Universidade Federal do Ceará - Campus Quixadá

Sistemas Operacionais
Prática 2 – Threads

Equipe:

Johnny Marcos Silva Soares	Matrícula: 385161
José Robertty de Freitas Costa	Matrícula: 385162

Curso: Engenharia de Computação
Professor: Carlos Igor

03/Junho/2017

- **Sumário**

- ## Introdução

A prática número dois tem como objetivo o entendimento sobre a criação, funcionamento e execução de Threads e como a exclusão mútua é aplicada para um melhor gerenciamento de recursos.

Tendo em vista esse objetivo, foi passado o jantar dos filósofos que possui as seguintes situações:

1. meditar (duração aleatória entre 0 e 2 segundos)
2. pegar palito à sua esquerda
3. pegar palito à sua direita
4. comer (duração aleatória entre 0 e 2 segundos)
5. soltar palitos.

A prática trata da resolução de um problema muito famoso, que é o problema do jantar dos filósofos. Esse problema cria várias possibilidades erros e deadlocks. Existe no problema uma situação de região crítica que é o momento que o filósofo vai pegar os palitos para comer, obviamente um palito não pode ficar com dois filósofos. Por esse motivo é necessário um estrutura que trate da exclusão mútua do problema, neste trabalho utilizamos a estrutura mutex por ser simples e a mesma é dedicada a resolver ao problema de exclusão mútua.

Foi idealizado que cada um dos filósofos fosse uma thread (um fluxo de execução), pois além das threads serem “leves” são também de fácil comunicação. Utilizamos então a biblioteca <pthread.h> e utilizamos os métodos pthread_create e pthread_join para criar e executar as threads respectivamente. Os métodos pthread_mutex_lock e pthread_mutex_unlock foram utilizados para criar a região crítica a partir de uma variável do tipo mutex.

Implementamos algumas abstrações durante a prática. Cada palito é um inteiro que pode assumir os valores 1 ou 0. Criamos a struct *filosofo* onde ela possui atributos:

- Id: Inteiro;
- Estado: inteiro (Essa variável pode receber os valores: FOME, COMENDO, MEDITANDO);
- Palito_Esquerdo: Ponteiro para o palito da esquerda;
- Palito_Direito: Ponteiro para o palito da direita;

Para representar cada filósofo foi criado uma thread para que possamos tratar este problema com as funções de threads. Cada filósofo possui duas funções que pode executar.

Utilizamos a biblioteca <unistd.h> para utilizarmos a função Sleep(time), onde passamos o tempo e o programa naquele ponto fica parado pelo tempo que é passado em segundos.

Cada filósofo terá que executar duas funções nesse problema, são elas: comer e meditar. Quando o filósofo estive no estado FOME ele irá chamar a função comer, caso contrário ele irá chamar a função meditar. Na função meditar perguntamos se o filósofo estar no estado FOME, se ele tiver nesse estado ele não irá poder executar essa função, caso contrário ele irá executar. Nessa função damos a linha de comando: sleep(rand() %3) para que ele possa esperar de 0 a 2 segundos meditando e quando ele terminar de meditar o filósofo irá para o estado FOME. A função comer inicia perguntando se o filósofo estar com fome, se ele não estiver não poderemos executar o restante da função. O restante da

função consiste em perguntar se o palito da esquerda do filósofo é zero (está disponível) caso não esteja ele não poderá comer, caso esteja ele irá colocar um no palito da direita (deixa o palito indisponível para os outros filósofos). O processo que fizemos para o palito da esquerda será repetido para o filho da direita e caso esteja tudo disponível o filósofo poderá comer livremente e chamará o comando sleep(rand() % 3) para esperar de 0 a 2 segundos, ao terminar iremos dizer que o filósofo terminou de comer e colocar os palitos como disponíveis.

Este trabalho foi basicamente dividido em três implementações assim como o problema nos sugeriu. Na primeira implementação não consideramos a existência da região crítica e simplesmente perguntamos se os palitos estão aptos a serem usados. Na segunda consideramos a região crítica a função comer, dessa forma iremos serializar a função comer, fazendo com que somente um filósofo coma por vez. Na terceira consideramos como região crítica a hora em que cada filósofo vai pegar o palito, ou seja, teremos duas regiões críticas uma para o palito da esquerda e um para o palito da direita, nessa implementação permite que dois filósofos que não estão vizinhos possam comer ao mesmo tempo.

