em threads Mulplix não podem retornar valores. No modelo Pthreads, a rotina executada no interior de uma thread espera um único parâmetro, definido pelo usuário no momento do disparo da thread. Rotinas executadas em threads *joinable* podem retornar valores.

Por fim, a terceira dificuldade está relacionada com o conhecimento do identificador de uma nova thread. No modelo Mulplix, somente a própria thread conhece seu identificador.

Esta informação não é retornada para a thread que disparou a nova linha de controle. Threads do modelo Pthreads, após disparadas, retornam seu identificador para a thread que as disparou.

Estes problemas foram solucionados com a definição de uma representação interna para cada nova thread disparada. A estrutura de dados *pthread* representa uma thread Pthreads em execução. Cada novo disparo de threads cria uma nova instância da estrutura de dados abaixo, onde as informações pertinentes a nova thread são armazenadas.

```
typedef struct
{
    pthread_attr_t   attr;          /* Atributos da thread              */
    void             *(*func)(void *); /* Função da thread               */
    void             *argumento;    /* Argumento da função              */
    void             *exitstatus;   /* Retorno da rotina                */
    int              tid;           /* Identificador da thread          */
    sigset_t         sigmask;       /* Mascara de sinais                */
    handler_node     *handlers;     /* "Cleanup handlers"               */
    int              joins;         /* Numero de "joins"                */
    EVENT            termino;       /* Indica o fim da thread           */
    int              pegou_status;  /* Pegou estado de termino ?        */
    int              cancelstate;   /* Modo de cancelamento             */
    int              canceltype;    /* Tipo de cancelamento             */
    int              cancelada;     /* Cancelamento deferido ?          */
} pthread;
```

O campo *attr* armazena os atributos da thread. A estrutura *pthread_attr_t* possui um campo para cada atributo de thread, exceto para a máscara de sinais, que é armazenada separadamente, no campo *sigmask*. Cada campo de atributo é manipulado separadamente pelas rotinas relacionadas com atributos de threads, que atuam sobre a estrutura *pthread_attr_t*. A implementação atual do modelo Pthreads não suporta a definição de prioridades de threads e as políticas de escalonamento em *round-robin* e *fifo*.

A rotina *pthread_create* preenche a representação interna da nova thread e chama a primitiva *thr_spawn* do modelo nativo Mulplix, para disparar a thread requerida. A nova thread inicialmente executa uma rotina de transição, chamada *thread_bridge*, que recebe a representação interna como parâmetro.

Como as threads Mulplix são disparadas em grupo e as threads Pthreads são disparadas separadamente, a primitiva *thr_spawn* dispara grupos compostos por apenas uma thread. Esta thread recebe como parâmetros sua ordem no grupo disparado e um ponteiro para a representação interna da thread.

A rotina *thread_bridge* é uma rotina intermediária, cujo objetivo é solucionar os problemas de compatibilidade entre os dois modelos de threads. Assim que é disparada na nova thread, a rotina *thread_bridge* escreve o identificador da thread no campo *tid* de sua representação interna, recebida como parâmetro. A rotina *pthread_create*, que possui uma referência para a representação interna da thread, detecta que o identificador da nova thread foi atualizado e retorna seu valor para a thread que a disparou. Este mecanismo soluciona o problema do conhecimento do identificador da nova thread pela thread disparadora.

Tendo reconhecido seu identificador, a rotina *thread_bridge* executa a rotina que foi planejada para a thread. O campo *func* da representação interna indica a rotina que será

executada na thread, enquanto o campo *argumento* contém o parâmetro que será passado para esta rotina. A rotina indicada no campo *func* é executada por *thread_bridge*, com o devido argumento. Esta chamada resolve o problema da diferença entre o número de parâmetros esperados pelas rotinas executadas em threads Mulplix e threads Pthreads.

Após a conclusão da rotina *func*, a rotina *thread_bridge* volta a executar, chamando a rotina *pthread_exit* para encerrar a execução da thread e liberar os recursos associados a ela. Esta rotina, que será detalhada na próxima seção, recebe como parâmetro o valor de retorno da rotina *func*.

