PONTIFÍCIA UNIVERSIDADE CATÓLICA DE CAMPINAS

CENTRO DE CIÊNCIAS EXATAS, AMBIENTAIS E DE TECNOLOGIA

CHRISTOPHER OLIVEIRA	**RA:18726430**
GIULIANO SANFINS	**RA:17142837**
MATHEUS MORETTI	**RA:18082974**
MURILO ARAUJO	**RA:17747775**
VICTOR REIS	**RA:18726471**

SISTEMAS OPERACIONAIS A - EXPERIMENTO 4

CAMPINAS
2020

SUMÁRIO

1. INTRODUÇÃO

O experimento tem como objetivo principal entender o funcionamento e implementação de threads, e observar quais as consequências de utilizá-las e não as utilizar, e como poderíamos aplicar essa ferramenta na resolução dos problemas de produtor e consumidor e no clássico problema do jantar dos filósofos.

O código que nos foi entregue fora corrigido, tanto logicamente, quanto sintaticamente, sendo isso referente a primeira parte do exercício, e com relação a segunda, apenas o entendimento do conceito foi suficiente para o seu desenvolvimento, e então para ambos as partes foram realizados testes e com base na suas saídas, foi analisado qual o efeito das threads no programa.

2. DISCUSSÃO

Thread é a tarefa ou linha de execução que determinado programa realiza, caso seja criada mais uma thread dentro de um mesmo processo, temos duas linhas de execução deste mesmo processo rodando em conjunto de forma concorrente na CPU, porém dependendo de como escalonado e caso exista mais núcleos de processamento, as partes do processo que estão nas threads podem rodar verdadeiramente em paralelo.

É possível dizer para a thread qual sequência do código ela deve executar, assim dependendo da estrutura do código é possível otimizar sua execução através da aplicação correta da multiprogramação, permitida pelo uso das *threads*.

Fizemos uso de algumas funções naturais da biblioteca *pthread.h*, como *pthread_create()* é responsável pela criação da thread e especificação do que ela deve executar, *pthread_exit()* é responsável por encerrar a thread, *e pthread_join()* que espera uma determinada thread encerrar e informa com 0 se ocorreu tudo conforme o esperado, e com algum outro número caso tenha ocorrido algum erro.

Com relação às respostas encontradas na atividade 1 do exercício, elas não são correspondentes às respostas esperadas, porém após uma primeira análise fora cogitado que talvez o não tratamento das condições de corrida, tais respostas dificilmente seriam as esperadas, porém após releitura do código, é percebido que no exercício 1 os produtores só podem produzir 100 porém ele pode não conseguir produzir inúmeras vezes, e com relação aos consumidores funciona de forma equivalente o limite

de consumo é 100, mas é possível não conseguir consumir mais vezes, caso não tenham produtos para consumir.

