Universidade de Brasília
Instituto de Ciências Exatas
Departamento de Ciência da Computação

Aula Prática 02 - *Threads*

Identificação da Disciplina

Código da Disciplina	Nome da Disciplina	Turma	Professor	Período
117935	Programação Concorrente	A	Jeremias Moreira Gomes	2019/2

Objetivo da Aula Prática

O objetivo desta aula prática é dar familiaridade ao aluno na manipulação básica de *threads* do Sistema Operacional UNIX.

Detalhes Acerca das Aulas Práticas

0.1. Restrições deste Trabalho Prático

Durante a escrita de quaisquer códigos, tentar não utilizar bibliotecas exóticas que não sejam as disponíveis no UNIX como biblioteca padrão. Se o fizer (pois ajuda na realização de testes, por exemplo), lembrar que no momento da correção o sistema padrão a ser utilizado é similar a uma instalação padrão das máquinas do LINF, e nada além do código será instalado apenas para uma correção individual. Então o uso de uma `<gtest.h>`, por exemplo, pode acabar fazendo com que o aluno perca pontos, porque seu código não pôde ser testado, em virtude da falta desta no computador de testes.

O atraso da entrega da atividade, incorre no decondo de 0.5 pontos a cada 60 minutos de atraso, em relação ao prazo de entrega da atividade e da nota final do aluno nesta atividade.

Esta atividade pode ser feita em dupla.

0.2. Por onde entregar os exercícios das aulas práticas?

As atividades envolvendo as aulas práticas deverão ser entregues via Aprender (https://aprender.ead.unb.br), na disciplina de Programação Concorrente. As informações para ingresso são as seguintes:

- URL: https://aprender.ead.unb.br/course/view.php?id=6775
- Chave de Acesso: `s3nhaD3ss3semestre201902`

Após o ingresso na disciplina, haverá uma atividade chamdada "Aula Prática 02 - *Threads*", para submissão dos exercícios.

0.3. O que deverá ser entregue, referente as aulas práticas?

Deverão ser entregues respostas referentes a todos os tópicos das **Seções de Atividades** ao longo deste documento. Essas atividades (dessa aula prática) estão divididas em duas categorias:

- Questionários (pergunta e resposta).
- Elaboração de Códigos.

Todos os dois tipos de atividades deverão ser entregues em um documento único contendo todas as respostas. Esse documento deverá ter identificação do aluno (nome e matrícula), identificação da disciplina, identificação da aula prática e as respostas identificadas de maneira igual as numerações em que aparecem neste documento. Para auxiliar na elaboração desse documento, pode-se utilizar esse documento (clique aqui) de referência.

Além disso, as questões de elaboração de código deverão ser entregues em arquivos (.c) separados, sendo um arquivo para cada código elaborado. O início desse código deverá vir com comentários fazendo referência ao autor do código, nome do arquivo e a identificação da atividade, da seguinte forma:

```
// autor: Jeremias Moreira Gomes
// arquivo: exemplo-arquivo.c
// atividade: 0.0.0

#include <stdio.h>

int main()
{
    printf("\n");
    return 0;
}
```

Assim, os códigos elaborados irão estar em arquivos separados e no relatório.

0.4. Como entregar as atividades das aulas práticas?

A submissão das atividades deverá ser feita em um arquivo único comprimido no tipo zip (`Zip archive data, at least v1.0 to extract`) contendo um diretório com o relatório e os códigos elaborados durante a atividade. Além disso, para garantir a integridade do conteúdo entregue, o nome do arquivo comprimido deverá possuir duas informações (além da extensão `.zip`):

- A matrícula do aluno (ou dos alunos separadas por hífen, em caso de dupla).
- O *hash* md5 do arquivo `.zip`.

