Relatório Técnico

Grupo

Integrante	Matrícula
Matheus Saad Hayakawa	11911BCC004
Marcelo Junio de Oliveira Teixeira	11911BCC024
Rafael Borges Morais	11911BCC040

Jantar dos Filósofos

Exercício A

Conforme pedido na letra a do exercício , foi implementado uma solução para o problema do jantar dos filósofos sem impasse, para isso foi-se utilizado a criação de 5 filósofos como threads definidos como variáveis globais junto a 5 mutexs que serão os garfos utilizados pelos filósofos e a variaveis ids dos filósofos.

Na main foi implementado um for(loop) indo de 0 até 4 para iniciar as regiões críticas de memória(mutex), através da função “pthread_mutex_init” e outro for indo de 0 até 4 para inicializar as threads através da função “pthread_create” responsável por inicializar cada filósofo junto da função “jantar”.No final da main existe um “while(1)” que funciona como um “while(true)” para que o programa não se encerre.

Nas criações das funções temos a “void pega” que recebera como parâmetro um ponteiro de inteiro, no caso esses serão os ids dos filósofos, já dentro da função possuímos uma outra função “pthread_mutex_lock” responsável por bloquear o acesso aos garfos(mutex) por outros filósofos(threads), ela bloqueará o garfo da mesma posição do filósofo pois nessa solução cada filósofo , com exceção do quinto, pega o garfo de mesma posição e o de "posição+1" para que o quinto filósofo possa pegar o de posicao 0 visto que não existirá um de “posição+1” já que se trata de uma mesa circular.

Assim sendo, existe um “if” na função pega para que essa tratativa seja feita, checando se o filósofo atual é o quinto, caso não seja o garfo de posição+1 será bloqueado, depois se é feito um “else” para tratar o caso do quinto filósofo em que ele pegará o garfo de posição 0.

Depois disso, possuímos a função “larga” responsável por destravar os garfos(mutex) para os próximos filósofos, para esta ação utilizamos a função “pthread_mutex_unlock”, a função “larga” possui os mesmos passos da função “pega” explicada acima com a diferença de desbloquear em vez de bloquear os mutex.

Pseudo-Código:

```
filosofos[5];
garfos[5];
id[5];

pega(inteiro filo){
      bloqueia garfo da posição igual a de  filo
Se(filo<4){
        bloqueia garfo da posição sucessora de filo
}
Se não
        bloqueia garfo da posição 0
}


larga(inteiro filo){
     desbloqueia garfo da  posição igual a de  filo
Se(filo<4){
        desbloqueia garfo da posição sucessora de filo
}
Se não
        desbloqueia garfo da posição 0
        exibe na tela: o filósofo da posição igual a de filo acabou de comer
}


Jantar(var){
         enquanto(verdadeiro){
                  exibir na tela:Filósofo da posição de filo está pensando
                  pausa durante 5 segundos
                  pega(filo)
                  exibir na tela:Filósofo da posição de filo está comendo
                  pausa durante 5 segundos
                  larga(filo)
     }
}


int main(){
        para(int i=0,i<5,i++){
            inicializar os garfos
}
        para(int j=0,j<5,j++){
            id[j] = j
inicializa threads filosofos e utiliza funcao jantar
}
enquanto(verdadeiro){
}

return 0;
}
```

Exercício B

Conforme pedido no enunciado da letra b , foi pensado uma solução serial com bloqueio, para isso foi-se adicionado um vetor de estados na implementação já criada da letra a, cada posição desse vetor se remeterá a um dos 5 filósofos e será atribuído o valor 0 para indicar que um filósofo está

comendo e o valor 1 para que ele está pensando, logo a única diferença no código da letra b para o da letra a, é o de que na função “pega” existe um if para checar se o estado do filósofo a esquerda e o da direita é diferente de 1 assim o filósofo atual poderá comer , uma vez que essa ação só pode ser liberada se os garfos à esquerda e à direita estiverem disponíveis, justificando assim essa tratativa.

Assim sendo, também existe mais uma diferença sendo esta na função “larga” em que ao se desbloquear o mutex do garfo de posição+1(largar o último garfo) o estado é mudado para 0.