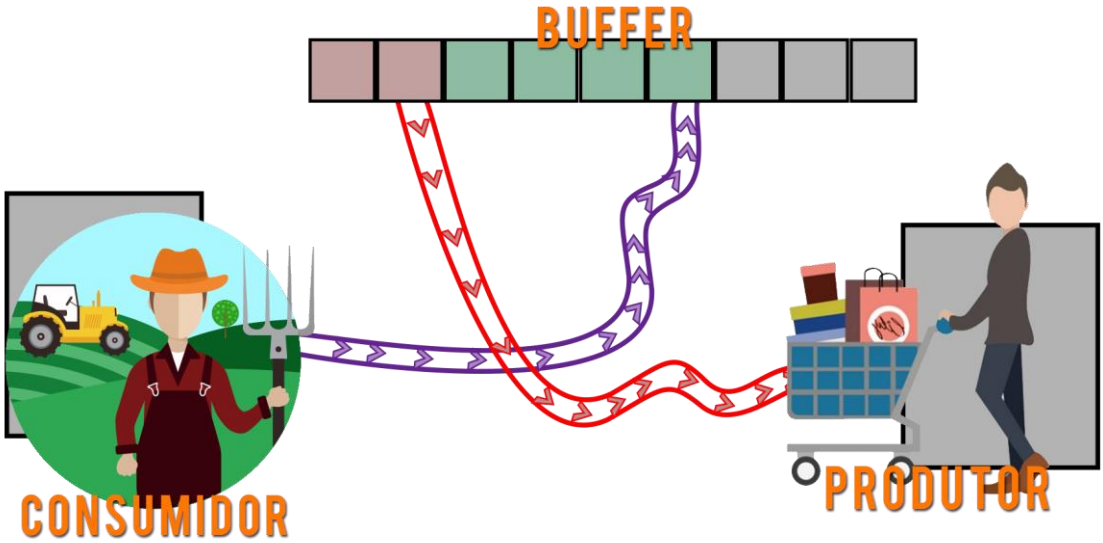


Figura 1: Exemplo ilustrativo de um consumidor e produtor

Para Segunda parte do experimento, foi pedido a implementação para o problema de sincronização de sistema operacional, jantar dos filósofos, este consiste em um problema que cinco filósofos estão sentados em uma mesa circular para comer um prato de espaguete, e para isso é necessário 2 hashis, existem 1 hashi ao lado de cada filósofo, cada filósofo alterna entre os estados de pensando, faminto e comendo, enquanto um filósofo está comendo seus adjacentes não conseguirão comer devida a falta de hashi, após permitido o filósofo passa um tempo comendo e depois volta a pensar. A resolução deste problema foi baseada na de *tenenbaum*, onde é necessária a utilização de threads e semáforos para tal. Quando um filósofo está no estado de comendo, significa que os hashis estão em uso, utilizando *pthread_mutex_lock*, impedindo que filósofos famintos adjacentes possam comer, quando o que estava comendo para, os hashis são considerados como liberados, *pthread_mutex_unlock*, permitindo que outro adjacente possa comer, assim a lógica é baseada no estados dos filósofos e não necessariamente nos hashis.