3.3 Ciclo de Vida das Threads

O ciclo de vida de uma thread começa quando esta é disparada e termina em quatro possíveis situações:

- quando a rotina executada na thread retorna. Neste caso, a thread volta a executar *thread_bridge*, que chama a rotina *pthread_exit*;
- quando a thread pede seu encerramento, chamando ela própria a rotina *pthread_exit*;
- quando a thread é encerrada por outra thread, através de um sinal;
- quando uma thread é cancelada.

A rotina *pthread_exit* encerra a execução da thread atual. Esta rotina se comporta de forma distinta nos dois tipos de threads definidos pelo modelo Pthreads. Quando encerrada, uma thread *detached* libera todos os recursos associados a ela e destrói seu fluxo de execução. Este tipo de thread não pode retornar valores, uma vez que nenhuma outra thread estará esperando por seu término.

Uma thread *joinable*, quando termina, libera todos os recursos associados a ela, mas permanece ativa até que uma segunda thread requisite seu valor de retorno. Retornado este valor, a thread destrói seu fluxo de execução.

A rotina *pthread_join* suspende a execução da thread corrente até a conclusão de uma segunda thread. Ao fim da thread alvo, *pthread_join* recebe o valor retornado pela rotina executada nesta thread. Apenas uma thread pode aguardar pela conclusão de uma thread *joinable*. Se diversas threads executarem a rotina *pthread_join* para a mesma thread alvo, somente a primeira aguardará o fim da thread.

Uma thread pode ser encerrada por outra thread através de um sinal, enviado através da rotina *pthread_kill*. Se este sinal for *SIGKILL*, *SIGABORT* ou *SIGQUIT*, a thread alvo é encerrada. Os sinais devem estar liberados pela máscara de sinais da thread.

Durante sua execução, uma thread pode requisitar recursos do sistema operacional, como semáforos, memória e identificadores de arquivos. Ao concluir sua execução, a thread deve liberar estes recursos, independentemente da forma com que ela termine. Como uma thread pode ser encerrada a qualquer momento por outra thread, através da rotina *pthread_*kill, deve existir uma forma da thread liberar seus recursos, mesmo após seu encerramento.

Para suprir esta necessidade, o modelo Pthreads define rotinas de limpeza - *cleanup handlers* -, que são rotinas implementadas pelo usuário, armazenadas em uma pilha interna da thread e executadas automaticamente quando a thread é concluída.

A rotina *pthread_cleanup_push* armazena uma rotina de limpeza, junto com um argumento que será passado para esta rotina no momento de sua execução. A rotina *pthread_cleanup_pop* desempilha e, opcionalmente, executa a rotina de limpeza do topo da pilha. Esta rotina pode ser chamada para liberar um recurso que não será mais utilizado ao longo da thread.

3.4 Implementação dos Mecanismos de Encerramento de Threads

Na implementação do modelo Pthreads sobre o modelo Mulplix, ao fim da execução da rotina original de uma thread, esta thread volta a executar a rotina *thread_bridge*. Esta rotina chama *pthread_exit*, passando como parâmetro o valor de retorno da rotina original.

A rotina *pthread_exit* armazena o valor de retorno da thread no campo *exit_status* da representação interna da thread e inicia a liberação dos recursos associados a thread. Se a thread for *detached*, todos os recursos podem ser liberados no momento de sua conclusão. Entretanto, se a thread for *joinable*, seu valor de retorno deve permanecer disponível para a thread que executar um *join* sobre ela.

As rotinas *prejoin_cleanup_thread* e *posjoin_cleanup_thread* são responsáveis pela liberação dos recursos associados a uma thread. Se a thread for *detached*, *pthread_exit* executa estas duas rotinas e retorna, encerrando a linha de execução da thread.

Se a thread for *joinable*, *pthread_exit* executa a rotina *prejoin_cleanup_thread* e aguarda que alguma thread execute um *join* sobre a thread atual. A rotina *posjoin_cleanup_thread* somente será executada após este *join*.

