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

Aula Prática 04 - Dormir e Acordar

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 apresentar técnicas básicas de comunicação entre processos/*threads* com sincronização por meio primitivas envolvendo bloqueio de processos.

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 04 - Dormir e Acordar”, 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 também 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. Mutexes

Mutexes são uma versão simplificada de semáforos (que serão explorados na seção seguinte) e são utilizados para exclusão mutua permitindo que apenas uma *thread* por vez acesse um recurso. Se uma *thread* tenta acessar um recurso que outra *thread* está bloqueando, ela é impedida e libera o processador para que outras threads executem. Isso garante que uma thread não desperdice processamento porque está aguardando por um recurso bloqueado por outra thread.

Em C, a própria biblioteca `<pthread.h>` possui funções para criação e manipulação de mutexes.

1.0.1. pthread_mutex_init

```
int pthread_mutex_init (pthread_mutex_t *mutex, const pthread_mutexattr_t *attr)
```

Descrição: Inicializa um objeto mutex.

Parâmetros:

- `pthread_mutex_t` - Deve ser um ponteiro para um objeto mutex.
- `pthread_mutexattr_t` - Atributos de inicialização do mutex (usualmente NULL).

1.0.2. pthread_mutex_destroy

```
int pthread_mutex_destroy (pthread_mutex_t *mutex)
```

Descrição: Destrói um objeto mutex.

Parâmetros:

- `pthread_mutex_t` - Um ponteiro para um objeto mutex.

1.0.3. pthread_mutex_lock

```
int pthread_mutex_lock (pthread_mutex_t *mutex)
```

Descrição: Obtém um objeto mutex ou é bloqueado.

Parâmetros:

- `pthread_mutex_t` - Um ponteiro para um objeto mutex.

1.0.4. pthread_mutex_trylock

```
int pthread_mutex_trylock (pthread_mutex_t *mutex)
```

Descrição: Obtém um objeto mutex ou falha.

Parâmetros:

- `pthread_mutex_t` - Um ponteiro para um objeto mutex.

1.0.5. pthread_mutex_unlock

```
int pthread_mutex_unlock (pthread_mutex_t *mutex)
```

Descrição: Libera objeto mutex.

Parâmetros:

- `pthread_mutex_t` - Um ponteiro para um objeto mutex.

Abaixo, segue um exemplo de utilização de mutex.

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

pthread_mutex_t mutex;

void *tarefa(void *empty)
{
    pthread_mutex_lock(&mutex);
    printf("Um por vez.\n");
    sleep(2);
    pthread_mutex_unlock(&mutex);
}

int main()
{
    pthread_t t[5];

    pthread_mutex_init(&mutex, 0);

    for (int i = 0; i < 5; i++) {
        pthread_create(&t[i], NULL, tarefa, NULL);
    }

    for (int i = 0; i < 5; i++) {
        pthread_join(t[i], NULL);
    }

    pthread_mutex_destroy(&mutex);

    return 0;
}
```

1.1. Atividade

1.1.1. Controle de Assinatura de uma Lista de Chamadas

Um dos exemplos mais simples de cenários estudantis envolvendo a disputa por recursos, é a assinatura da lista de chamadas pelos alunos no final de uma aula. Sempre que o professor libera a lista para assinatura, somente um aluno por vez consegue assinar a lista (considere que só existe esse cenário). Dessa forma, implemente um programa, utilizando *pthreads* e as funções `pthread_mutex_lock` e `pthread_mutex_unlock`, que gerencie o recurso compartilhado "lista de chamadas", por meio de um contador (assinaturas) e um vetor em que cada aluno (representado por uma *thread* que chama uma função chamada `aluno`). Quando de posse da lista de chamadas (que deverá ser exclusiva), o aluno irá incrementar o contador, colocar seu id no vetor (também compartilhado) na posição em que ele recebeu antes de incrementar o contador, dormir por 1 segundo (para representar o tempo gasto pelo aluno assinando a lista) e imprimir a frase "`Aluno <id> assinou a lista.`". O número n de alunos ($1 < n < 100$) da turma deverá ser informado na entrada do programa e a seção crítica deverá se controlada por um mutex chamado `possedalista`. Após todos os alunos assinarem a lista, a *thread main* deverá imprimir o seguinte: "`A ordem de assinatura da lista foi: X Y Z W B''`, onde "`X Y Z W B\verb`" deverão ser os ids das *threads* na ordem em que as assinaturas foram realizadas.

2. Semáforos

São mecanismos que permitem que um determinado número de *threads* tenham acesso a um recurso. Agindo como um contador que não deixa ultrapassar um limite.

No momento em que um objeto de semáforo é criado, é especificada a quantidade máxima de *threads* que ele deve permitir. Então, cada *thread* que queira acessar o recurso, deve chamar uma função que decrementa em 1 o semáforo (*down*) e, após utilizar o recurso, chamar uma função que incremente em 1 o semáforo (*up*). Quando o contador do semáforo chega a zero, significa que o número de threads chegou ao limite e o recurso ficará bloqueado para as threads que chegarem depois, até que pelo menos uma das threads que estão utilizando o recurso o libere, incrementado o contador do semáforo.

Em C, a biblioteca para utilização de semáforos é a `<semaphore.h>` que possui as seguintes funções (principais, mas não únicas), para realizar a utilização de semáforos:

2.0.1. sem_init