- ## Desenvolvimento

  O desenvolvimento foi dividido em três partes:

  1. Parte 1, no qual não foi usado nenhum tipo de exclusão mútua para gerenciar o acesso das threads.
  2. Parte 2, a gerência foi feita usando Mutex, entretanto todos os recursos eram bloqueados enquanto uma thread estivesse usando, com isso apenas um filósofo poderia comer em um certo tempo.
  3. Parte 3, se baseia no princípio usado na Parte 2, entretanto, é possível que dois filósofos possam comer juntos.

  Foi utilizado funções específicas para a execução das threads, sendo elas:

  - **meditar(filosofo *f)**, que recebe como parâmetro um filósofo, sendo ele uma *struct*. Essa função irá fazer o filósofo meditar, caso ele não esteja com fome, e após meditar o estado do filósofo vai ser de FOME.
  - **comer(filosofo *f),** quando o filósofo está com fome essa função é chamada e ele poderá pegar o palito esquerdo e o direito, e após isso comer.
  - **gerenciar(void *arg)**, essa função irá fazer a gerência dos filósofos, a partir do estado uma função específica será chamada.

  Também foi usado função já definidas da biblioteca *pthread.h*, sendo elas:

  - **pthread_create()**, que criará uma thread chamando uma certa função
  - **pthread_join()**, essa função faça com que tenhamos a garantia de que em uma execução do nosso programa a thread será executada;

Parte 1

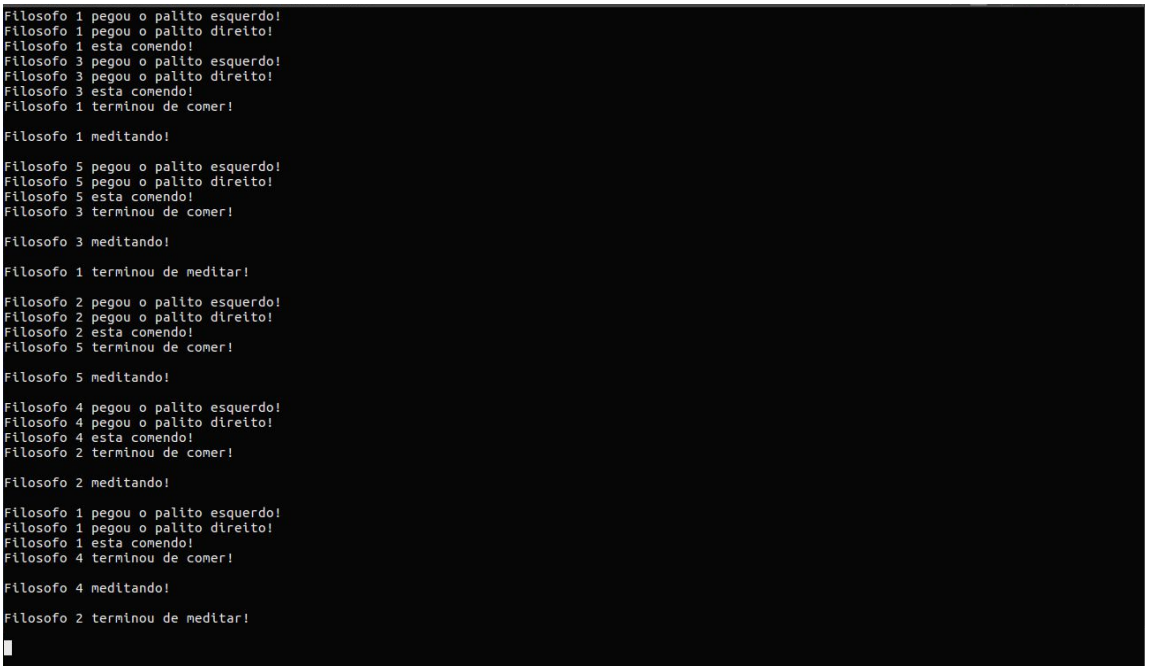
Na parte 1 como não possui nenhum tipo de gerenciamento dos recursos usados, então todas as threads irão acessar os recursos quase simultaneamente. Quando a função *gerenciar()* for chamada todas as threads serão executadas em paralelo, todas elas tentando pegar os palitos, quando o estado for de FOME.

