

Trabalho Prático I – Versão 1 – 20/03/2014

Implementação de Biblioteca de *Threads*

## 1. Descrição Geral

O objetivo deste trabalho é a aplicação dos conceitos de sistemas operacionais relacionados a escalonamento e ao contexto de execução, o que inclui a criação, chaveamento e destruição de contextos. Esses conceitos serão empregados no desenvolvimento de uma biblioteca de *threads* em nível de usuário (modelo N:1). Essa biblioteca de *threads*, denominada de **microthread** ( ou apenas *mthread*), deverá oferecer capacidades básicas para programação com *threads* como criação, execução, sincronização, término e trocas de contexto.

O trabalho poderá ser desenvolvido em duplas: **não serão aceitos trabalhos desenvolvidos por grupos com mais de dois componentes**. A biblioteca *mthread* deverá ser implementada, OBRIGATORIAMENTE, na linguagem “C” e sem o uso de outras bibliotecas (além da *libc*, é claro). Além disso, a implementação deverá executar em ambiente UNIX.

## 2. Descrição Geral

A biblioteca *mthread* deverá ser capaz de gerenciar uma quantidade variável de *threads* (potencialmente grande), limitada pela capacidade de memória RAM disponível na máquina. Cada *thread* deverá ser associada a um identificador único (*tid* – *thread identifier*) que será um número inteiro, com sinal, de 32 bits (*int*).

O diagrama de transição de estados é o fornecido na figura 1 e possui os seguintes estados.

**Apto:** estado que indica que uma *thread* está pronta para ser executada e que está apenas esperando a sua vez para ser selecionada pelo escalonador. Há quatro eventos que levam uma *thread* a entrar nesse estado: (i) criação da *thread* (primitiva *mcreate*); (ii) cedência voluntária (primitiva *myield*); (iii) quando a *thread* está bloqueada esperando para entrar em uma seção crítica (*mlock*) e outra *thread* libera essa seção crítica (primitiva *munlock*), e; (iv) quando a *thread* estiver bloqueada pela primitiva *mjoin* esperando por uma *thread* e essa *thread* esperada termina

**Executando:** representa o estado em que a *thread* está usando o processador. Uma *thread* nesse estado pode passar para os estados *apto*, *bloqueado* ou *término*. Uma *thread* ~~executando~~ passa para *apto* sempre que executar uma primitiva *myield*. Uma *thread* pode passar de *executando* para *bloqueado* através da execução das primitivas *mjoin* ou *mlock*. Finalmente, uma *thread* passa ao estado *término* quando efetuar o comando *return* ou quando chegar ao final da função que executava.

**Bloqueado:** uma *thread* passa para o estado *bloqueado* sempre que executar uma primitiva *mjoin*, para esperar a conclusão de outra *thread*, ou ao tentar entrar em uma seção crítica – primitiva *mlock* – e a mesma já estiver sendo usada por outra *thread*.

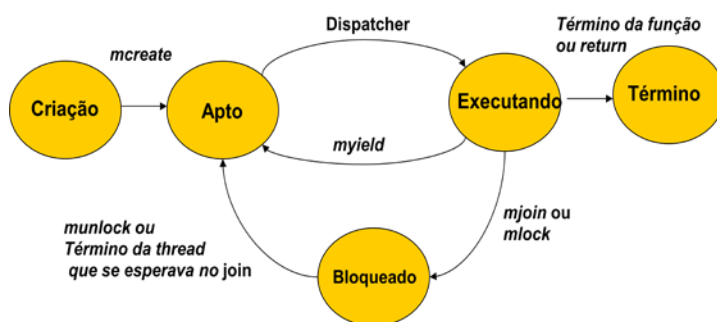


Figura 1 – Diagrama de estados e transições da *mthread*

O escalonador a ser utilizado deve seguir uma política **Shortest Process Next (SPN) não preemptiva**. Na política SPN (uma variante do SJF), as *threads*, ao entrarem no estado *Apto*, são reordenadas de acordo com a previsão futura do tempo de uso de CPU, ou seja, elas são organizadas em uma lista ordenada por ordem crescente de tempo. O cálculo de

tempo futuro deve ser feito usando uma média exponencial, como a vista em aula, com  $\alpha = 1$  (considera que o tempo da previsão futuro é idêntico ao último tempo de uso da CPU).

