INE 5412 Sistemas Operacionais I - 2011/2

Trabalho Implementação 1 Data de entrega/apresentação: 16-Nov-2011

OBS:

`O Trabalho poderá ser executado em grupos de até 3 alunos.`

Título: [Unix, C/C++] Implementação de Gerenciador de Threads

Motivação

A estrutura de um pacote gerenciador de threads no nível de usuário é bem semelhante a estrutura de um sistema operacional simples. Trabalhar com este código é bem mais fácil do que trabalhar em um SO real e proporciona o mesmo tipo de experiência. Além disso, depois que voce terminar o projeto voce poderá executar programas baseados em threads na sua própria biblioteca de threads e não ter de confiar na implementação do Solaris threads, ou Pthreads por exemplo.

1. Descrição

Projeto e implementação de um pacote *gerenciador de threads de nível de usuário*, **MT – Minhas Threads**.

A implementação será feita no contexto de processo de usuário, você tem um único processo UNIX e irá criar várias atividades (threads) no contexto do processo UNIX e gerenciar estas threads. O UNIX continuará a ver isto como um único processo e é responsabilidade da biblioteca desenvolvida por você fornecer rotinas de *criação*/*terminação*/*gerencia* de threads para o usuário. As rotinas de interface da biblioteca serão descritas abaixo. O gerenciador poderá ser desenvolvido em qualquer sistema UNIX-like que ofereça o suporte necessário. O LINUX e o MINIX3 oferecem este suporte.

O gerenciador deve ser capaz de chavear de uma thread para outra a partir de uma chamada explícita ***MT_yield()*** que pode ser invocada por uma thread. O chaveamento de contexto envolve apenas pilha e registradores relativos ao contexto das threads que devem ser salvos e restaurados. Além disso, o gerenciador deve implementar um esquema de escalonamento *round-robin* que chaveia entre threads com um *quantum* que pode variar de 10-100 milisegundos. Este escalonador também deve usar a *prioridade da thread* para tomar decisões de escalonamento. Cada thread deve ser descrita por um **descritor de thread** que deve conter todas as informações de gerencia da thread, inclusive informações de escalonamento.

2. API do Pacote MT

Esta seção descreve a interface de programação da aplicação (API) que o pacote deve fornecer. As funções devem ser implementadas precisamente como especificadas.

int MT_init(void)

Esta função inicializa a biblioteca threads e deve ser chamada antes de qualquer outra função. A chamada torna o processo a primeira de várias threads. Retorna 0 se bem sucedido e -1 na existência de erro. Esta função deve ser responsável por inicializar as estruturas de controle do gerenciador de threads. A função deve ser chamada antes de qualquer outra invocação ao gerenciador.

*int MT_create(void (*func) (int), int arg, int pri);*
Esta função cria uma nova thread para executar a função especificada ***(*func) (int)*** passando para a mesma o argumento especificado ***arg***. A prioridade da thread também é especificada em ***pri***. A função a ser executada deve ser do tipo *void func (int arg)*. O valor de retorno é o identificador da thread (**idt**) no caso de sucesso ou -1 para indicar erro.
Ex: *idt = MT_create(thread1(), argumento1, prioridade)*

*int MT_join(int idt, int *result);*
Esta função faz a thread chamadora esperar que a thread especificada termine, funciona como um *wait()*. Se o segundo argumento não é nulo, o código de término da thread é colocado na posição indicada pelo apontador. É permitido múltiplas threads esperarem pelo término de uma determinada thread. O valor de retorno é 0 se bem sucedida e -1 na ocorrência de erro (quando a thread especificada não existe). É permitido esperar uma thread que já terminou. Entretanto, depois da primeira chamada de *TU_join* os recursos da thread terminada serão devolvidos e a thread não existirá mais. Os parâmetros da função permitem esperar por uma determinada thread *idt* recebendo o estado de término da mesma em *result.*

void MT_exit(int status);
Esta função termina a thread chamadora. O parâmetro desta função permite que a thread chamadora indique o estado de término.

void MT_yield();
Esta função libera a CPU, ou seja, a thread corrente libera voluntariamente a CPU. Neste caso o escalonador deverá escolher uma nova thread para executar, ou seja, chavear contexto.

int MT_gettid(void);
Esta função retorna o identificador da thread (*idt*) chamadora.