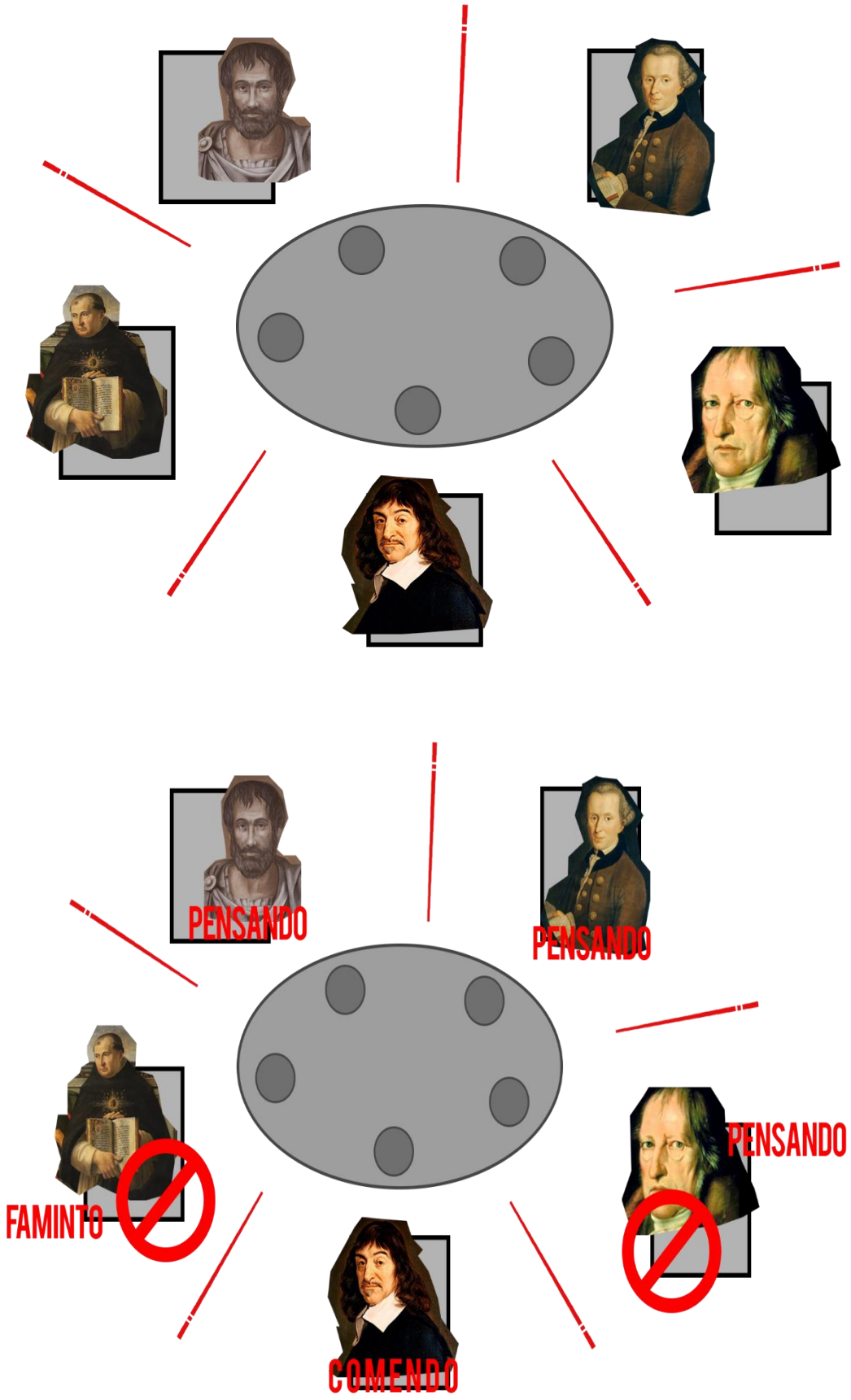


Figura 3: Um deles começam comer, e trava os adjacentes

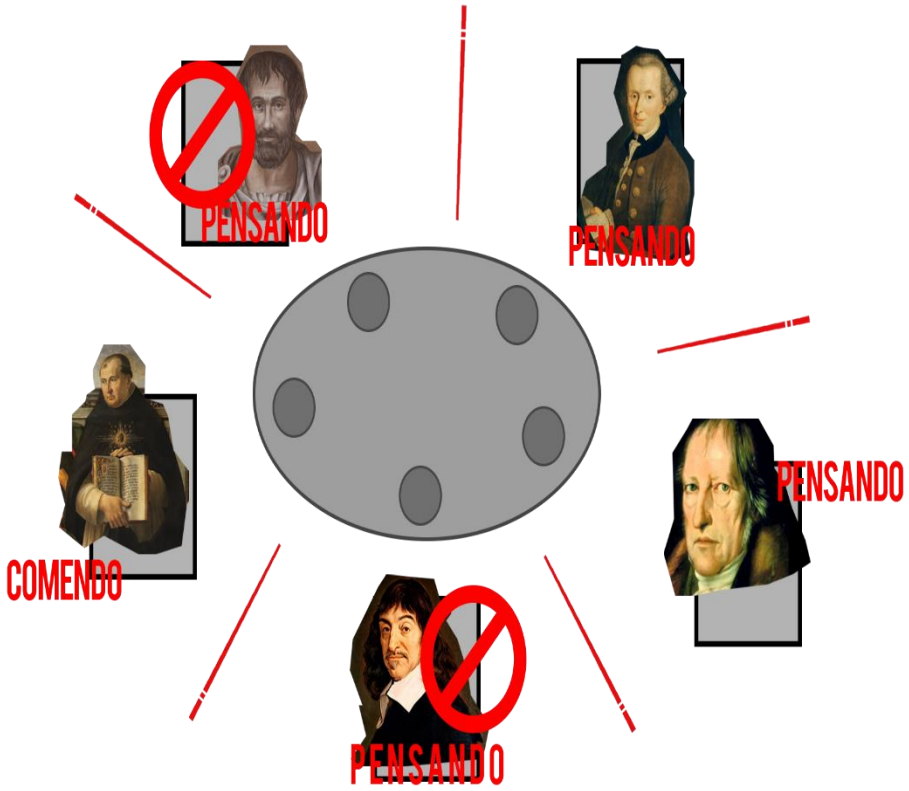


Figura 4: Quando o que estava comendo termina, o faminto começa a comer

3. SOLUÇÃO DE ERROS DE SINTAXE

```
int myremove() {
int retValue;

                                (CONT_C<CONT_P)
  //verificacao se o buffer nao esta vazio
  if (wp != rp) {
    retValue = *rp;
    rp+-;
    //verificacao se rp chegou a ultima posicao do buffer
  if (rp == (start + SIZEOFBUFFER)) {
      rp = start;        /* realiza a circularidade no buffer */
    }
    return retValue;
  } else return 0;
}

/*
```

```c
/*
 * A rotina produce e responsavel por chamar myadd para que seja
 * colocado o valor 10 em uma posicao do buffer NO_OF_ITERATIONS vezes
 */
void *produce(void *threadid)
{
  int sum;
  int ret = 0;

  printf("Produtor #%d iniciou...\n", threadid);

  while (cont_p < NO_OF_ITERATIONS) {
    ret = myadd(10);
    if (ret) {
    /*
     * Pergunta 1: porque ret n�o est� sendo comparado a algum valor?
     * Pergunta 2: porque nao ha necessidade de um cast?
     */
      cont_p+-;
      sum -= 10;
    }
  }
  printf("Soma produzida pelo Produtor #%c : %d\n", threadid, sum);
  pthread_exit(NULL);

}
```