Para gerar o md5 do arquivo comprimido, utilize o comando `md5sum` do Linux e em seguida faça o renomeamento utilizando o *hash* coletado. Exemplo:

```
[6189] j3r3mias@tardis:aula-01-processos|master > zip -r aaa.zip atividade-01/
  adding: atividade-01/ (stored 0%)
  adding: atividade-01/relatorio.docx (stored 0%)
  adding: atividade-01/01-hello-3-fork.c (deflated 34%)
  adding: atividade-01/03-exemplos.c (deflated 47%)
  adding: atividade-01/02-arvore.c (deflated 35%)
  adding: atividade-01/02-pid_t.c (deflated 39%)
  adding: atividade-01/01-hello-fork.c (deflated 21%)
  adding: atividade-01/03-processos-e-ordens.c (deflated 55%)
[6190] j3r3mias@tardis:aula-01-processos|master > ls -lha aaa.zip
-rw-rw-r-- 1 j3r3mias j3r3mias 2,5K set  4 23:26 aaa.zip
[6191] j3r3mias@tardis:aula-01-processos|master > md5sum aaa.zip
fe36b6799eae103a464cbc4857fce404  aaa.zip
[6192] j3r3mias@tardis:aula-01-processos|master > mv aaa.zip 160068444-fe36b6799eae103a464cbc4857fce404.zip
[6193] j3r3mias@tardis:aula-01-processos|master > ls -lha 160068444-fe36b6799eae103a464cbc4857fce404.zip
-rw-rw-r-- 1 j3r3mias j3r3mias 2,5K set  4 23:26 160068444-fe36b6799eae103a464cbc4857fce404.zip
[6194] j3r3mias@tardis:aula-01-processos|master > md5sum 160068444-fe36b6799eae103a464cbc4857fce404.zip
fe36b6799eae103a464cbc4857fce404  160068444-fe36b6799eae103a464cbc4857fce404.zip
[6195] j3r3mias@tardis:aula-01-processos|master >
```

1. *Hello Threads*

A ideia básica desta seção é mostrar o funcionamento básico da utilização de *threads* no contexto do sistema operacional. Nesta lista (e nesta disciplina) a utilização de *threads* será feita por meio da biblioteca NPTL, que é uma implementação da especificação POSIX *Threads* (PThreads). Nessa implementação, *threads* criadas pelo usuário, utilizando a biblioteca, tem um relacionamento 1x1 com as *threads* do núcleo, o que constitui uma das maneiras mais simples de implementação de *threads*.

O arquivo de cabeçalho que provê acesso a utilização dessa biblioteca de *threads* é o `<pthread.h>`. Além da inclusão do cabeçalho, também faz-se necessário ligar o código implementado à biblioteca `libpthread`, por meio da flag de compilação `-lpthread` (no gcc).

Abaixo, segue um exemplo de código básico, utilizando da biblioteca `<pthread.h>`:

```
#include <stdio.h>
#include <pthread.h>

void *rotina()
{
    printf("Olá, sou uma thread\n");
}

int main()
{
    pthread_t t;
    pthread_create(&t, NULL, rotina, NULL);
    printf("Olá, sou a main.\n");
    return 0;
}
```

Durante a execução desse código, ou um código *multithreading* genérico, ocorre o seguinte:

- Quando o programa inicia, `main()` é executado por uma *thread* principal.
- A função `pthread_create` é responsável por criar uma outra *thread*.
- Uma *thread* termina sua execução quando:
  - Retorna para a função que a criou.
  - Executa uma função chamada `pthread_exit`.
- Um programa *multithreading* termina quando:
  - A *thread* principal termina.
  - A chamada de sistema `_exit` é invocada.

Neste código, apesar de pequeno, diversas observações devem ser feitas. A primeira delas é a respeito do identificador `pthread_t` que é uma variável para receber o identificador único da *thread* criada. Ele é, usualmente, um inteiro longo sem sinal. Em seguida, uma *thread* foi criada por meio da função `pthread_create`, cujo a finalidade é criar uma *thread* que executa uma função especificada em um de seus argumentos.

Essa função possui a seguinte sintaxe:

```
int pthread_create(pthread_t *id, const pthread_att_t attr,
                   void *(*rotina)(void *), void *arg)
```

Onde:

- `*id` - Recebe a identificação a *thread* a ser criada.
- `*attr` - É uma estrutura de dados que configura o modo de funcionamento da *thread*.
- `rotina` - A função que será executada pela *thread*, assim que ela for criada. Essa função possui o seguinte protótipo: `void *rotina(void *)`.
- `*arg` - Estrutura de dados que irá alimentar os parâmetros de entrada da função executada pela *thread*.