A rotina *prejoin_cleanup_thread* executa as rotinas de limpeza da thread, armazenadas em uma pilha, cujo topo é indicado pelo campo *handlers* da representação interna da thread. Em seguida, a rotina executa os destrutores das chaves de dados privativos da thread, que serão descritos posteriormente.

A rotina *posjoin_cleanup_thread* libera um semáforo de ordem parcial, associado ao campo *termino* da representação interna da thread, e libera a própria estrutura interna. O semáforo, utilizado apenas por threads *joinable*, atua como uma barreira para a thread *joinable* e a thread que aguarda sua conclusão.

A rotina *pthread_join* inicialmente verifica se está aguardando por sua própria conclusão ou pela conclusão de uma thread que não está em execução. Em seguida, *pthread_join* incrementa um contador, definido no campo *joins* da representação interna da thread. Este contador permite que a rotina determine se mais de uma thread está aguardando pela conclusão da thread alvo.

Tendo validado a espera pela conclusão da thread alvo, *pthread_join* entra na barreira criada pelo semáforo de ordem parcial *termino*. Em threads *joinable*, a rotina *pthread_exit* também entra nesta barreira. A barreira é liberada quando a thread alvo e a thread efetuando o *join* estiverem na barreira.

Liberada da barreira do semáforo de ordem parcial, a rotina *pthread_join* captura o valor de retorno da thread do campo *exit_status* de sua representação interna. Em seguida, *pthread_join* escreve um valor diferente de zero no campo *pegou_status* da representação interna. Este campo indica que a rotina *pthread_exit* pode liberar os demais recursos associados a thread, através da rotina *posjoin_cleanup_thread*, e retornar, concluindo a thread.

Em virtude da implementação atual do sistema operacional Mulplix (junho/97) não prover mecanismos de sinalização entre threads, a rotina *pthread_kill* se limita a concluir uma thread, quando esta recebe os sinais *SIGKILL*, *SIGABORT* ou *SIGQUIT*. Os recursos associados a thread são liberados por chamadas as rotinas *prejoin_cleanup_thread* e *posjoin_cleanup_thread*. A primitiva *thr_kill*, do Mulplix, é utilizada para o encerramento da thread alvo.

3.5 Cancelamento

Cancelamento é um mecanismo alternativo para o encerramento de threads, que difere dos métodos anteriores porque pode ser ignorado pela thread ou adiado. Cada thread possui um atributo que indica se a thread pode ser cancelada. Pedidos de cancelamento enviados a threads que não possam ser canceladas são ignorados.

Um cancelamento pode ser assíncrono ou deferido. Em cancelamentos assíncronos, a thread cancelada é encerrada no momento em que recebe o pedido de cancelamento. Em cancelamentos deferidos, a thread que recebe o pedido de cancelamento somente será encerrada em pontos específicos do programa, chamados pontos de cancelamento.

Cancelamento deferido pode ser utilizado por threads que utilizam recursos do sistema operacional durante sua execução. Se estas threads são encerradas assincronamente, sem prévia liberação dos recursos a elas associados, estes podem ser perdidos. Através do cancelamento deferido, a thread recebe o aviso de cancelamento, mas somente acata este aviso quando puder concluir sua execução, sem prejuízo para o restante do processo.

A rotina *pthread_cancel* cancela uma thread. A rotina *pthread_setcancelstate* altera o atributo que indica se a thread pode ser cancelada. Se a thread pode ser cancelada, um segundo atributo indica se este cancelamento será assíncrono ou deferido. A rotina *pthread_setcanceltype* altera o valor deste atributo.

Em threads que aceitem cancelamento deferido, os pontos de cancelamento devem ser definidos explicitamente no programa, através da rotina *pthread_testcancel*. Algumas rotinas da biblioteca padrão do compilador C e da biblioteca do modelo Pthreads definem pontos de cancelamento implícitos.

3.6 A Implementação do Cancelamento

Os três últimos campos da representação interna da thread referem-se a implementação do mecanismo de cancelamento. O campo *cancelstate* representa o atributo que indica se a thread pode ser cancelada. O campo *canceltype* representa o atributo que indica o tipo de cancelamento da thread: assíncrono ou deferido.