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

#define COMENDO 1
#define MEDITANDO 2
#define FOME 3

typedef struct{
        int id;
        int estado;
        int *palito_esquerdo;
        int *palito_direito;
}filosofo;

void meditar(filosofo *f){
        if(f->estado == FOME) return;
        f->estado = MEDITANDO;
        printf("Filosofo %d meditando!\n\n",f->id);
        sleep(rand() % 3);
        printf("Filosofo %d terminou de meditar!\n\n",f->id);
        f->estado = FOME;
}

void comer(filosofo *f){
        if(f->estado != FOME) return;
        if(*f->palito_esquerdo == 0 && *f->palito_direito == 0){
                *f->palito_esquerdo = 1;
                printf("Filosofo %d pegou o palito esquerdo!\n",f->id);
                *f->palito_direito = 1;
                printf("Filosofo %d pegou o palito direito!\n",f->id);
                f->estado = COMENDO;
                printf("Filosofo %d esta comendo!\n",f->id);
                sleep(rand() % 3);
                printf("Filosofo %d terminou de comer!\n\n",f->id);
                *f->palito_esquerdo = 0;
                *f->palito_direito = 0;
        }
}

void *gerenciar(void *arg){
        filosofo *f = (filosofo *)(arg);
        while(1){
                comer(f);
                meditar(f);
        }
}


int main(){
        int p1 = 0, p2 = 0, p3 = 0, p4 = 0, p5 = 0;
        filosofo f1, f2, f3, f4, f5;
        pthread_t filosofo1, filosofo2, filosofo3, filosofo4, filosofo5;
        //configurando o filosofo 1
        f1.id = 1;
        f1.estado = FOME;
        f1.palito_direito = &p1;
        f1.palito_esquerdo = &p2;

        //configurando o filosofo 2
        f2.id = 2;
        f2.estado = FOME;
        f2.palito_direito = &p2;
        f2.palito_esquerdo = &p3;

        //configurando o filosofo 3
        f3.id = 3;
        f3.estado = FOME;
        f3.palito_direito = &p3;
        f3.palito_esquerdo = &p4;

        //configurando o filosofo 4
        f4.id = 4;
        f4.estado = FOME;
        f4.palito_direito = &p4;
        f4.palito_esquerdo = &p5;
```

```
        //configurando o filosofo 5
        f5.id = 5;
        f5.estado = FOME;
        f5.palito_direito = &p5;
        f5.palito_esquerdo = &p1;


        pthread_create(&filosofo1, NULL, gerenciar, &f1);
        pthread_create(&filosofo2, NULL, gerenciar, &f2);
        pthread_create(&filosofo3, NULL, gerenciar, &f3);
        pthread_create(&filosofo4, NULL, gerenciar, &f4);
        pthread_create(&filosofo5, NULL, gerenciar, &f5);