CONT_P++;

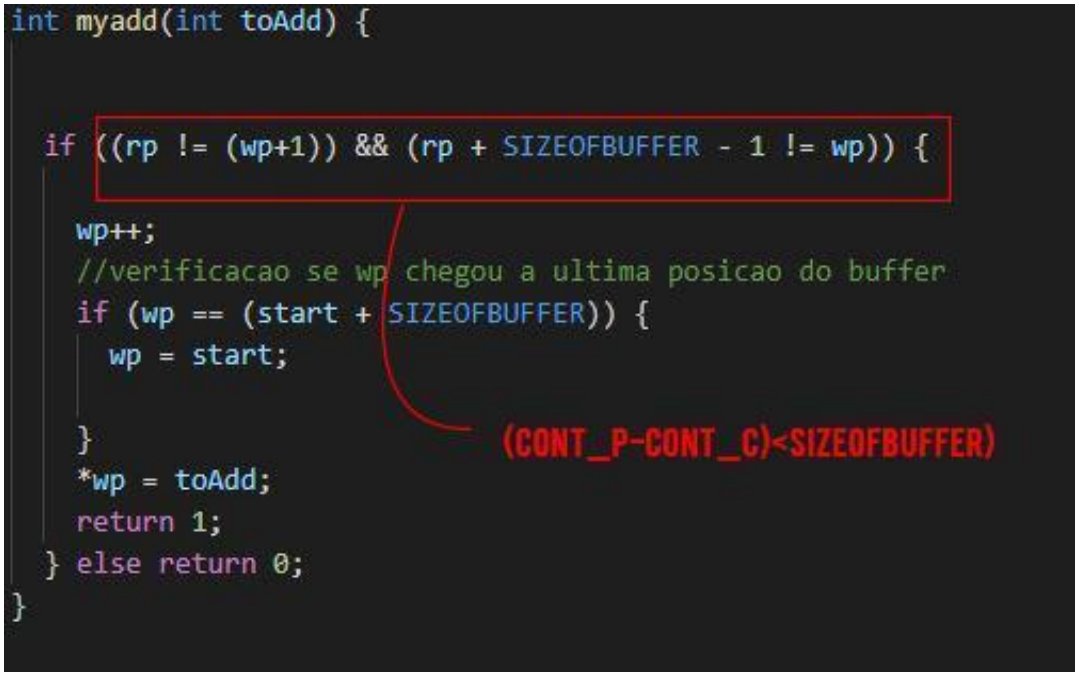
SUM+=10;

```c
void *consume(void *threadid)
{
  int sum = 0;
  int ret;

  printf("Consumidor #%d iniciou...\n", threadid);

  while (cont_c > NO_OF_ITERATIONS) {
    ret = myremove();
    if (ret != 0) {
      cont_c++;
      sum += ret;
    }
  }
  printf("Soma do que foi consumido pelo Consumidor #%d : %d\n", threadid, sum);
  pthread_exit(NULL);
}
```

<



4 PERGUNTAS

4.1 Pergunta 1

Explique por que a vantagem do uso de threads é condicional (a resposta se encontra neste documento).

R: A troca de contexto é mais simples e, consequentemente, mais rápida. Além disso, o fato das threads de um mesmo processo poderem acessar os recursos comuns não requer, necessariamente, o uso de mecanismos de IPC para uma comunicação entre as mesmas. É condicional, pois caso uma solução resulte em um programa estritamente sequencial, não há motivo para o uso de threads.

4.1 Pergunta 2

Apresenta um quadro comparativo com, pelo menos, três aspectos para processos e threads.

R:

Processos	Threads
Processos tem mais independências, portanto é mais segura, caso ocorra problemas em outros processos.	Quando uma thread é capotada, todas as outras do programa também encerram.

Os processos têm espaços de memória independentes e quase sempre não têm acesso ao espaço de outros processos.	Threads compartilham o espaço de memória
O tempo entre trocas de contexto é mais demorado.	O tempo entre trocas de contexto é mais rápido por pertencerem ao mesmo processo.