O cancelamento é implementado através das rotinas *pthread_kill*, quando assíncrono, e *pthread_exit*, quando deferido. A rotina *pthread_cancel* verifica se a thread pode ser cancelada. Se o cancelamento for assíncrono, *pthread_cancel* chama *pthread_kill* para concluir a execução da thread alvo. Se o cancelamento for deferido, *pthread_cancel* altera o valor do campo *cancelada* da representação interna da thread para um valor diferente de zero. A rotina *pthread_testcancel* testa o valor deste campo, chamando a rotina *pthread_exit* se ele for diferente de zero.

3.7 Memória Privativa de Threads

No modelo Pthreads, além de compartilhar a memória global do processo com as outras threads deste processo, uma thread pode requisitar regiões privativas de memória. Estas regiões de memória são manipuladas através de chaves de dados privativos.

Uma chave de dados privativos armazena uma informação para cada thread em execução no processo. A chave de dados deve ser global ao processo, permitindo que todas as threads tenham acesso a região de memória manipulada pela chave.

O processo de manipulação de dados privativos se resume a criar uma chave de dados, alterar e consultar as informações mantidas por cada thread e, quando estas informações não forem mais necessárias, destruir a chave de dados.

A rotina *pthread_key_create* cria uma nova chave de dados, associando-a opcionalmente a um destrutor. Quando uma thread conclui sua execução, o destrutor de cada chave de dados é executado, recebendo, como parâmetro, o valor que a thread armazenava na chave de dados. O destrutor pode ser utilizado para liberação de recursos armazenados na memória privativa.

A rotina *pthread_setspecific* armazena uma informação em uma chave de dados, referente a thread atual. A rotina *pthread_getspecific* consulta a informação referente a thread atual, armazenada em uma chave de dados. A rotina *pthread_key_destroy* libera as regiões de memória manipuladas por uma chave de dados. O destrutor da chave de dados não é executado quando a chave é destruída, mas quando cada thread é encerrada.

3.8 Implementação de Memória Privativa

Na implementação de memória privativa de threads, cada chave de dados é representada por uma estrutura *pthread_key*. Esta estrutura possui dois campos: um ponteiro para o destrutor da chave de dados e um vetor de ponteiros sem tipo. O vetor contém um ponteiro para cada thread, onde as informações privativas são armazenadas. Quando uma nova thread é disparada, seus ponteiros são inicializados com valores nulos.

```
typedef struct _pthread_key
{
   void    (*destructor)(void *value);          /* Funcao destrutora      */
   void    *threads[PTHREAD_THREADS_MAX];       /* Dados (por thread)     */
} pthread_key;
```

As chaves de dados são armazenadas em um vetor global. Este vetor é percorrido sempre que uma thread conclui sua execução, para a ativação dos destrutores associados às informações armazenadas por esta thread. O percurso do vetor exige a proteção de um semáforo de exclusão mútua (mutex), inicializado na primeira chamada a rotina *pthread_create*.

A rotina *pthread_key_create* percorre o vetor de chaves de dados, procurando uma entrada livre para armazenar a nova chave de dados. A rotina *pthread_key_destroy* libera a posição do vetor ocupada por uma chave. As rotinas *pthread_getspecific* e *pthread_setspecific* indexam o vetor interno da estrutura *pthread_key* pelo identificador da thread, lendo ou alterando o valor armazenado nesta posição do vetor.

4. Mecanismos de Sincronização

Três mecanismos de sincronização são definidos pelo modelo Pthreads: os semáforos de exclusão mútua (mutex), os semáforos condicionais e as chaves de execução única.

4.1 Semáforos de Exclusão Mútua

Semáforos de exclusão mútua previnem múltiplas threads de executarem simultaneamente regiões críticas de código, que manipulem recursos compartilhados. Todas as threads devem ter acesso ao semáforo que proteja a região crítica.

A rotina *pthread_mutex_init* cria um semáforo de exclusão mútua. Assim como as threads, os semáforos de exclusão mútua possuem atributos, definidos durante sua criação. O único atributo de um semáforo é seu escopo, que determina se ele será utilizado para sincronização entre threads do mesmo processo ou de processos distintos. O valor *default* deste atributo indica que o semáforo será utilizado apenas para a sincronização de threads do mesmo processo.