Ao ser criada, uma *thread* é inserida no início da lista ordenada que representa a fila de aptos, uma vez que não se tem uma avaliação do tempo previsto para sua execução. Excetuando-se a criação, toda vez que uma *thread* for colocada no estado *Apto*, ela deve ser inserida na fila de aptos, segundo seu tempo estimado de execução. Assim, quando a CPU ficar livre, o escalonador deverá selecionar a primeira *thread* da lista de aptos para receber a CPU (passar para o estado *Executando*).

### 3. Interface de programação

Para que seja possível o desenvolvimento de programas com a biblioteca *threads* é necessário que essa ofereça uma interface de programação (API) para suas funções. As funções a serem implementadas são descritas a seguir e devem, OBRIGATORIAMENTE, seguir os *prototypes* dados no arquivo *pthread.h*, fornecido como parte desta especificação. O arquivo *pthread.h* **NÃO DEVE SER ALTERADO**.

**Criação de uma *thread*:** A criação de uma *thread* envolve a alocação das estruturas necessárias à gerência das mesmas (*Thread Control Blocks*, por exemplo) e a devida inicialização. Ao final do processo de criação, a *thread* deverá ser inserida no início da lista de aptos e estará pronta para ser escolhida pelo escalonador para executar. A função da biblioteca responsável pela criação de uma *thread* está descrita no quadro abaixo:

```
int pthread_create ( void (*start_routine)(void *), void *arg);
```

**Parâmetros:**

*start\_routine*: ponteiro para a função que a *thread* executará.

*arg*: ponteiro para os parâmetros que podem ser passados para a *thread* na sua criação.

**Retorno:**

Quando executada corretamente, Retorna um valor positivo, que representa o identificador da *thread* criada; caso contrário, retorna o valor -1.

[slc1] Comentário: O tipo "uth\_tid" não foi declarado na pthread.h. Aqui deve ser "int", usado para identificar a thread criada

**Liberando voluntariamente a CPU:** uma *thread* pode liberar a CPU de forma voluntária com o auxílio da primitiva *pthread\_yield*. Se isso acontecer, a *thread* que executou *pthread\_yield* retorna ao estado *apto* através da transição *pthread\_yield* (figura 1). A notar que a *thread* é reinserida na lista de aptos de acordo com o seu tempo de execução (política SPN) e o escalonador é então chamado para selecionar a *thread* que receberá a CPU. Para realizar a liberação voluntária, deverá ser implementada a função *pthread\_yield*, conforme protótipo abaixo:

```
int pthread_yield ( void );
```

**Retorno:**

Retorna o valor 0 (zero) se a função foi realizada com sucesso; caso contrário, retorna -1.

**Sincronização de término:** a chamada *pthread\_join* bloqueia a *thread* em execução até que a *thread* identificada pelo argumento *thr* termine. Quando a *thread* identificada por *thr* terminar, a função *pthread\_join* retorna com um valor inteiro indicando o sucesso ou não em sua chamada. Uma *thread* só pode ser esperada por uma única outra *thread*, isso é, se duas *threads* fizerem *pthread\_join* esperando pelo término de uma mesma *thread*, apenas uma delas será executada com sucesso. A outra retornará **IMEDIATAMENTE** com o indicativo de erro.

```
int pthread_join(pthread_t_t thr);
```

**Retorno:**

Retorna o valor 0 (zero, se a função foi realizada com sucesso; caso contrário, retorna -1.

[slc2] Comentário: Esclarecimento dado em aula. A questão é "quando" deverá retornar: se imediatamente ou se após ter sido bloqueada. A resposta é "imediatamente".