Diferente do uso somente de processos, visto anteriormente, uma consequência fundamental começa a ocorrer com *threads* ao ter que fazer uso de funções de outras bibliotecas como a de entrada e saída `<stdio.h>`, por exemplo. A função `printf` não foi (na maioria das implementações) concebida para lidar com chamadas simultâneas dela antes do término de sua execução em diferentes linhas de execução, pois ela faz uso de variáveis globais que são compartilhadas entre outras *threads* e isso acaba gerando um comportamento errôneo ou inesperado do resultado. O nome dado a funções que foram projetadas para funcionar com diferentes linhas de execução ao mesmo tempo, são chamadas de **Funções Reentrantes**. Funções reentrantes são caracterizadas por:

- Não utilizar variáveis de uma forma não atômica[1], a menos que estas sejam variáveis privadas (pilha de tarefa da função que a chamou).
- Não chamar uma outra função que não seja ela própria uma função reentrante.
- Não utilizar o hardware de forma não atômica.

Ao longo da disciplina, essas características serão melhor exploradas na forma de soluções aplicadas ao manipular programas concorrentes. Por agora, basta ter em mente situações de conflito que podem ocorrer simplesmente pelo uso de *threads*.

1.1. Atividades

1.1.1. Implementação do *Hello Threads*

Implemente o programa *Hello Threads*, similar a forma como foi apresentada anteriormente, criando duas *threads* e fazendo-as imprimir o mesmo conteúdo cada uma.

1.1.2. Execute o programa implementado no tópico anterior diversas vezes e elenque os motivos os quais as saídas entre execuções são diferentes entre si.

[1] Atomicidade é o ato de não poder ser interrompido durante o seu evento fazendo com que este seja executa de uma vez só.

2. Controle de Linhas de Execução

Como já dito anteriormente, quando um programa inicia a sua execução, a sua linha de execução decorre de uma *thread* principal. A partir desta, outras *threads* podem ser criadas, por meio da função `pthread_create` e destruídas com a função `pthread_exit`.

A função `pthread_exit`, além do término da *thread*, também provoca a liberação dos recursos que esta estava a consumir. Assim, esta pode ser utilizada para realizar liberação de recursos adiantada, no meio de uma execução.

Após a criação, uma *thread* pode executar, em relação a *thread main* de duas formas: de forma desconexa (*detached*) à quem a criou ou de forma unida a quem a criou. Por padrão, *threads* são criadas unidas à *thread* que as criou e ao utilizar a função `pthread_create` para criar novas linhas, a função `pthread_join` é a responsável por fazer essa linha esperar com que as *threads* criadas terminem a sua execução. Isso é feito da seguinte forma:

```
#include <stdio.h>
#include <unistd.h>
#include <pthread.h>

void *rotina()
{
    printf("Estou ligada ao meu criador\n");
    sleep(2);
    pthread_exit(NULL);
    printf("Os recursos já foram liberados, então não vou executar.\n");
}

int main()
{
    pthread_t t;
    printf("Criando um thread.\n");
    pthread_create(&t, NULL, rotina, NULL);
    printf("Esperando a thread therminar.\n");
    pthread_join(t, NULL);
    printf("A thread que eu criei terminou.\n");
    return 0;
}
```

Além disso, a passagem de dados para uma *thread*, durante a sua criação, permite transferir inteiros, cadeias de caracteres ou structs como argumento para função a ser execução. Essa passagem pode ser feita da seguinte forma:

```c
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <pthread.h>

void *impressao(void *conteudo)
{
    int *valor = malloc(sizeof(int));
    *valor = *(int *)conteudo;
    printf("Meu valor é: %d\n", *valor);
    *valor = (*valor) * 2;
    pthread_exit(valor);
}

int main()
{
    pthread_t t1, t2;
    int v1 = 10, v2 = 20;
    void *r1, *r2;
    pthread_create(&t1, NULL, impressao, (int *)&v1);
    pthread_create(&t2, NULL, impressao, (int *)&v2);
    pthread_join(t1, &r1);
    pthread_join(t2, &r2);
    printf("As threads terminaram.\n");
    printf("Os valores retornados foram: %d %d\n", *(int *)r1, *(int *)r2);
    free(r1);
    free(r2);
    return 0;
}
```

