Sistemas Operacionais - Trabalho 1 - Relatório

Isadora Pedrini Possebon - 00228551

2016/2

1 Proposta

O trabalho aqui apresentado consiste em desenvolver uma biblioteca de criação e gerenciamento de threads a nível de usuário (modelo N:1). Abaixo, são apresentadas a descrição da implementação de cada primitiva, bem como os mecanismos de escalonamento e despacho e testes.

2 Escalonador

Implementado de acordo com o que foi estudado em aula, a função de escalonamento de threads consiste em escolher a thread que ocupará a CPU. Porque a política é do tipo sorteio, o primeiro passo é sortear um ticket. A thread cujo ticket mais se aproximar a aquele sorteado tem o direito de ocupar a CPU. Em caso de empate, a thread de menor TID deve ganhar a CPU.

Após definir a thread que será executada, chama-se o *dispatcher* com o TID da thread correspondente, para que salve e defina o contexto a ser executado. A decisão de explicitar duas funções diferentes para cada etapa (escalonador e despachante), bem como a passagem do parâmetro entre uma e outra deve-se a maior clareza do fluxo. Como explicado na seção 6, o código foi estruturado de forma a deixar as ações sempre explícitas.

3 Dispatcher

O dispatcher é o responsável por, dada a thread que ocupará a CPU, retirá-la da fila *ready* e colocá-la em execução; isto é, passar o controle da execução para a mesma, utilizando a função *setcontext*. Para isso, a função implementada recebe o TID da thread que será executada. Ainda, é importante destacar que a thread que está em execução é indicada pela variável *current_tcb*, que detém o TCB da thread atual.

4 Primitivas

Todas as primitivas apresentadas abaixo funcionam corretamente para todos os testes e exemplos propostos. Sua implementação corresponde ao que foi estudado em aula e detalhes mais específicos serão abordados para cada primitiva.

4.1 ccreate

A implementação desta primitiva seguiu os conceitos vistos em aula. Cria-se um molde para o novo contexto com a primitiva *getcontext* e, em seguida, criamos, de fato, o contexto desta nova thread, passando o ponto de entrada, o ponto final e os parâmetros - utilizando *makecontext*. Após criado o contexto, atribui-se o mesmo para a thread, gera-se um novo TID e um ticket para esta thread. Finalmente, a thread é inserida na fila de aptos (*ready*) e a primitiva retorna o TID correspondente, se executada corretamente. Caso contrário, retorna -1.

Os testes utilizados consistiam em verificar a correta criação de cada thread, com um novo TID e contexto definido pela função enviada como parâmetro. Os testes verificam, essencialmente, a passagem de parâmetros para a função que deve ser executada quando a thread for chamada e a inserção da nova thread criada na fila *ready*. Os arquivos utilizados exclusivamente para este fim foram *teste_ccreate.c* e *teste4.c*, que serão abordados na seção 5.

4.2 cyield

Ao contrário do esperado, esta primitiva não utiliza a funçaõ *swapcontext*. Em vez disso, utiliza uma flag de controle e a função *getcontext*. A flag controla o fluxo de execucação, de forma que após salvar o contexto da thread atual, alteramos o valor da flag, inserimos a thread na fila *ready* e chamamos o escalonador. Desta maneira, quando recuperarmos esse contexto, a thread voltará para este mesmo ponto, mas não será recolocada na fila *ready* e nem chamará o escalonador - continuará para o seu contexto final. Essa decisão deve-se a dificuldade de visualização dos passos do programa; optei por deixar mais claro o fluxo, como será explicado na seção 6.

O teste dedicado a esta primitiva está no arquivo *teste_cyield*, que consiste em criar uma thread e ceder a CPU, para que esta thread seja executada. Em seguida, outra thread é criada e realiza-se *cjoin* para as duas threads já criadas. O programa finaliza quando as duas threads executam e retornam à função *main*.

4.3 cjoin

Além de ter sido implementada com o mecanismo de flag e *getcontext* explicado na subseção anterior, esta primitiva utiliza uma lista chamada *joined_tcbs*.

Essa lista contém estruturas do tipo *JOINED*, que relacionam duas threads: uma thread que está à espera do término da outra - código abaixo.

```
1 typedef struct s_JOINED{
2       int tid;
3       int waiting_for_tid;
4 } JOINED;
```

Com isso em mente, ao fazermos um *cjoin*, uma estrutura deste tipo é criada é adicionada à lista *joined_tcbs*. Assim, quando uma thread for finalizada, procuramos por seu TID na lista *joined_tcbs* e desbloqueamos todas as threads que estavam à sua espera - essas threads voltam para a fila *ready*.