**Exclusão mútua:** o sistema prevê o emprego de variáveis *mutex* para realizar a sincronização de acesso a recursos compartilhados (seção crítica). As primitivas existentes são *pthread\_mutex\_init*, *pthread\_mutex\_lock* e *pthread\_mutex\_unlock*. A primitiva *pthread\_mutex\_init* é usada para inicializar a variável *mutex* e deve ser chamada, obrigatoriamente, antes de utilizar a variável com as primitivas *pthread\_mutex\_lock* e *pthread\_mutex\_unlock*.

A função *pthread\_mutex\_init* inicializa uma variável do tipo *pthread\_mutex\_t*. A inicialização consiste em deixar essa variável no estado livre, isso é, liberado para qualquer *thread* tentar adquiri-lo para acessar uma seção crítica. Ainda, deve fazer parte dessa variável uma estrutura que armazena quais *threads* estão bloqueadas esperando por sua liberação. Na inicialização essa lista estará vazia.

```
int mmutex_init (mutex_t *);
```

Retorno:

Retorna o valor 0 (zero), se a função foi realizada com sucesso; caso contrário, retorna -1.

A primitiva *mlock* será usada para indicar a entrada na seção crítica. Se a seção crítica estiver livre, a entrada da *thread* corrente na seção crítica é autorizada e o valor da variável *mutex* deve passar para ocupado. Se, por outro lado, a seção crítica estiver ocupada, a *thread* será bloqueada (transição de *executando* para *bloqueado*).

```
int mlock (mutex_t *);
```

Retorno:

Retorna o valor 0 (zero), se a função foi realizada com sucesso; caso contrário retorna -1.

Ao encerra a execução da seção crítica, a *thread* deve realizar a chamada a *munlock* para liberar a seção crítica. Em havendo mais de uma *thread* no estado *bloqueado*, esperando pela liberação da seção crítica, a primeira delas deverá passar para o estado *Apto* e as demais continuarão no estado *Bloqueado*. Esse comportamento configura o comportamento de uma fila FIFO para espera do *mutex*.

```
int munlock (mutex_t *);
```

Retorno:

Retorna o valor 0 (zero), se a função foi realizada com sucesso; caso contrário, retorna -1.

#### 4. Geração da *libmthread*

As funcionalidades da *mthread* deverão ser disponibilizadas através da biblioteca denominada *libmthread.a*. Uma biblioteca é um tipo especial de programa objeto que possui o código de funções que são chamadas por outros programas. Para que isso seja possível, deve-se ligar o código da biblioteca com o do programa chamador, formando um único executável. Dessa forma, uma biblioteca é gerada a partir dos códigos fontes que implementam as funções que possui e que devem ser organizados em um formato específico. Por exemplo, suponha que os programas *arquivo1.c* e *arquivo2.c* possuem as funções que fazem parte da biblioteca *libexemplo.a*, então, o primeiro passo na geração dessa biblioteca é a compilação, cujos comandos são:

```
gcc -c arquivo1.c -Wall  
gcc -c arquivo2.c -Wall
```

Esses dois comandos geram os objetos *arquivo1.o* e *arquivo2.o*. A opção *-Wall* solicita ao compilador que informe mensagens de alerta (*warnings*) sobre possíveis erros de atribuição de valores a variáveis e incompatibilidade na quantidade ou no tipo de argumentos em chamadas de função. Os dois arquivos objetos devem ser agrupados para gerar a biblioteca *libexemplo.a*. Isso é feito através da seguinte linha de comando:

```
ar crs libexemplo.a arquivo1.o arquivo2.o
```

Após os passos acima, a biblioteca *libexemplo.a* passa a existir. Nesse arquivo estão as funções implementadas nos arquivos *arquivo1.c* e *arquivo2.c*, e que podem ser chamadas por outros programas. Suponha agora que o programa *myprog.c* faça chamadas as funções de *libexemplo.a*. Então, além de compilar *myprog.c* é preciso ligar seu código com o da biblioteca, o que pode ser feito com o comando:

```
gcc -o myprog myprog.c -lexemplo -Wall
```