Pseudo-Código:

```
filosofos[5];
garfos[5];
estado[5];
id[5];

pega(inteiro filo){

Se (estado na posição filo for diferente de 1 e estado na posição sucessora de filo for diferente de 1){
      bloqueia garfo da posição igual a de  filo
Se(filo<4){
        bloqueia garfo da posição sucessora de filo
        estado na posição filo recebe 1
}
Se não{
        bloqueia garfo da posição 0
        estado na posição filo recebe 1
    }

 }
}


larga(inteiro filo){
     desbloqueia garfo da  posição igual a de  filo
Se(filo<4){
        desbloqueia garfo da posição sucessora de filo
        estado na posição filo recebe 0
}
Se não{
        desbloqueia garfo da posição 0
        estado na posição filo recebe 0
}

        exibe na tela: o filósofo da posição igual a de filo acabou de comer
}


Jantar(var){
         enquanto(verdadeiro){
                  exibir na tela:Filósofo da posição de filo está pensando
                  pausa durante 5 segundos
                  pega(filo)
                  exibir na tela:Filósofo da posição de filo está comendo
                  pausa durante 5 segundos
                  larga(filo)
```

```
     }
}


int main(){
        para(int i=0,i<5,i++){
            inicializar os garfos
}
        para(int j=0,j<5,j++){
            id[j] = j
inicializa threads filosofos e utiliza funcao jantar
}
enquanto(verdadeiro){
}
return 0;
}
```

Exercício C

Na implementação com o uso paralelo de recursos, temos que organizar os filósofos como sendo “threads” e os garfos como “semáforos”, assim, podemos contabilizar o garfo como um recurso que está ou não disponível, e os filósofos como utilizadores independentes desses recursos.

Por se tratar de uma mesa circular e, cada filósofo ter acesso apenas aos garfos ao seu lado, a implementação também exigiu uma ordenação, feita com o “id” dentro da estrutura “Filosofo”, e para simular a circularidade da “mesa”, o primeiro caso foi tratado separadamente, já que o índice 0, ao ser “percorrido” para a esquerda, deve seguir para o último, e não para um valor negativo.

Com esses detalhes em mente, podemos começar com a declaração do array de “pthread_t”, utilizado para a criação de threads, e a inicialização do semáforo dentro da estrutura “filosofos”(a declaração dessa estrutura foi uma decisão feita para que os semáforos e os índices pudessem ser inicializados mais facilmente, e não deve ser confundida com as threads, que por sua vez, simulam o comportamento dos filósofos em si), assim como o índice “id”.

Com toda essa estrutura, podemos ir para a rotina “filosofo”, que simula o comportamento de cada filósofo no jantar, e guia a thread em um loop infinito, onde o filósofo começa pensando por um tempo aleatório de 0 a 5 segundos, coloca em espera o “garfo”(representado por um semáforo) ao seu lado esquerdo, e o seu próprio. Caso esses estejam ocupados, ele esperará até que ambos fiquem disponíveis, e continuará para a função “comer”, que novamente esperará um valor aleatório de 0 a 5 segundos.

Após terminar de “comer”, ele libera ambos semáforos (o do índice à sua esquerda e o seu próprio) para serem utilizados por outro filósofo, seguindo para o início do loop já descrito. Após a execução do programa ser encerrada(apenas é encerrada caso o usuário pressione uma tecla), os semáforos são destruídos e o programa enfim fechará.

Para demonstrar a resolução do problema, podemos seguir o seguinte pseudo-código:

```
FILOSOFO(){
  LOOP(){
    PENSA()
    PEGA_GARFO_ESQUERDO()
    PEGA_GARFO_DIREITO()
    SE(GARFO_ESQUERDO && GARFO_DIREITO){
      COME()
    }
    LARGA_GARFO_ESQUERDO()
    LARGA_GARFO_DIREITO()
  }
}

PENSA(){
  ESPERA(0-5 segundos)
}

COME(){
  ESPERA(0-5 segundos)
}

PEGA_GARFO_ESQUERDO(x){
  ENQUANTO(OCUPADO(GARFO[x])){
    ESPERA(0.1 segundos)
  }
  OCUPA(GARFO[x])
}

PEGA_GARFO_DIREITO(x){
  ENQUANTO(OCUPADO(GARFO[x])){
    ESPERA(0.1 segundos)
  }
  OCUPA(GARFO[x])
}

LOOP(5){
  CRIA_THREAD_FILOSOFO(FILOSOFO, ID)
}

ESPERA_TECLA()
```