O atributo de um semáforo é inicializado pela rotina *pthread_mutexattr_init*. Após a inicialização do semáforo, seus atributos podem ser destruídos, através de uma chamada a rotina *pthread_mutexattr_destroy*. A rotina *pthread_mutexattr_setpshared* altera o escopo de um semáforo, enquanto *pthread_mutexatt_getpshared* retorna o escopo de um semáforo.

A rotina *pthread_mutex_lock* suspende a execução da thread atual até que o semáforo esteja livre, capturando-o em seguida. A rotina *pthread_mutex_unlock* libera um semáforo capturado pela thread. A rotina *pthread_mutex_destroy* destrói um semáforo. A rotina *pthread_mutex_trylock* verifica se o semáforo está liberado, capturando-o para a thread. Se o semáforo estiver associado a outra thread, esta rotina não suspende a execução da thread atual mas retorna um código de erro.

4.2 A Implementação de Semáforos de Exclusão Mútua

A implementação dos semáforos de exclusão mútua efetua um mapeamento direto entre as rotinas do modelo Pthreads e as rotinas de manipulação de semáforos de exclusão mútua simples do modelo Mulplix. Um semáforo Pthreads é representado pela estrutura de dados *pthread_mutex_t*. Nesta estrutura, o campo *mutex* representa um semáforo do sistema operacional, enquanto o campo *attr* representa os atributos do semáforo.

```
typedef struct _pthread_mutex
{
   MUTEX                mutex;        /* Semáforo interno      */
   pthread_mutexattr_t  attr;         /* Atributos do semáforo */

} pthread_mutex_t;
```

Na implementação atual, semáforos não podem ser compartilhados entre processos. Um semáforo de exclusão mútua somente pode sincronizar threads dentro de um mesmo processo. Esperamos implementar esta funcionalidade em uma futura versão do sistema operacional Mulplix, que permita a criação de recursos compartilhados entre processos.

4.3 Semáforos Condicionais

Em determinadas aplicações, uma thread executando em uma região crítica deve esperar por um evento. No modelo Pthreads, quando uma thread espera que outra thread comunique a ocorrência de um evento, ela deve utilizar um semáforo condicional em conjunto com um semáforo de exclusão mútua.

Um semáforo condicional permite que threads bloqueiem e testem uma condição, sob a proteção de um semáforo de exclusão mútua, até que a condição seja satisfeita. Se a condição for falsa, a thread bloqueia no semáforo condicional e libera o semáforo de exclusão mútua, permitindo que outra thread teste ou altere o estado da condição. Se alguma thread altera o estado da condição, ela deve sinalizar esta mudança para as threads que estão aguardando. Estas, por sua vez, recapturam o semáforo de exclusão mútua e reavaliam a condição.

A rotina *pthread_cond_init* cria um semáforo condicional. Semáforos condicionais também possuem atributos, definidos durante sua criação. Como nos semáforos de exclusão mútua, o único atributo de um semáforo condicional é seu escopo, que determina se ele será utilizado para sincronização entre threads do mesmo processo ou de processos distintos. O valor *default* deste atributo indica que o semáforo será utilizado para a sincronização de threads do mesmo processo.

O atributo de um semáforo condicional é inicializado pela rotina *pthread_condattr_init*. Depois da inicialização do próprio semáforo, seus atributos podem ser destruídos, através de uma chamada a rotina *pthread_condattr_destroy*. A rotina *pthread_condattr_setpshared* altera o escopo de um semáforo, enquanto *pthread_condattr_getpshared* retorna o escopo de um semáforo.

A rotina *pthread_cond_wait* bloqueia a thread corrente em um semáforo condicional, liberando o semáforo de exclusão mútua associado à condição. Quando a condição é sinalizada, o semáforo de exclusão mútua é recapturado pela thread, que é liberada para continuar sua execução. A rotina *pthread_cond_wait* define um ponto de cancelamento de thread. A rotina *pthread_cond_timedwait* é similar a rotina *pthread_cond_wait*, exceto que ela retorna com erro se a condição tiver ocorrido após um determinado intervalo de tempo, definido em um parâmetro.