A opção *-l* indica o nome da biblioteca a ser ligada. Observe que o prefixo *lib* e sufixo *.a* são assumidos automaticamente, por isso, a menção apenas ao nome *exemplo*. Essa linha de comando pressupõe ainda que o arquivo da biblioteca está localizado no diretório corrente. Se esse não for o caso, é preciso indicar a localização da biblioteca no sistema de arquivos com o auxílio da opção *-L*. No exemplo abaixo está sendo indicado que a biblioteca encontra-se no diretório */usr/lib*:

```
gcc -o myprog.c -L/usr/lib -lexemplo -Wall
```

Assim, a implementação deste trabalho resulta na criação da biblioteca *libmthread.a* que, posteriormente, para fins de teste, será ligada com um programa chamador (fornecido pelo professor). Para fazer a ligação deve-se utilizar a seguinte linha de comando:

```
gcc -o <nome_exec> <nome_chamador.c> -L<lib_dir> -lmthread -Wall
```

onde *<nome\_exec>* é o nome do executável; *<nome\_chamador.c>* é o nome do arquivo fonte onde é realizada a chamada das funções da biblioteca; e *<lib\_dir>* é o caminho do diretório onde está a biblioteca *libmthread.a*.

Faz ainda parte dessa solução o arquivo de cabeçalho (*header files*) *mthread.h* com os *prototypes* das funções disponibilizadas por *libmthread.a*, ou seja, aquelas descritas na seção 3. O arquivo de cabeçalho deve obrigatoriamente se chamar *mthread.h* e estar dentro do subdiretório *include* na estrutura de diretórios descrita na seção 8.

## 5. Empregando a *mthread*: execução e programação

A programação com *mthread* é similar aquela vista na atividade experimental 2, que utilizou *pthread* (POSIX threads). A partir do *main* de um programa C poderão ser lançadas várias *threads* através da primitiva de criação de *threads*. Cada *thread* corresponderá, na verdade, a execução de uma função desse programa C. Todas as funções da biblioteca (seção 3) podem ser chamadas pela *main*, como, por exemplo, *mjoin()*, para a *thread main* esperar por suas *threads* filhas antes de terminar.

No Moodle está disponibilizado um programa chamado “teste.c”, onde é ilustrada a criação de threads. Esse programa considera a estrutura de diretórios conforme especificado na seção 8.

Após desenvolver um programa que utilize a biblioteca, ele deve ser compilado e ligado com a biblioteca que implementa a *mthread*. A linha de comando abaixo realiza esta etapa:

```
user% gcc -o exemplo exemplo.c -L../lib -lmthread -Wall
```

Então, para executar o programa, basta fornecer o seu nome precedido de “./”, a partir da linha de comandos, conforme abaixo:

```
user% ./exemplo
```

ATENÇÃO: a opção “-L” fornece o caminho no sistema de arquivos onde estão armazenadas as bibliotecas específicas. No exemplo foi usado “../lib”, que é o caminho até a biblioteca em relação ao diretório onde o programa de teste está sendo compilado e executado (subdiretório *testes* – vide seção 8).

## 6. Material suplementar de apoio

A biblioteca definida constitui o que se chama de *biblioteca de threads em nível de usuário* (modelo N:1). Na realidade, o que está sendo implementado é uma espécie de máquina virtual que realiza o escalonamento de *threads* sobre um processo do sistema operacional. Na Internet pode-se encontrar várias implementações de bibliotecas de *threads* similares ao que está sendo solicitado. ENTRETANTO, NÃO SE ILUDAM!! NÃO É SÓ COPIAR!! Esses códigos são muitos mais completos e complexos do que aquilo que vocês precisam fazer. Utilize-os como uma fonte de inspiração.