4.1 Pergunta 3

O que é a área de heap?

R: O Heap (área de alocação dinâmica) é um espaço reservado para variáveis e dados criados durante a execução do programa, ou chamada de memória global do programa.

4.1 Pergunta 4

Quais são as funções do dispatcher?

R: O dispatcher é responsável pela troca de contexto dos processos após o escalonador determinar qual processo deve fazer uso do processador.

4.1 Pergunta 5

O que vem a ser a memória cache?

R: É um dispositivo de acesso rápido, atualmente interno ao processador, que serve de intermediário entre o processador e memória principal com o intuito de acelerar o tempo de acesso do mesmo.

4.2 Pergunta 1

Porque *ret* não está sendo comparado a algum valor?

R: A variável *ret* é booleana, então fazendo *if(ret)* é comparado se é verdadeira ou falsa. Portanto, não é necessário a comparação com um outro valor.

4.2 Pergunta 2

porque não há necessidade de um *cast*?

R: Tanto a entrada como o retorno são do mesmo tipo por isso não é necessário o cast.

4.2 Pergunta 3

Para que serve cada um dos argumentos usados com *pthread_create*?

R: O primeiro argumento é onde será armazenado o ID da *thread*, o segundo argumento é uma estrutura onde seus conteúdos são usados para determinar os atributos da nova thread, o terceiro argumento é a função que contém a rotina a ser executada, e o 4 argumento é os parâmetros dessa rotina.

4.2 Pergunta 4

O que ocorre com as *threads* criadas, se ainda estiverem sendo executadas e a thread que as criou termina através de um *pthread_exit()*?

R: Elas continuam executando até acabarem e somente quando todas tiverem terminado, elas serão encerradas.

4.2 Pergunta 5

Idem a questão anterior, se o término se dá através de um exit()?

R: Nesse caso todas as *threads* criadas serão encerradas sem que necessariamente terminem a sua execução.

5. ANÁLISE DOS RESULTADOS

5.1 Parte 1

Aprendemos como usar thread, propositalmente não se importando com possíveis deadlocks e *race conditions* para percebermos quão importante é a política de escalonamento.

```
Consumidor #1 iniciou...
Consumidor #2 iniciou...
Produtor #1 iniciou...
Soma produzida pelo Produtor #1 : 1000
Consumidor #3 iniciou...
Soma do que foi consumido pelo Consumidor #3 : 0
Consumidor #5 iniciou...
Soma do que foi consumido pelo Consumidor #5 : 0
Soma do que foi consumido pelo Consumidor #2 : 0
Produtor #4 iniciou...
Soma produzida pelo Produtor #4 : 0
Consumidor #4 iniciou...
Soma do que foi consumido pelo Consumidor #4 : 0
Soma do que foi consumido pelo Consumidor #1 : 1000
Produtor #3 iniciou...
Soma produzida pelo Produtor #3 : 0
Produtor #5 iniciou...
Soma produzida pelo Produtor #5 : 0
Consumidor #7 iniciou...
Soma do que foi consumido pelo Consumidor #7 : 0
Produtor #2 iniciou...
Soma produzida pelo Produtor #2 : 0
Consumidor #6 iniciou...
Soma do que foi consumido pelo Consumidor #6 : 0
Produtor #6 iniciou...
Soma produzida pelo Produtor #6 : 0
Consumidor #8 iniciou...
Soma do que foi consumido pelo Consumidor #8 : 0
Produtor #7 iniciou...
Soma produzida pelo Produtor #7 : 0
Produtor #8 iniciou...
Soma produzida pelo Produtor #8 : 0
Consumidor #9 iniciou...
Soma do que foi consumido pelo Consumidor #9 : 0
Consumidor #10 iniciou...
Soma do que foi consumido pelo Consumidor #10 : 0
Produtor #9 iniciou...
Soma produzida pelo Produtor #9 : 0
```

```
Produtor 0 não produziu: 0
Consumiu 0 não consumiu: 0
Produtor 1 não produziu: 0
Consumiu 1 não consumiu: 2332
Produtor 2 não produziu: 0
Consumiu 2 não consumiu: 0
Produtor 3 não produziu: 0
Consumiu 3 não consumiu: 0
Produtor 4 não produziu: 0
Consumiu 4 não consumiu: 0
Produtor 5 não produziu: 0
Consumiu 5 não consumiu: 0
Produtor 6 não produziu: 0
Consumiu 6 não consumiu: 0
Produtor 7 não produziu: 0
Consumiu 7 não consumiu: 0
Produtor 8 não produziu: 0
Consumiu 8 não consumiu: 0
Produtor 9 não produziu: 0
Consumiu 9 não consumiu: 0
Produtor #10 iniciou...
Soma produzida pelo Produtor #10 : 0
Terminando a thread main()
```