        pthread_join(filosofo1, NULL);
        pthread_join(filosofo2, NULL);
        pthread_join(filosofo3, NULL);
        pthread_join(filosofo4, NULL);
        pthread_join(filosofo5, NULL);
        return 0;
}
```

- **Resultado**

O resultado foi que todas as threads tentaram pegar os palitos, com isso foi possível perceber que em alguns pontos (principalmente no início) ocorreu que mais de dois filósofo comer ao mesmo tempo.



Parte 2

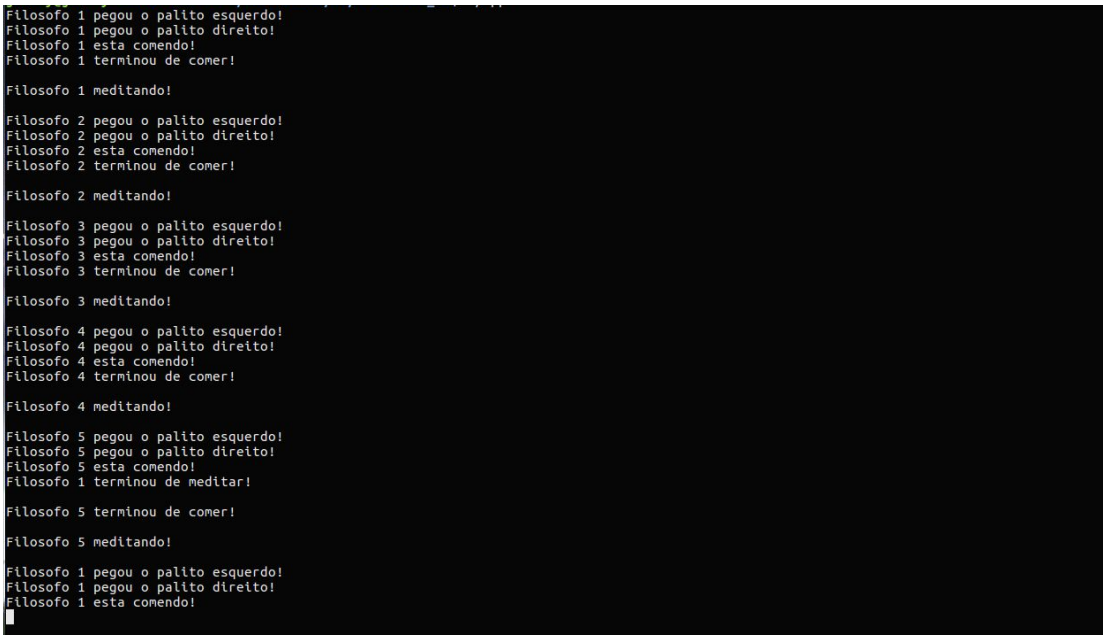
Na segunda parte com o uso do MUTEX na função *comer()* apenas uma thread poderá acessar a função por vez, pois quando uma thread mandar o status lock para o Mutex, as outras threads não poderão chamar essa função. Portanto, apenas um filósofo consegue comer por vez.

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

#define COMENDO 1
#define MEDITANDO 2
#define FOME 3

pthread_mutex_t Comer;
typedef struct{
        int id;
        int estado;
        int *palito_esquerdo;
        int *palito_direito;
}filosofo;

void meditar(filosofo *f){
        if(f->estado == FOME) return;
        f->estado = MEDITANDO;
        printf("Filosofo %d meditando!\n\n",f->id);
        sleep(rand() % 3);
        printf("Filosofo %d terminou de meditar!\n\n",f->id);
        f->estado = FOME;
}

void comer(filosofo *f){
        if(f->estado != FOME) return;
        if(*f->palito_esquerdo == 0 && *f->palito_direito == 0){
                *f->palito_esquerdo = 1;
                printf("Filosofo %d pegou o palito esquerdo!\n",f->id);
                *f->palito_direito = 1;
                printf("Filosofo %d pegou o palito direito!\n",f->id);
                f->estado = COMENDO;
                printf("Filosofo %d esta comendo!\n",f->id);
                sleep(rand() % 3);
                printf("Filosofo %d terminou de comer!\n\n",f->id);
                *f->palito_esquerdo = 0;
                *f->palito_direito = 0;
        }
}

void *gerenciar(void *arg){
        filosofo *f = (filosofo *)(arg);
        while(1){
                pthread_mutex_lock(&Comer);
                comer(f);
                pthread_mutex_unlock(&Comer);
                meditar(f);
        }
}


int main(){
        int p1 = 0, p2 = 0, p3 = 0, p4 = 0, p5 = 0;
        filosofo f1, f2, f3, f4, f5;
        pthread_t filosofo1, filosofo2, filosofo3, filosofo4, filosofo5;
        //configurando o filosofo 1
        f1.id = 1;
        f1.estado = FOME;
        f1.palito_direito = &p1;
        f1.palito_esquerdo = &p2;

        //configurando o filosofo 2
        f2.id = 2;
        f2.estado = FOME;
        f2.palito_direito = &p2;
        f2.palito_esquerdo = &p3;

        //configurando o filosofo 3
        f3.id = 3;
        f3.estado = FOME;
        f3.palito_direito = &p3;
        f3.palito_esquerdo = &p4;

        //configurando o filosofo 4
        f4.id = 4;
        f4.estado = FOME;
        f4.palito_direito = &p4;
        f4.palito_esquerdo = &p5;
```

```
        //configurando o filosofo 5
        f5.id = 5;
        f5.estado = FOME;
        f5.palito_direito = &p5;
        f5.palito_esquerdo = &p1;


        pthread_create(&filosofo1, NULL, gerenciar, &f1);
        pthread_create(&filosofo2, NULL, gerenciar, &f2);
        pthread_create(&filosofo3, NULL, gerenciar, &f3);
        pthread_create(&filosofo4, NULL, gerenciar, &f4);
        pthread_create(&filosofo5, NULL, gerenciar, &f5);

        pthread_join(filosofo1, NULL);
        pthread_join(filosofo2, NULL);
        pthread_join(filosofo3, NULL);
        pthread_join(filosofo4, NULL);
        pthread_join(filosofo5, NULL);
        return 0;
}
```