Para tratar casos em que a thread pela qual deveríamos esperar não existe ou já terminou, utiliza-se uma função auxiliar *existsTID*, que verifica se o TID correspondente está na lista *blocked* ou *ready*. Se não estiver, é porque a thread não existe, está executando ou já executou.

Para verificar o funcionamento desta primitiva, utilizei testes como *teste1.c*, que cria 50 threads, fazendo com que a main espere o término de todas elas. Os testes *teste4.c* e *teste5.c* também foram utilizados, criando um grande número de threads e esperando o seu término - geralmente intercalando com um *cyield()*, dado que o escalonador implementado é não-preemptivo.

4.4 csem_init

A inicialização do semáforo é feita atribuindo o número de recursos disponíveis e alocando a fila de espera do mesmo. Todos os testes de semáforos foram realizados juntos e estão, essencialmente, no arquivo *teste_sem.c*. Para maior segurança, os exemplos fornecidos também foram testados utilizando as libs e funcionaram corretamente.

4.5 cwait

A primitiva *cwait* também foi implementada sem a utilização da função *swapcontext*, utilizando a flag anteriormente mencionada e chamando o escalonador, caso o recurso não esteja disponível.

Como apresentado em aula, a thread chama a primitiva *cwait* para solicitar o recurso associado a um dado semáforo. Se este recurso estiver disponíve, isto é, se o contador do semáforo for positivo, o recurso é atribuído a esta thread, que continua sua execução. Caso contrário, decrementa-se o contador do semáforo, adiciona-se a thread que solicitou o recurso na fila de espera do semáforo, salvamos o seu contexto atual e chamamos o escalonador. Isto é feito com a flag explicada anteriormente e a função *getcontext*.

4.6 csignal

No caso de *csginal*, a thread que chamou a primitiva está liberando o recurso em questão. Isto é, caso alguma thread estivesse à espera do recurso, a primeira thread da fila de espera é selecionada. Essa thread sai de *blocked* e retorna ao estado apto - o que é feito através da função auxiliar *unlockThread*.

4.7 cidentify

Para a implementação desta primitiva, aloca-se a memória necessária para a variável *name*, de acordo com o tamanho passado por parâmetro e copia-se os dados necessários para esta variável - nome e número do cartão do aluno estão na variável *student*.

5 Testes

Abaixo, estão descritos todos os arquivos de testes utilizados para as primitivas. Funcionalidades mais específicas foram testadas ao longo do desenvolvimento do projeto e, portanto, não possuem arquivos específicos.

As primitivas desenvolvidas executaram corretamente para todos os testes implementados.

5.1 teste_ccreate.c

O objetivo deste teste é simplesmente criar 3 e retornar a main com o TID de cada uma. Ao retornar à main, imprime-se a mensagem "Teste finalizado com sucesso".

Ainda, este teste também serve para testar a primitiva *cidentify*. Quando executado corretamente, imprime a string do aluno (Isadora Pedrini Possebon) e o número do cartão com apenas 15 caracteres e, na linha abaixo, as informações completas.

5.2 teste_cjoin.c

Este teste cria duas threads com a primitiva ccreate e atribui a cada uma delas uma função que altera os valores de i e j, respectivamente. Em seguida, a thread main faz *cjoin* para cada uma das duas novas threads. Sabendo que i e j são variáveis globais, ao final do teste, retorna-se à main e imprime-se os novos valores de i e j, isto é, $i = 10$ e $j = 50$.

Além de testar o correto retorno à thread main, este teste visa verificar se a main de fato espera pelo término da execução das duas threads - permanecendo bloqueada até que terminem -, o que é verificado pela mudança de valor das variáveis.

5.3 teste_cyield.c

Duas threads são criadas na função main. No entanto, entre a criação dessas duas threads, chama-se *cyield*, para que a thread que foi criada imediatamente antes possa executar antes de criar a segunda thread, quando retornar à main.

A ideia é testar se a thread vai para a fila *ready* e é escolhida pelo escalonador (visto que será a única em *ready*). Ainda, o teste garante que ao finalizar sua execução, a primeira thread retornará à main para então criar e executar a segunda thread.

5.4 teste1.c

Este teste cria 50 threads e faz, na main, *cjoin* para cada uma delas. Isto é, o teste executa corretamente quando todas as 50 threads imprimem a sua mensagem e retornam à main. O objetivo desde arquivo é testar a primitiva *cjoin*.