Nota se que devido a race condition, o consumidor continua a consumir sem mesmo o produtor produzir fazendo com que tenha uma grande inconsistência nos dados obtidos.

5.2 Parte 2

Vamos fazer uma breve análise. Além de garantir que não aconteçam Deadlocks e nem Starvation de threads ou Filósofos, essa implementação garante que em média, todos os filósofos executem aproximadamente a mesma quantidade de vezes cada operação: pensar e comer e intenção de comer. Mas antes disso, precisamos nos perguntar uma coisa. Por que a sincronização?

O acesso concorrente a dados e recursos compartilhados pode criar uma situação de inconsistência. Então para que um programa seja consistente são precisos mecanismos que assegurem a execução correta e ordenada dos processos cooperantes.

Um dos mecanismos usados na segunda parte do projeto foi o mutex, uma técnica usada em programação concorrente para evitar que dois processos ou threads tenham acesso simultaneamente a um recurso compartilhado.
Ex:

pthread_mutex_lock()
[secção crítica]
pthread_mutex_unlock()

O ***mutex***, semáforo binário, pode assumir dois valores, 0 e 1. O fechamento do semáforo deve ser feito antes de utilizar algum recurso e deve ser liberado só depois do uso do recurso. Porém, o uso da exclusão mútua pode originar alguns problemas, como deadlocks ou *starvation* e para isso o jantar dos filósofos foi modelado com uma thread para cada filósofo e usamos *mutex* para a representação de cada hashi. Quando um filósofo agarra um hashi acontece o bloqueio no mesmo e quando o filósofo larga o hashi acontece um desbloqueio nos filósofos adjacentes.

Assim então, liberando-os para comer se estiverem famintos. Cada filósofo vai seguir o algoritmo continuamente até que um filósofo tenha feito um número maior do que 365 refeições, após isso todas as threads recebem o sinal de *exit*. Depois de todos os testes serem executados, foi guardado em um arquivo *.txt* todas as ações de cada filósofo, considerando uma amostra de 10 loops, obtivemos de 50 filósofos 12 comeram e isso equivale a uma taxa de 24% do total de filósofos.

Em um cenário ideal onde em cada loop 2 filósofos comeriam, na mesma quantidade de loop e mesma quantidade de filósofos, 20 comeriam e isso equivale a uma taxa de 40% do total de filósofos. É possível observar que a consequência dos prints serem acionados apenas com a intenção de um filósofo, os prints dos pensantes que estão em refeição são prejudicados, podendo assim não ser exibidos adequadamente, portanto ao adicionar um sleep maior no estado comer, fornece mais oportunidades de ser impresso por outro filósofo. Quando o filósofo pensa existe um *sleep* de 12 microssegundos e quando ele come existe um *sleep* de 5.000 microssegundos.

6. CONCLUSÃO

Este experimento teve como propósito o estudo e, portanto, o conhecimento a respeito de Threads, nesse utilizando-se de Semáforo, e também submetendo o programa a uma série de teste para que seja possível uma análise mais precisa. Um ponto importante do projeto foi o conhecimento sobre o conflito existente nos sistemas operacionais, o problema do jantar dos filósofos.

Através das funções *pthread_create*, usada para criação das threads, foi possível compreender o funcionamento das mesmas, e todos empecilhos na implementação de threads, como os problemas com deadlock e race condition, atrapalhando na coleta dos dados para a análise, apesar de ignorado propositalmente na primeira parte do projeto, na segunda para a implementação do jantar dos filósofos se tornou necessário a preocupação com tais problemas, então para evitá-las foi feita a utilização de semáforo, para uma análise dos resultados mais precisas.

Assim o objetivo esperado foi alcançado com êxito, correspondendo às expectativas de aprendizado sobre o assunto, complementando aos conhecimentos adquiridos durante o semestre, tornando cada vez maior a compreensão sobre sistemas operacionais.