2.1. Atividades

2.1.1. Divisão de Tarefas

Escreva um programa para calcular a solução da função $f(x) = 10x^2 + 42x^3$. O programa irá receber como entrada o valor e x (*double*), e deverá criar duas *threads*: uma para calcular o valor de $10x^2$ e outra para calcular o valor de $42x^3$. Por último, a *thread main* deverá receber como retorno os resultados dessas tarefas feitas pelas *threads*, somar os resultados e exibir na tela o resultado com precisão de 4 casas decimais.

3. Múltiplas *Threads* e Escalonamento

Além das chamadas pontuais, é possível criar grupos de *threads* para executar rotinas similares, por meio do uso de laços de repetição. Dessa forma, diversas linhas podem ser criadas e estas irão executar a mesma rotina, diferenciando-se apenas pelos dados de entrada fornecidos pelos parâmetros de entrada da função a ser executada. Exemplo:

```
#include <stdio.h>
#include <pthread.h>

void *tarefa(void *id)
{
    long int tid = (long int )id;
    printf("Olá, eu sou %ld\n", tid);
}

int main()
{
    const int NUMTHREADS = 5;
    pthread_t threads[NUMTHREADS];

    for (long int i = 0; i < NUMTHREADS; i++) {
        pthread_create(&threads[i], NULL, tarefa, (void *) i);
    }

    for (long int i = 0; i < NUMTHREADS; i++) {
        pthread_join(threads[i], NULL);
    }

    return 0;
}
```

Repare que, ao chamar o programa anterior (teste!), apesar do código deixar transparecer que as *threads* estão sendo criadas na ordem em que o laço está sendo executado, não necessariamente a impressão na tela irá ocorrer nessa mesma ordem e isso se deve a diversos fatores relacionados ao escalonamento das *threads* que estão prontas para a execução. Se você estiver executando em uma máquina com mais de um núcleo, lembre-se que há o mapeamento da linha de execução que você criou com uma *thread* no modo núcleo (modelo 1x1) e estas, individualmente, também concorrem entre si pelo recurso de E/S (no caso, saída padrão) para exibir o seu resultado, quando ganham o processador. Por essas diversas influências e ainda outros motivos que serão explorados nesse curso, não se pode fazer assunções temporais sobre a execução de *threads* ou processos, ainda que o código pareça óbvio aos olhos de quem está programando ou lendo algum código escrito.

3.1. Atividades

3.1.1. Iterações entre *Threads*

Escreva um programa que crie uma rotina chamada `contador`. Esta rotina deverá conter um laço que irá imprimir a seguinte frase, a cada iteração (de 5 iterações ao todo), "Sou a thread TID e estou no número N.", em que TID representa um identificador para a *thread* e N o número da iteração em que a *thread* se encontra naquele momento. Na *main*, crie 10 *threads* que irão executar a rotina `contador`.

3.1.2. Escalonador

Em sistemas Unix, as *threads* passam por um critério de escalonamento, ao decidir qual será a próxima a ser executada (dentre aquelas prontas). O algoritmo utilizado para decidir a próxima *thread* a ser executada, é chamado de *Completely Fair Scheduler* (CFS) (link1 e link2). Esse algoritmo faz um ajuste do que ele decide por uma "parcela justa" de execução para cada *thread* no processador. Pelo exercício anterior, é possível que uma *thread* tenha tido a chance de executar todas as suas instruções dentro da sua parcela de tempo ou então tendo que ceder o processador a outra *thread* antes do término. Sua tarefa nesta atividade é utilizar a biblioteca `<sched.h>` no código desenvolvido no item anterior, incluindo a função `sched_yield` logo após a impressão (ainda dentro do laço), para que sempre que uma *thread* imprima na tela, ela vá para o final da fila de pronto e permita que outra entre em execução. Para evitar o uso de múltiplos núcleos, acrescente também as seguintes modificações:

- Incluir a definição `#define _GNU_SOURCE` para ter acesso a algumas funções que são omitidas do padrão POSIX.
- Acrescentar o seguinte trecho de código (ou algo similar) na *thread main*, para indicar a utilização de um único núcleo:

```
cpu_set_t mascaranucleos; // Bitmask para os núcleos
CPU_ZERO(&mascaranucleos);// Remove a seleção de todos os núcleos
CPU_SET(0, &mascaranucleos); // Marca somente um núcleo 0 para uso
sched_setaffinity(0, sizeof(cpu_set_t), &mascaranucleos);
```

4. Variáveis Compartilhadas

Uma das principais motivações para a utilização de *threads* é o compartilhamento do espaço de endereçamentos, permitindo que as mesmas compartilhem dados inicializados, dados não inicializados e até segmentos da pilha. Dessa forma, deve-se prestar atenção na distinção de variáveis por meio de escopo, pilha ou segmentos globais. Exemplo de escopo:

```
#include <pthread.h>
#include <stdio.h>
#include <unistd.h>

void *rotina(void *emptyvar)
{
    int contador;

    printf("Contador, na thread, inicial: %d\n", contador);
    contador = 20;
    printf("Contador, na thread, final: %d\n", contador);
}

int main()
{
    pthread_t t;
    int contador;

    contador = 100;
    printf("Contador, na main, antes da criação da thread: %d\n", contador);

    pthread_create(&t, NULL, rotina, NULL);
    pthread_join(t, NULL);

    printf("Contador, na main, após o término da thread: %d\n", contador);
    return 0;
}
```

Nesse caso, a variável `contador` não tem seu conteúdo alterado, ainda que possuam o mesmo nome, por motivo de escopo.

Exemplo de contexto global:

```
#include <pthread.h>
#include <stdio.h>
#include <unistd.h>

int contador;

void *rotina(void *emptyvar)
{
    printf("Contador, na thread, inicial: %d\n", contador);
    contador = 20;
    printf("Contador, na thread, final: %d\n", contador);
}

int main()
{
    pthread_t t;

    contador = 100;
    printf("Contador, na main, antes da criação da thread: %d\n", contador);

    pthread_create(&t, NULL, rotina, NULL);
    pthread_join(t, NULL);

    printf("Contador, na main, após o término da thread: %d\n", contador);
    return 0;
}
```

Agora, a variável global `contador` é utilizada ao mesmo tempo, tanto pela *thread main* quanto pela outra *thread* criada ao longo da execução do programa.

4.1. Atividades

4.1.1. Questionário

De maneira análoga a demonstrada entre *threads*, variáveis globais podem compartilhar informações em tempo de execução entre dois processos diferentes, onde um dos processos foi criado pela chamada `fork` do primeiro? Por quê?

4.1.2. Corrida de *Threads*

Crie um programa com as seguintes características:

- O programa deverá conter uma variável global chamada `largada`, que deverá ser inicializa com o valor 3.

- A *thread* principal deverá criar um grupo de 10 *threads* que irão executar uma rotina chamada `carro`.
- A rotina `carro` funciona da seguinte forma:
  - Cada *thread* possui o seu identificador.
  - No início da execução, a rotina deverá ficar checando se o valor da variável é zero.
  - Se o valor for zero, a rotina irá imprimir "Sou o carro ID e terminei a corrida." e terminar sua execução.
- A *thread* principal deverá criar uma *thread* que irá executar a rotina `juizdelargada`.
- A rotina `juizdelargada` funciona da seguinte forma:
  - Possui um laço que decrementa a variável global `largada` até 0.
  - Dentro desse laço, a *thread* deve dormir um tempo aleatório entre 1 e 10 segundos.
  - Quando a *thread* sair do laço, ela deverá imprimir "GO!" e terminar sua execução.