- **Resultado**

Como apenas um filósofo consegue comer por vez ocorreu uma saída sequencial, no qual um filósofo pega o palito esquerdo, logo após o direito, muda seu status para comendo e depois termina de comer. Deixando os palitos livres para o próximo filósofo, que irá repetir o ciclo.



Parte 3

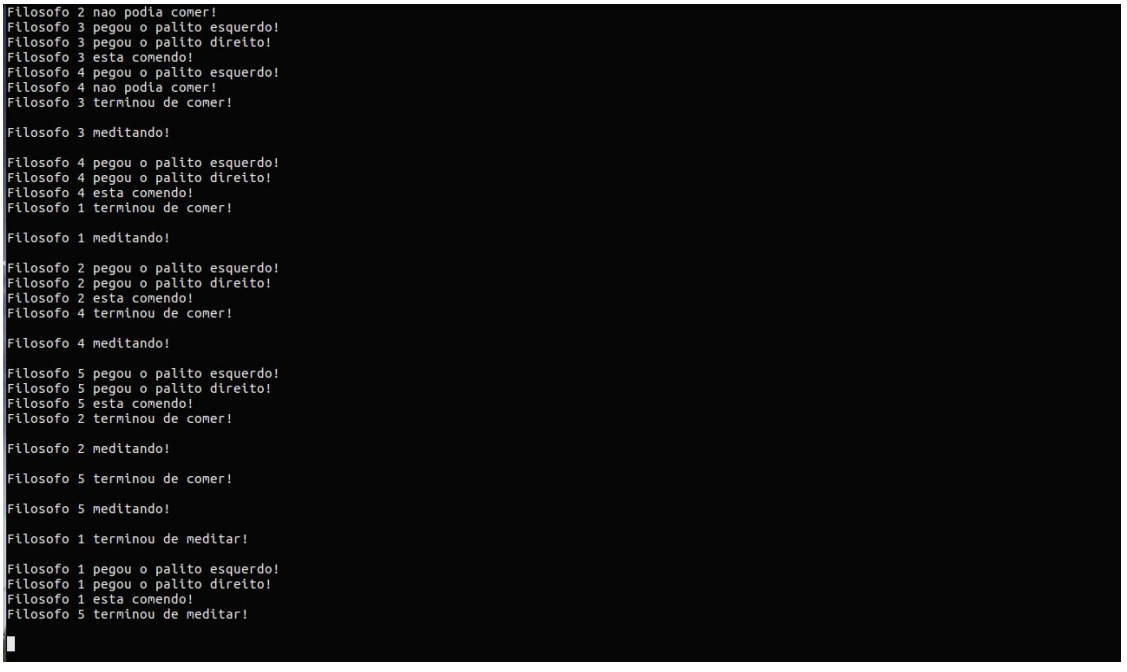
Na última parte vimos que o recurso palito foi melhor gerenciado, e evitamos o problema encontrado na parte 1. Aqui criamos uma condição diferente: caso ele já tenha o palito esquerdo e quando for acessar o palito direito ele esteja sendo utilizado então devemos “devolver o palito esquerdo a mesa”. Isso faz com que evitemos uma situação de deadlock. Aqui o as chamadas lock e unlock foram colocados estrategicamente somente na hora do filósofo “pegar o palito” tanto o da esquerda como o da direita.

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

#define COMENDO 1
#define MEDITANDO 2
#define FOME 3

pthread_mutex_t Pegar_Palitos;

typedef struct{
        int id;
        int estado;
        int *palito_esquerdo;
        int *palito_direito;
}filosofo;

void meditar(filosofo *f){
        if(f->estado == FOME) return;
        f->estado = MEDITANDO;
        printf("Filosofo %d meditando!\n\n",f->id);
        sleep(rand() % 3);
        printf("Filosofo %d terminou de meditar!\n\n",f->id);
        f->estado = FOME;
}

void comer(filosofo *f){
        if(f->estado != FOME) return;
        if(*f->palito_esquerdo == 0 && *f->palito_direito == 0){
                pthread_mutex_lock(&Pegar_Palitos);
                *f->palito_esquerdo = 1;
                pthread_mutex_unlock(&Pegar_Palitos);
                printf("Filosofo %d pegou o palito esquerdo!\n",f->id);
                if(*f->palito_direito == 0){
                        pthread_mutex_lock(&Pegar_Palitos);
                        *f->palito_direito = 1;
                        pthread_mutex_unlock(&Pegar_Palitos);
                        printf("Filosofo %d pegou o palito direito!\n",f->id);
                        f->estado = COMENDO;