```
int sem_init (sem_t *semaforo, int pshared, int valor);
```

Descrição: Inicializa um objeto de semáforo.

Parâmetros:

- `semaforo` - Deve ser um ponteiro para um objeto semáforo.
- `pshared` - Define o escopo do semáforo, que também pode ser utilizado para sincronizar *threads* em processos diferentes. Se todas as *threads* são do mesmo processo, então o valor deve ser 0 (zero).
- `valor` - valor do semáforo, ou seja, a quantidade de *threads* que ele deve suportar.

2.0.2. sem_destroy

```
int sem_destroy (sem_t *semaforo);
```

Descrição: Destrói um objeto de semáforo, liberando a memória utilizada.

Parâmetros:

- `semaforo` - Ponteiro para um objeto semáforo.

2.0.3. sem_wait

```
int sem_wait (sem_t *semaforo);
```

Descrição: A função `sem_wait` faz uma requisição de acesso ao semáforo, se o número de *threads* ainda não chegou ao limite, então a *thread* obtém o acesso, senão aguarda até receber acesso.

Parâmetros:

- `semaforo` - Ponteiro para um objeto semáforo.

2.0.4. sem_post

```
int sem_post (sem_t *semaforo);
```

Descrição: A função `sem_post` deve ser usada após a *thread* ter chamado a função `sem_wait`, indicando que não precisa mais de acesso ao recurso compartilhado e permitindo que outras threads que estejam esperando pelo recurso, obtenham acesso.

Parâmetros:

- `semaforo` - Ponteiro para um objeto semáforo.

Abaixo, segue um exemplo de utilização de semáforos com uma limitação de acesso de três *threads* por vez à região crítica.

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

#define NUMTHREADS 10
#define AOMESMOTEMPO 3

sem_t semaforo;
```

```
void *tarefa(void *args)
{
    int id = *((int *)args) ;

    sem_wait(&semaforo);
    printf("%d\n", id);
    sleep(3);
    sem_post(&semaforo);
}

int main()
{
    pthread_t t[NUMTHREADS];
    int *id;

    sem_init(&semaforo, 0, AOMESMOTEMPO);

    for (int i = 0; i < NUMTHREADS; i++) {
        id = (int *) malloc(sizeof(int));
        *id = i;
        pthread_create(&t[i], NULL, tarefa, (void *)id);
    }

    for (int i = 0; i < NUMTHREADS; i++) {
        pthread_join(t[i], NULL);
    }
    sem_destroy(&semaforo);

    return 0;
}
```

2.1. Atividade

2.1.1. O Problema da Visitação em um Museu

Um grupo de 90 turistas resolveram realizar uma visitação a um museu de artes todos ao mesmo tempo. Esse museu é composto por três salas, cada um com suas respectivas capacidades e o gerenciamento de visitas do museu obriga os visitantes a percorrerem as três salas em ordem, para que todos consigam passar por tudo e para que o guia turístico consiga explicar bem os conteúdos da sala para todos os visitantes. A sala 1 tem capacidade para 10 pessoas, enquanto a sala 2 possui capacidade para 6 pessoas e a sala 3 possui capacidade para 18 pessoas. Cada sala possui um guia, e as explicações para essa sala só se iniciam quando toda uma leva de turistas completa essa sala.