Uma condição é sinalizada pelas rotinas *pthread_cond_signal* ou *pthread_cond_broadcast*, ou interrompidas por um sinal. A rotina *pthread_cond_signal* libera a primeira thread bloqueada em um semáforo condicional. A rotina *pthread_cond_broadcast* libera todas as threads bloqueadas em um semáforo condicional. Se não houverem threads bloqueadas no semáforo condicional, estas rotinas não tem efeito. A rotina *pthread_cond_destroy* destrói um semáforo condicional.

4.4 A Implementação de Semáforos Condicionais

Na implementação do modelo Pthreads, semáforos condicionais são representados pela estrutura *pthread_cond_t*. Esta estrutura mantém uma fila com as threads que estão aguardando pela condição, representada pelo campo *fila*. O campo *interno* representa um semáforo de exclusão mútua, utilizado para serializar acessos paralelos a fila de threads. O campo *attr* representa os atributos do semáforo condicional.

A fila de threads aguardando pela condição é duplamente encadeada, para facilitar a remoção de threads da fila. Cada entrada desta fila é representada pela estrutura *cond_queue*. Os campos *next* e *prev* propiciam o encadeamento da fila. O campo *tid* contém o identificador da thread que está aguardando pela condição. O campo *esperando* indica se a condição já foi sinalizada para a thread.

```
typedef struct _pthread_cond
{
    MUTEX               interno;
    cond_queue          *fila;
    pthread_condattr_t  attr;

} pthread_cond_t;
```

```
typedef struct _cond_queue
{
    int                 esperando;
    pthread_t           tid;
    struct _cond_queue  *next;
    struct _cond_queue  *prev;

} cond_queue;
```

As rotinas *pthread_cond_wait* e *pthread_cond_timedwait* inserem a thread atual na fila do semáforo, testam a condição e liberam o semáforo de exclusão mútua associado a ela. Em seguida, as rotinas entram em loop, testando consecutivamente se a condição foi liberada, através do campo *esperando* da fila da condição. Este loop define um ponto de cancelamento. Por fim, quando a condição é liberada, a thread é retirada da fila da condição e o semáforo é recapturado.

A rotina *pthread_cond_signal* percorre a fila de threads, liberando a condição para a primeira thread que estiver aguardando. Esta liberação é realizada escrevendo-se um valor diferente de zero no campo *esperando* da entrada da thread na fila. A rotina *pthread_cond_broadcast* funciona de forma análoga, liberando a condição para todas as threads que estejam na fila.

Como nos semáforos de exclusão mútua, semáforos condicionais não podem ser compartilhados entre processos na implementação atual. Um semáforo condicional somente pode sincronizar threads dentro de um mesmo processo.

4.5 Chaves de Execução Única

Em algumas situações, diversas threads desejam realizar uma tarefa que somente pode ser executada por uma delas, normalmente a primeira thread do grupo. Um exemplo destas situações é a criação de um semáforo: diversas threads podem utilizar um semáforo, mas somente uma thread deve criá-lo.

O modelo Pthreads define um mecanismo de sincronização, chamado chave de execução única, que facilita a implementação desta classe de problemas. Chaves de execução única são associadas a uma rotina, cuja execução será disputada por todas as threads. Entretanto, a chave garante que somente uma thread executará a rotina. As demais threads serão bloqueadas, até que a execução da rotina esteja concluída. A chave de execução única deve ser global a todas as threads.

A rotina *pthread_once* recebe, como parâmetros, uma chave de execução única e a rotina associada a esta. A primeira thread que chamar a *pthread_once* executará esta rotina, enquanto as outras threads aguardarão pelo fim desta execução. A chave de execução única deve ser previamente inicializada com a macro *PTHREAD_ONCE_INIT*.