                         printf("Filosofo %d esta comendo!\n",f->id);
                         sleep(rand() % 3);
                         printf("Filosofo %d terminou de comer!\n\n",f->id);
                         *f->palito_esquerdo = 0;
                         *f->palito_direito = 0;
                }else{
                         *f->palito_esquerdo = 0;
                         printf("Filosofo %d nao podia comer!\n",f->id);
                }
        }
}

void *gerenciar(void *arg){
        filosofo *f = (filosofo *)(arg);
        while(1){
                comer(f);
                meditar(f);
        }
}


int main(){
        int p1 = 0, p2 = 0, p3 = 0, p4 = 0, p5 = 0;
        filosofo f1, f2, f3, f4, f5;
        pthread_t filosofo1, filosofo2, filosofo3, filosofo4, filosofo5;
        //configurando o filosofo 1
        f1.id = 1;
        f1.estado = FOME;
        f1.palito_direito = &p1;
        f1.palito_esquerdo = &p2;

        //configurando o filosofo 2
        f2.id = 2;
        f2.estado = FOME;
        f2.palito_direito = &p2;
        f2.palito_esquerdo = &p3;
```

```
        //configurando o filosofo 3
        f3.id = 3;
        f3.estado = FOME;
        f3.palito_direito = &p3;
        f3.palito_esquerdo = &p4;

        //configurando o filosofo 4
        f4.id = 4;
        f4.estado = FOME;
        f4.palito_direito = &p4;
        f4.palito_esquerdo = &p5;

        //configurando o filosofo 5
        f5.id = 5;
        f5.estado = FOME;
        f5.palito_direito = &p5;
        f5.palito_esquerdo = &p1;


        pthread_create(&filosofo1, NULL, gerenciar, &f1);
        pthread_create(&filosofo2, NULL, gerenciar, &f2);
        pthread_create(&filosofo3, NULL, gerenciar, &f3);
        pthread_create(&filosofo4, NULL, gerenciar, &f4);
        pthread_create(&filosofo5, NULL, gerenciar, &f5);

        pthread_join(filosofo1, NULL);
        pthread_join(filosofo2, NULL);
        pthread_join(filosofo3, NULL);
        pthread_join(filosofo4, NULL);
        pthread_join(filosofo5, NULL);
        return 0;
}
```

- **Resultado**

Como resultado de saída no terminal dois filósofos conseguiram comer ao mesmo tempo, enquanto os outros estavam meditando, após o termino da função *comer()* outro filósofo consegue pegar os palitos e comer, sendo que outro já está comendo.



- ## Conclusão

Diante dos resultados das implementações, podemos supor que somente a implementação 3 é uma ótima solução para o problema do jantar dos filósofos. Já que a implementação 1 teve o problema de em alguns momentos dois filósofos comerem ao mesmo tempo com um palito em comum, isso caracteriza uma situação inconsistente ao problema. A implementação 2 força a somente um filósofo comer de cada vez, essa situação faz com que a mesma comparada com a implementação se torne ruim.

Este trabalho fez com que compreendermos como as threads funcionam realmente e quais os problemas que elas podem gerar. A resolução problema do jantar dos filósofos empregada a utilização de threads e mutex fez com que esse entendimento do conteúdo fosse facilitado.

- **Referências**

Carissimi, A. S. **Sistemas Operacionais**. 4. ed. Bookman, 2010.
Posix Threads Programming (LLNL) (http://www.llnl.gov/computing/tutorials/pthreads
Programming Posix Threads (http://www.humanfactor.com/pthreads/pthreadlinks.html)
http://en.wikipedia.org/wiki/Dining_philosophers_problem
http://www.cs.utk.edu/~plank/plank/classes/cs560/560/notes/Dphil/lecture.html