5.5 teste2.c

Desenvolvido para testar o correto funcionamento do semáforo, este teste cria um grande número de threads compartilhando o mesmo recurso, a variável *workdone*. Utilizando as primitivas *csem_init*, que inicializa um semáforo com um recurso disponível; *cwait* para solicitar o acesso à variável; e *csignal* para liberar o recurso.

Quando executado corretamente, o teste imprime "Work # done", onde # correponde ao n-ésimo trabalho realizado. Uma vez que temos apenas um recurso disponível, apenas um work pode ser executado de cada vez. Portanto, a variável work será incrementada de uma unidade por vez, retornando à main ao final de todas as threads.

5.6 teste3.c

Este teste é bem simples e apenas cria 500 threads e faz a main esperar pelo término de todas elas. O objetivo aqui era testar se a main não teria problemas em esperar por um número grande de threads.

Quando executado corretamente, o teste imprime que todas as threads foram executadas corretamente e imprime uma mensagem dizendo que a main foi finalizada.

5.7 teste4.c

O objetivo do teste é testar a passagem de parâmetros entre a *ccreate* e a função que deve ser executada pela thread. Ainda, como isso funciona em

conjunto com a primitiva *cjoin*. Para isso, são criadas 5 threads, cada uma passando um parâmetro diferente para a função que deve executar. Ainda, fazemos um *cjoin* na main para todas essas threads.

Quando executado corretamente, o teste imprime todos os parâmetros na tela (números de 1 a 5), na ordem em que as threads foram executadas.

5.8 teste_sem.c

Este teste cria 4 threads compartilhando a mesma função e variável global *resource* na main e faz *cjoin* para todas elas. Espera-se que, em momento algum, o número de threads utilizando o recurso (mensagem que é impressa na tela) seja maior do que o valor do recurso disponível. Ainda, todas as threads devem ser executadas e o programa apenas será finalizado após o término de todas essas threads, imprimindo "Fim" quando retornar à main e encerrar sua execução.

6 Dificuldades

A principal dificuldade encontrada está relacionada a abstração dos dados. Foi difícil concretizar os conceitos de thread, troca de contexto e o ponto de retorno da thread - mais especificamente, como criar um contexto para o escalonador. Por conta dessas dificuldades, optei por fazer um código maior e mais explícito (não utilizei swapcontext, por exemplo, para não ocultar a informação de salvar o contexto atual), para facilitar o entendimento do fluxo. Também, criei uma função que serve como ponto final para todas as threads executadas, chamada *finishThread*; essa função é utilizada para criar o contexto final de cada uma das threads.

Também contornar estas dificuldades, os programas exemplo foram de grande ajuda, bem como os estudos dirigidos. Destaque para os programas que mostravam como a biblioteca seria utilizada: foi importante para entender melhor o contexto do trabalho. Uma vez entendidos estes conceitos, o desenvolvimento do trabalho foi mais simples.

Outra dificuldade encontrada foi o ponto exato do código em que deveria ser feito free das posições de memória alocadas. Inicialmente, essa operação era feita na função auxiliar *finishThread*, anteriormente explicada. No entanto, não obtive resultado e decidi seguir com o desenvolvimento sem me preocupar com o gerenciamento de memória neste nível, visto que não era prioritário no escopo deste trabalho.

7 Considerações finais

No makefile geral do projeto, a libcthread é criada de acordo com as especificações do trabalho. No entanto, obtive diversos problemas ao executar os programas de teste utilizando a lib (utilizando as flags -L../src -lm -lthread). O programa era compilado sem erros e executava até que uma primitiva da lib fosse chamada; quando chamava alguma primitiva, recebia um segmentation fault. A única exceção deste problema é na primitiva cidentify, que ao invés de retornar um segmentation fault, apenas não salvava os dados na variável enviada - retornando lixo na variável em questão.

Após pesquisas e ajuda de alguns colegas, acredito que o problema esteja relacionado a memória: memory leak eou buffer overflow. No entanto, não consegui resolver o problema e optei por compilar os programas de teste incluindo os arquivos necessários. Para mostrar melhor o problema, dois arquivos Makefile estão disponíveis: o arquivo testes/Makefile contém a compilação incluindo os arquivos necessários e resulta na correta execução de todos os programas; o arquivo testes/lib_Makefile contém as regras de compilação utilizando a lib criada, que resulta em segmentation fault.