4.6 A Implementação de Chaves de Execução Única

Chaves de execução única são representadas pela estrutura *pthread_once_t*. Esta estrutura contém um semáforo, no campo *mutex*, que proporciona exclusão mútua na execução da rotina associada a chave. O campo *executou* indica se a rotina associada já foi executada por alguma thread.

```
typedef struct _pthread_once_t
{
   int          executou;          /* Rotina foi executada ? */
   MUTEX        mutex;             /* Semáforo interno       */
} pthread_once_t;
```

A rotina *pthread_once* captura o semáforo da estrutura e verifica se a rotina associada já foi executada. Neste caso, o campo *executou*, inicialmente com zero, conterá um valor diferente de zero. Caso contrário, a rotina associada é executada e, em seu retorno, o campo *executou* é alterado para um valor diferente de zero. Apenas a primeira thread a capturar o semáforo executará a rotina.

A chave de execução única deve ser inicializada por uma macro. Como o semáforo da estrutura *pthread_once_t* deve ser inicializado através de uma chamada ao sistema, a rotina *pthread_once* utiliza um semáforo global para prover exclusão mútua a inicialização do semáforo da estrutura. O semáforo global é criado na primeira chamada a rotina *pthread_create*.

A rotina *pthread_once_init*, definida como uma extensão ao modelo Pthreads, representa uma alternativa a macro de inicialização de chaves de execução única. A rotina *pthread_once_destroy* outra extensão ao modelo Pthreads, libera os recursos utilizados a uma chave, destruindo seu semáforo interno.

5. Experimentos

O modelo Pthreads está operacional e seus primeiros testes foram realizados utilizando-se uma estação de trabalho SPARCStation 20 com quatro processadores HyperSPARC 100MHz. Os resultados experimentais foram derivados da utilização da implementação do modelo Pthreads sobre as primitivas do modelo Mulplix, disponíveis como uma biblioteca de rotinas sobre Solaris Light Weight Processes (LWP).

As Tabelas 3 e 4 comparam o desempenho da implementação discutida para o modelo Pthreads com o modelo Pthreads disponível para a plataforma Solaris. Dois algoritmos foram utilizados nesta comparação: SOR e eliminação gaussiana.

O algoritmo SOR utilizou uma matriz 1000 x 1000. Na paralelização da aplicação, utilizamos o paradigma mestre-escravo. A matriz foi dividida em faixas de colunas, distribuídas para threads distintas. A thread principal não participou dos cálculos, servindo apenas para a criação da matriz, o disparo e espera pela conclusão das demais threads. Além da matriz original do problema, uma matriz auxiliar, também dividida em faixas de colunas, foi utilizada para armazenamento temporário do resultado de cada iteração do algoritmo.

A Tabela 3 apresenta os tempos de execução, em segundos, representando a média dos tempos de 5 execuções da aplicação. O número de threads apresentado na tabela representa o números de threads disparadas pela thread principal, não contabilizando esta thread.

Número de Threads	Pthreads para Mulplix	Pthreads para Solaris
1	190.03	189.40
2	99.45	98.35
3	70.27	69.07
4	68.53	55.99

Tabela 3 - SOR para uma matriz 1000 x 1000

O algoritmo de eliminação gaussiana atuou sobre uma matriz 1000 x 1000. O paradigma mestre-escravo também foi utilizado na paralelização da aplicação. A matriz foi dividida em faixas de linhas, distribuídas para threads distintas. A thread principal inicializa a matriz, dispara as demais threads e espera pela conclusão destas, sem participar dos cálculos. A primeira thread disparada ficou responsável pela manipulação da linha do pivô. A Tabela 4 apresenta os tempos de execução, em segundos, representando a média dos tempos de 5 execuções da eliminação gaussiana. O número de threads apresentado na tabela representa o números de threads disparadas pela thread principal, não contabilizando esta thread.

Número de Threads	Pthreads para Mulplix	Pthreads para Solaris
1	92.28	92.29
2	48.40	48.45
3	36.77	34.90
4	35.01	30.76

Tabela 4 - Eliminação gaussiana para uma matriz 1000 x 1000

Os resultados das tabelas 3 e 4 apresentam uma pequena diferença entre os tempos de execução dos algoritmos na implementação do modelo Pthreads sobre as primitivas do Mulplix, em relação ao modelo Pthreads disponível para Solaris, até o limite de 3 threads. Julgamos que esta diferença decorra da utilização, no primeiro caso, de uma camada adicional representada pela implementação do modelo Mulplix sobre LWP. Por sua vez, o modelo Pthreads para Solaris está implementado diretamente sobre LWP, o que o torna mais rápido.