A base para elaboração e manipulação das *mthreads* são as chamadas de sistema providas pelo GNU/Linux: *makecontext()*, *setcontext()*, *getcontext()* e *swapcontext()*. Estude o comportamento dessas funções. A atividade experimental 2 fornece exemplos úteis para a compreensão dessas funções. Analise-os!

Para implementar o escalonador é necessário obter o tempo gasto por cada processo durante sua fatia de tempo de execução. Para isso, deve-se usar funções com resolução mínima de “micro-segundos”. Existem algumas opções. Sugere-se o uso da função *clock\_gettime()*, disponível na *librt*. Para utilizar essa função essa biblioteca deverá ser incluída na ligação dos módulos, usando a opção “-lrt”.

## 7. Road map para a implementação

Algumas dicas do que precisará ser feito:

1. É preciso definir uma estrutura de dados para representar uma *thread* (o equivalente ao PCB, só que para *thread*, ou seja, um TCB). No TCB estarão todas as informações relativas a uma *thread* (*tid*, estado, contexto, etc).
2. Serão necessárias rotinas para tratamento de listas encadeadas, prevendo inserção e remoção de elementos. Os elementos da lista são os TCBs. Há no mínimo duas listas: a que implementa o estado *Apto* e a que implementa o estado *Bloqueado*. Além dessas, ainda será necessária uma fila para cada *mutex*.

3. Implementar as funções da interface. Essas funções estão intimamente relacionadas com as estruturas de dados assim como com o escalonador.

4. Implementar o escalonador com a política solicitada e o despachante (*dispatcher*). Nesse módulo serão utilizadas as funções de contexto e as de medição de tempo.

5. Será preciso elaborar um conjunto de programas de teste, junto com uma metodologia de aplicação desses programas.

## 8. Entregáveis: o que deve ser entregue?

Todos os arquivos e diretórios devem ser entregues compactados em um arquivo *tar.gz*. Esse arquivo deve estar organizado da seguinte forma:

| \mthread |               |   |
|----------|---------------|---|
|          | makefile      | ARQUIVO: arquivo <i>makefile</i> para gerar a <i>libmthread.a</i><br>Deve possuir uma regra “clean”, para limpar todos os arquivos de biblioteca gerados            |
|          | relatorio.pdf | ARQUIVO: arquivo PDF com o relatório do trabalho (respostas do questionário)  |
|          | src           | DIRETÓRIO: local onde colocar <b>todos</b> os arquivo “.c” que formam a <i>mthread</i>  |
|          | include       | DIRETÓRIO: local onde colocar <b>todos</b> os arquivo “.h” necessários para a <i>mthread</i><br>Nesse diretório deve estar o “mthread.h”                            |
|          | bin           | DIRETÓRIO: local onde será gerado o programa executável<br>(junção do executável com a biblioteca <i>libmthread</i> )   |
|          | lib           | DIRETÓRIO: local onde será gerada a biblioteca <i>libmthread.a</i> .  |
|          | testes        | DIRETÓRIO: fonte dos programas empregados para teste da biblioteca.<br>Deverá ser fornecido um <i>makefile</i> para compilar esses programas junto com a biblioteca |

O trabalho deverá ser entregue até a **data prevista**. Admite-se a entrega do trabalho com até duas semana de atraso. Trabalhos com até uma semana de atraso serão descontados em 20 pontos (do total de 100 pontos); trabalhos com mais de uma semana e menos do que duas semanas de atraso serão descontados em 40 pontos (do total de 100 pontos). Não serão aceitos trabalhos entregues além das duas semanas de tolerância.

## 9. Avaliação

Para que um trabalho possa ser avaliado ele deverá cumprir com as seguintes condições:

- Entrega dentro dos prazos estabelecidos;
- Obediência à especificação (formato e nome das funções);
- Compilação e geração da biblioteca e dos programas de teste sem mensagens de erro ou *warnings*;
- Fornecimento do *tar.gz* com a estrutura interna solicitada;
- O relatório deve estar completo.