3. O Escalonador
Toda thread recebe um *timeslice*. Quando o *timeslice* termina, uma decisão de escalonamento é tomada para encontrar a próxima thread a executar. A biblioteca deve suportar 10 níveis de prioridade sendo 0 o mais alto e 9 o mais baixo. Utilizar política *round-robin* para encontrar a próxima thread entre threads do mesmo nível de prioridade.

4. Dicas
O mais difícil na implementação de threads é a necessidade de chaveamento entre elas. Sempre que uma função é chamada, um *stack frame* é colocado na pilha do processo. Isto armazena informação de estado da computação (função) pendente – quais os valores correntes dos registradores, valores de variáveis locais e onde retornar uma vez terminada a função.

Como fazer isto no nível de usuário? Na verdade o que precisamos é a habilidade de salvar e restaurar os registradores de uso geral da CPU e atribuir valor ao apontador de pilha para apontar para a pilha corrente. Isto na verdade é feito utilizando código assembler apropriado, não em linguagem de alto nível.

Entretanto, não é preciso acesso aos registradores especiais de mapeamento de memória e assim não é preciso executar em modo kernel para chavear contexto entre threads. Existe uma maneira de armazenar os valores dos registradores em um buffer e recarregar estes valores mais tarde. Estas funções são *setjmp* e *longjmp (<setjmp.h>).* Um material bem explicativo sobre estas funções pode ser encontrado em :
http://web.eecs.utk.edu/~huangj/cs360/360/notes/Setjmp/lecture.html
http://en.wikipedia.org/wiki/Setjmp.h
http://pubs.opengroup.org/onlinepubs/009695399/functions/setjmp.html
http://pubs.opengroup.org/onlinepubs/009695399/functions/longjmp.html

Outra alternativa oferecida pelo sistema Unix é as funções *makecontext*, *setcontext*, *getcontext*, *swapcontext (<ucontext.h>)* que são chamadas de sistema para fazer troca de contexto. Segue alguns links para material destas funções:
http://en.wikipedia.org/wiki/Setcontext
http://pubs.opengroup.org/onlinepubs/007908799/xsh/ucontext.h.html
http://www.squarebox.co.uk/cgi-squarebox/manServer/usr/share/man/man0p/ucontext.h.0p
http://www.opensource.apple.com/source/xnu/xnu-1456.1.26/bsd/sys/ucontext.h

Para a construção da biblioteca de gerenciamento de threads proposta vocês devem verificar qual das abordagens é mais adequada (entendimento de cada grupo) e utiliza-la. Estas ferramentas facilitam principalmente a questão de troca de contexto, salvamento e carga de contexto. Estas ferramentas estão bem documentadas e vocês podem buscar a descrição mais formal das mesmas no *man* do Unix. Exemplos de utilização podem ser encontrados em **http://www.inf.ufsc.br/~fernando/ine5355/programas/threads** .

A seguir algumas dicas de utilização das funções em funcionalidades do pacote:
- ***Tabela de threads*** : Cada thread tem um BCT- Bloco de Controle da Thread, que contém as informações de contexto da thread.
- ***Criação de novas threads*** : Utilizar a função *setjmp()* ou a chamada *makecontext()*
- ***Chaveamento entre threads***: Utilizar as funções *setjmp()* e *longjmp()* conjuntamente ou as chamadas *getcontext()/setcontext()/swapcontext()*
- ***Escalonamento***: Utilizar as funções *getitimer() e setitimer()* e tratadores de sinal para capturar *SIGALRM* e escalonar a próxima thread de acordo com a política de escalonamento *round robin*. Com estes recursos é possível implementar a contagem de tempo (*quantum*).

5. Testes
Para a realização de testes no ambiente implementado deverá ser utilizado o programa desenvolvido no laboratório 4 – utilizando a API Pthreads, com as devidas adaptações com relação a utilização da API do pacote desenvolvido. Outros programas teste estão disponíveis em **http://www.inf.ufsc.br/~fernando/ine5355/programas/threads**