Trabalho 1

Relatório

André Dexheimer Carneiro - 00243653

INF01142 - Sistemas Operacionais 1N
Professor Alexande Carissimi

1. Funções da biblioteca cthread

1.01 - ccreate() - Funcionando corretamente
1.02 - cyield() - Funcionando corretamente
1.03 - cjoin() - Funcionando corretamente
1.04 - csem_init() - Funcionando corretamente
1.05 - csignal() - Funcionando corretamente
1.06 - cwait() - Funcionando corretamente

2. Programas de teste

Todos os programas citados a seguir funcionaram como o planejado obtiveram o resultado esperado.

2.1 - 1teste_thread:

Funções testadas: *ccreate(), cyield()*

O objetivo desse teste é verificar o correto funcionamento do escalonador e os valores de retorno das funções.

Para isso o programa cria 3 threads e designa a cada uma delas uma função diferente, a cada função é passado um valor inteiro arbitrário como parêmetro, os valores são colocados na tela por cada thread a fim de verificar se estão corretos.

Além disso, a função *main* faz duas chamadas à *cyield()* para obrigar todas as *theads* a serem executadas antes do seu término.

Como saída do teste é esperado o seguinte fluxo de execução, o qual pode ser verificado pela ordem em que são feitas as escritas na tela:

main(cyield()) → func0 → func1 → main(cyield()) → func2 → main

2.2 - 2teste_thread:

Funções testadas: *ccreate(), cyield(), cjoin()*

Com este teste, espera-se verificar o correto funcionamento da *cjoin(),* para isso são estabelecidas algumas dependências que levam algumas *threads* ao estado bloqueado a espera do fim da execução de outras, lembrando que uma *thread* só pode ser aguardada por uma outra *thread*.

O fluxo de execução esperado é o seguinte:

main(cjoin(ID1)) → ID0(cyield()) → ID1(cjoin(ID0)) → ID2(cjoin(ID0) negado) → ID3 → ID0 → ID1 → main

2.3 - 3teste_thread:

Funções testadas: *ccreate(), cyield(), csem_init(),cjoin(), cwait(), csignal()*

O foco deste programa é testar as chamadas que envolvem semáforo, para tanto, ele traça um fluxo de execução usando as funções *cyield(), cwait()* e *cjoin().* O semáforo utilizado no programa possui *count = 2,* o que permite a duas threads executar a região crítica ao mesmo tempo, qualquer outra que tente adentrar é colocada na fila dos bloqueados.

Além disso o programa ainda testa a função *cjoin()* nos casos em que o TID inserido:

- Pertence à uma thread inexistente.
- Pertence à uma thread que já está sendo esperada por outra .
- É igual ao da thread que está chamando a função.

O fluxo de execução esperado é o seguinte:

main → ID0 → ID1 → ID2 → ID3 → ID0 → main → ID1 → main

3. Dificuldades encontradas na implementação do trabalho

1: Funções relativas ao contexto:

A primeira dificuldade foi utilizar as funções de contexto corretamente, pois o uso delas involve uma maneira diferente de organizar o código a fim de impedir que uma *thread* volte ao ponto errado quando fizer um *setcontext()* para um contexto salvo anteriormente. Isso resultou em problemas na criação da função do escalonador e na função *ccreate()*.

2: *Floating Point Exception*

Definitivamente, o maior problema foi o erro de *floating point exception,* que apareceu diversas vezes quando era executada alguma *thread*. O problema ocorria sempre depois que a *thread* acabava a execução da sua função correspondente, quando ela se dirigia ao *uc_link.* Após dias tentando achar o problema e uma visita ao gabinete do professor Alexandre, consegui solucionar o erro, que se devia a alguns problemas. O primeiro era a maneira como era organizada a chamada à função *getcontext()* para definir o contexto do escalonador, a qual acabava levando a mesma *thread* a executar uma recursão não desejada e sutil. O segundo era a própria chamada à função *getcontext()*, que era feita no momento errado e fazia com que a thread voltasse em muito no código. A solução do problema foi reorganizar o escalonador e definir uma *thread* ao contexto, fazendo uma chamada a *makecontext()* e definindo a área de pilha ao invés de simplesmente uma chamada a *getcontext()*.