Dessa forma, sua tarefa é implementar a visitação dos turistas a esse museu, utilizando *pthreads* e semáforos. Cada turista será representado por uma *thread*, cada sala um semáforo e os três guias representados por um único semáforo binário (para fins de simplificação, pois será o momento em que a função

`printf` será chamada). Cada *thread* deverá chamar a função `turista`, que executará a visitação em cada uma das salas. O guia turístico deverá controlar a quantidade de pessoas dentro da sala, e sempre que um novo grupo de turistas completar a sala, ele deverá imprimir "Sala X completa, começando a explicação.". E por último, antes de saírem de cada sala, os turistas deverão dormir por 2 segundos, como forma de representar o tempo gasto por cada um dentro de cada sala.

3. Variáveis de Condição

As variáveis de condição possibilitam outro modo de sincronização de *threads*. Enquanto as variáveis de exclusão mútua implementam a sincronização através do controle do acesso aos dados, as variáveis de condição permitem a sincronização de *trheads* através do valor desses dados. Sem a utilização destas variáveis, *threads* têm que verificar continuamente se uma dada variável tem um valor específico, ocupando assim tempo de processamento. Uma variável de condição obtém os mesmos resultados sem a necessidade de verificar continuamente algum valor.

De maneira análoga aos mutexes, e por utilizarem também o pacote `<pthread.h>`, suas chamadas são similres (não vou colocar aqui para não ocupar mais espaço). Dessa forma, segue um exemplo:

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

#define NUMTHREADS 1

pthread_cond_t condespera;
pthread_mutex_t mutex;

void *liberador(void *empty)
{
    pthread_mutex_lock(&mutex);
        printf("Sou o incrementador. Daqui um segundo, libero o esperador\n");
        sleep(1);
        pthread_cond_signal(&condespera);
    pthread_mutex_unlock(&mutex);
}

void *esperador(void *empty)
{
    pthread_mutex_lock(&mutex);
        pthread_cond_wait(&condespera, &mutex);
        printf("Sou o esperador e já esperei.\n");
    pthread_mutex_unlock(&mutex);

}

int main()
```

```
{
    pthread_t l, e;

    pthread_mutex_init(&mutex, NULL);
    pthread_cond_init(&condespera, NULL);

    printf("Criando o esperador.\n");
    pthread_create(&e, NULL, esperador, NULL);
    printf("Dormindo 2 segundos antes de criar o incrementador.\n");
    sleep(2);

    pthread_create(&l, NULL, liberador, NULL);
    pthread_join(l, NULL);
    pthread_join(e, NULL);

    pthread_mutex_destroy(&mutex);
    pthread_cond_destroy(&condespera);

    return 0;
}
```

3.1. Atividade

3.1.1. Entrega de Pacotes

Implemente um programa para simular a recepção (atendimento) e a entrega de pacotes de uma agência dos correios de uma cidade pequena que possui um único funcionário. Seu programa deverá utilizar mutex(es) e variáveis condicionais, para resolver o problema.

Inicialmente, o funcionário permanece na agência, esperando que existam pelo menos 10 pacotes para ele sair para entregar (uma *thread* que permanece dormindo até que seja acionada). Quando o número de pacotes chega a 10, o funcionário deve parar de receber pacotes dos clientes, sair para realizar todas as entregas na cidade e retornar com sua caminhonete vazia para realizar um novo lote de 10 entregas.

Os clientes (um número n de clientes ($1 < n < 100$)) possuem um número fixo de 30 pacotes cada, mas como a cidade é pequenas e eles são justos uns com os outros, somente um pacote (um contador) é entregue ao funcionário por vez. Dessa forma, após ser atendido, o cliente ajuda o funcionário e coloca o pacote na caminhonete e retorna para o fim da fila para pedir por uma nova demanda. Caso o funcionário tenha saído para entrega, o cliente deve esperar até que o funcionário retorne, para conseguir ser atendido e colocar um pacote dentro da caminhonete.

Por último, quando um funcionário colocar um pacote na caminhonete, exiba a seguinte mensagem na tela: “Cliente X solicitou entrega de um pacote”. Quando o funcionário for realizar uma entrega, ele deverá imprimir “Saindo para entregar 10 pacotes”, dormir por 2 segundos, esvaziar a caminhonete e imprimir “Retornando das entregas e pronto para atender mais clientes.”, antes de começar a receber novos pacotes.