Os resultados das tabelas 3 e 4 apresentam uma sensível diferença entre os tempos de execução das duas implementações, quando 4 threads são disparadas pela thread principal. Neste caso, temos cinco threads em execução. Entretanto, a thread principal estará apenas aguardando pela conclusão das demais threads. Na implementação sobre o modelo Mulplix, utilizamos um semáforo de ordem parcial para aguardar a conclusão de uma thread. Este semáforo utiliza um semáforo condicional de LWP, na implementação das primitivas do sistema Mulplix sobre Solaris.

A implementação do modelo Pthreads sobre Solaris, que usa LWP's diretamente, utiliza a primitiva *_lwp_wait*, que faz com que a thread corrente seja retirada do escalonador, aguardando pela conclusão de um LWP sem ocupação de processador. Acreditamos que este recurso seja mais eficiente que os semáforos condicionais de sistema.

As duas condições apresentadas acima desaparecerão quando uma versão do sistema operacional Mulplix estiver disponível para testes. Neste caso, a implementação discutida será executada diretamente sobre as primitivas do sistema operacional.

6.Conclusões

Este artigo apresentou a implementação do modelo de programação paralela Pthreads sobre as primitivas do sistema operacional Mulplix. Resultados experimentais comprovaram o baixo *overhead* introduzido e o bom desempenho da implementação, comparando-a com a implementação do mesmo modelo sobre a plataforma Solaris.

Os programas utilizados nos experimentos foram compilados, sem qualquer alteração, nas duas plataformas. A compatibilidade da implementação discutida com o modelo original é quase total. A única restrição se refere à impossibilidade dos semáforos condicionais e de exclusão mútua sincronizarem threads pertencentes a processos distintos. Esperamos solucionar este problema em uma futura versão do sistema operacional Mulplix, que defina primitivas para manipulação de memória compartilhada entre processos.

Agradecimentos

Os autores gostariam de agradecer a FINEP, CNPq, RHAE e FAPERJ pelo suporte financeiro ao desenvolvimento deste trabalho.

Referências Bibliográficas

[AUD96] Aude, Júlio S. et al. "The Multiplus/Mulplix Parallel Processing Environment", Proceedings of the 1996 International Symposium on Parallel Architectures, Algorithms and Networks, Beijing, China, 1996

[AZE93] Azevedo, Rafael P. "Mulplix: Um Sistema Operacional UNIX-like para Programação Paralela", Tese de Mestrado, COPPE/UFRJ, 1993

[FALL89] Faller, N., Salenbauch, P., "Plurix: A multiprocessing Unix-like operating system", Proceedings of the 2nd Workshop on Workstation Operating Systems, IEEE Computer Society Press, Washington, DC, USA, pp. 29-36, September 1989

[GEI94] Geist Al, Beguelin A., Dongarra J., Jiang W., Mancheck R., Sunderam V., "PVM - A users guide and tutorial for Network Parallel Computing", The MIT Press, Massachusetts, 1994

[GRA95] Grahan, John R. "Solaris 2.x: Internals and Architecture", McGraw-Hill, Inc., 1995

[IEE94] Institute for Electrical and Electronic Engineers, POSIX P1003.4a, "Threads Extension for Portable Operating Systems", 1994

[MPI95] Message Passing Interface Forum, "MPI: A Message-Passing Interface Standard", MPI Forum Draft, June 1995

[SAN97] Santos, Cláudio M. P. "M-PVM: A Multithreaded PVM for Shared-Memory Architectures", Proceedings of the Ninth International Conference on Parallel and Distributed Computing and Systems (PDCS'97) to be held in Washington, D.C., U.S.A., October 13-16, 1997.

[SUN95] Sun Microsystems, Inc. "Multithreaded Programming Guide", 1995