O trabalho será avaliado da seguinte forma:

1. 1 ponto: uso das melhores práticas de programação: clareza e organização do código, programação modular, *makefiles*, arquivos de inclusão bem feitos (sem código C dentro de um *include*!!) e comentários adequados. Obediência à especificação significa gerar a biblioteca conforme especificado, entregar os arquivos do trabalho em *tar.gz*, seguir estrutura de diretórios fornecida na seção 8, etc.
2. 3 pontos: documentação. Resposta ao questionário e a correta associação entre a implementação e os conceitos vistos em aula.
3. 6 pontos: funcionamento da *libmthread* de acordo com a especificação. Para essa verificação serão empregados programas de teste desenvolvidos pelo professor para essa finalidade.

## 10. Observações

Recomenda-se a troca de ideias entre os alunos. Entretanto, a identificação de cópias de trabalhos acarretará na aplicação do Código Disciplinar Discente e a tomada das medidas cabíveis para essa situação.

O professor da disciplina reserva-se o direito, caso necessário, de solicitar uma demonstração do programa, onde o grupo será arguido sobre o trabalho como um todo. Nesse caso, a nota final do trabalho levará em consideração o resultado da demonstração.

#### 11. Arquivo *mthread.h* – fornecido através do Moodle

É obrigatório o uso do arquivo de inclusão (*header file*) “*mthread.h*”, onde estão os protótipos das funções a serem implementadas na realização do trabalho. O arquivo “*mthread.h*” **NÃO DEVE SER ALTERADO**.

Notar que no arquivo *mthread.h* foi incluído o arquivo “*mdata.h*”. O arquivo “*mdata.h*” deve ser fornecido pelo grupo e nele devem ser colocadas todas as definições das estruturas de dados criadas pelo grupo e que são necessárias para a compilação da biblioteca e dos programas de teste.

#### 12. Questionário – deve ser respondido no relatório

1. Nome dos componentes do grupo e número do cartão.
2. Descrição da plataforma utilizada para desenvolvimento. Qual o tipo de processador (número de cores, com ou sem suporte HT)? Qual a distribuição GNU/Linux utilizada e a versão do núcleo? Qual a versão do gcc? O trabalho foi desenvolvido em um ambiente virtualizado? Em caso afirmativo, qual a máquina virtual foi utilizada (versão)?
3. Indique, para cada uma das funções que formam a interface, se as mesmas estão funcionando corretamente. Para o caso de não estarem funcionando adequadamente, descrever qual é a sua visão do por que desse “não funcionamento”.
4. Descreva todas as estruturas de dados utilizadas na implementação, inclusive a *mmutex\_t*. É importante associar sua implementação com os conceitos estudados.
5. Liste todas as chamadas de sistema e descreva brevemente suas operações desde que uma *thread* chamou a função *myyield()* até que a próxima *thread* seja posta em execução. É importante associar sua implementação com os conceitos vistos em aula.
6. Liste todas as chamadas de sistema e descreva brevemente suas operações desde que uma *thread* chamou a função *mjoin()* até que a próxima *thread* seja posta em execução. Explícite o que acontece caso uma *thread* execute *mjoin()* para esperar por uma *thread* que já tenha encerrado. É importante associar sua implementação com os conceitos vistos em aula.
7. Liste todas as chamadas de sistema e descreva brevemente suas operações desde que uma *thread* chamou a função *mlock()* até ela continuar em execução ou ser bloqueada. É importante associar sua implementação com os conceitos vistos em aula.
8. Liste todas as chamadas de sistema e descreva brevemente suas operações desde que uma *thread* chamou a função *munlock()* e tenha provocado o desbloqueio de outra *thread*. É importante associar sua implementação com os conceitos vistos em aula.
9. Qual foi a metodologia utilizada para testar a biblioteca? Isso é, quais foram os passos (e programas) efetuados para testar a *mthread* desenvolvida? Foi utilizado um *debugger*? Qual?
10. Quais as principais dificuldades encontradas no desenvolvimento e quais as soluções empregadas para contorná-las.