Universidade Federal do Rio Grande do Sul
Instituto de Informática
Departamento de Informática Aplicada
INF01142 - Sistemas Operacionais IN

Trabalho Prático 1
Implementação de Biblioteca de Threads

Rodrigo Favalessa Peruch - 237626
Vinícius Pittigliani Perego - 242287

Porto Alegre, 29 de Setembro de 2016

Funções

- *Ccreate*: funcionando corretamente.
- *Cyield:* funcionando corretamente.
- *Cjoin:* funcionando corretamente.
- *Csem_init:* funcionando corretamente.
- *Cwait:* funcionando corretamente.
- *Csignal:* funcionando corretamente.

Testes

- *foo_bar:* nesse teste é utilizado as funções *ccreate, cyield e cjoin.* São criadas duas threads a partir da main, que se bloqueia esperando a primeira thread executar. As duas threads ficam se intercalando em um simples laço da função que cede a CPU toda vez que é encontrado um múltiplo de 15.
- *exclusao_mutua:* criação de um *mutex* (semáforo onde o número de recursos é 1) e duas threads. A primeira thread realiza o bloqueio do semáforo e cede a CPU. Outra thread tenta acessar o mesmo recurso e é bloqueada. Esse teste serve para garantir o funcionamento de exclusão mútua do semáforo, testando as funções de semáforo (*csem_init*, *cwait* e *csignal*).

Dificuldades

Após implementar corretamente a função *ccreate* e validar o escalonamento por loteria sem nenhum problema, foi iniciado o desenvolvimento da função *cyield.*

Essa primeira função que exige troca de contexto entre os processos gerou dificuldades na implementação quando as threads encerravam pois ocorriam problemas do tipo *segmentation fault* ou o processo todo era encerrado por uma thread que não era a main (mesmo sem um comando nas funções).

Decidimos então não continuar o desenvolvimento das outras funções até que a *cyield* estivesse funcionando corretamente pois, caso contrário, só iríamos propagar mais erros adiante e ficaria extremamente complicado o processo de depuração.

A primeira ideia para contornar esse problema foi alocar memória para o contexto *uc_link* de cada contexto criado dentro da função *ccreate* e dentro das funções *cyield*, *cjoin* e *cwait* salvar esse contexto com *getcontext()* em *uc_link* para que uma vez terminada a thread, a thread que chamou a execução anteriormente voltaria a executar a partir dali.

Apesar de ter funcionado nos testes mais simples, novos problemas surgiram quando ocorriam múltiplas chamadas em threads filhas das funções de troca de

contexto em casos onde o contexto salvo do *uc_link* não era mais válido pois ele estava apontando para um contexto de uma função que já tinha terminado.

Concluímos que nosso entendimento das funções de contexto não estava sólido o suficiente e que precisávamos entender claramente o funcionamento de tais funções para que pudéssemos obter sucesso em nosso trabalho.

Após várias tentativas frustradas e muito estudo sobre como poderíamos resolver o problema tivemos a idéia de criar uma função responsável dispatcher. Se trata de um contexto inicializado ao usar a biblioteca *cthread* que aponta para o início da função dispatcher. Todas as threads agora ao serem criadas com *ccreate* possuem o campo *uc_link* apontando para o contexto desse dispatcher. Uma vez terminada a execução de uma thread, o dispatcher assume a execução e se encarrega de escalonar outra thread e a colocar em execução.

Grande parte da inspiração que tivemos para superar esta dificuldade, que foi a maior no decorrer do trabalho, vem de um conselho que nos foi dado pelo Prof. Carissimi. Ele sugeriu que começássemos o desenvolvimento somente após entender perfeitamente e claramente o funcionamento das funções de contexto.

Depois de compreendido o projeto, a implementação das demais funções *cjoin*, *csem_init*, *cwait* e *csignal* ocorreu sem problemas ou complicações com todos os arquivos de exemplo fornecidos funcionando corretamente.