Relatório:

Trabalho I - Fábrica de Lâmpadas

Mauricio Konrath e Nicolas Elias

Universidade Federal de Santa Catarina

Curso de Bacharelado em Ciências da Computação

Disciplina INE5410 - Programação Concorrente

Prof. Frank Siqueira
Prof. Márcio Castro

Sobre a elaboração do trabalho: Foi utilizado o LiveShare disponível no Visual Studio Code e o Discord para fazermos chamadas, dessa forma os dois participavam e podiam editar ao mesmo tempo.

Principais dificuldades encontradas na construção do trabalho: Um dos nossos principais problemas foi entender a estruturação do código, entender o que tínhamos que fazer.

- **Estruturação do código:** Foi uma das partes mais desafiadoras, pois pegar um código pronto e entender o que está se passando e o que tem que acontecer é uma tarefa que pode ser complicada e demanda tempo para entender. Então, após algum tempo, estruturamos o código com as partes faltantes, e acreditamos que as ideias do que deve acontecer estão de certa forma corretas.

Principais pontos da implementação:

AGV:

```
39  void * agv_executa(void *arg)
40  {
41      agv_t *agv = (agv_t *) arg;
42
43      if (agv->posicionado == false) {
44          sem_wait(&max_agv_pos);
45          while (true) {
46              pthread_mutex_lock(&recicla);
47              // MUDA A CADA VEZ QUE UMA THREAD PASSA AQUI
48              if (!agv->posicionado){
49                  if (reciclagem) {
50                      reciclagem = false;
51                  } else {
52                      reciclagem = true;
53                  }
54              }
55              pthread_mutex_unlock(&recicla);
56
57              pthread_mutex_lock(&posiciona);
58              // CHAMA METODO POSICIONA
59              if (!agv->posicionado)
60                  agv_posiciona(agv, reciclagem);
61              pthread_mutex_unlock(&posiciona);
62
63              // AGUARDA ESSA CONDIÇÃO SER SATISFEITA
64              while (agv->cont_lampadas < config.capacidade_agv && !config.lampada_final) {}
65              agv_transporta(agv);
66              sem_post(&max_agv_pos);
67
68              if (config.lampada_final) {
69                  sem_post(&ultima_lampada);
70                  break;
71              }
72          }
73      }
```

No agv, a principal parte fica no loop executa que consiste em um semáforo com duas posições em que permite posicionar apenas dois agvs, e uma variável global setada para true que auxilia em configurar a reciclagem de cada agv. Temos uma verificação que consiste em checar se o agv está cheio ou se é a última lâmpada inserida, se for satisfeito ele transporta e incrementa a capacidade de posicionar um dos agvs, após isso temos um if verificando se foi a última lâmpada inserida, caso seja, ele destrava o semáforo que finaliza a esteira e os demais componentes.

Bancada:

```
22  void bancada_insere(bancada_t *self, lampada_t *lampada)
23  {
24      /* TODO: Adicionar código nesta função se necessário! */
25      /* Incrementa a quantidade de slots ocupados na bancada. */
26      while (self->slots_ocupados == config.capacidade_bancada) {}
27      self->slots[self->slots_ocupados] = *lampada;
28      self->slots_ocupados++;
29      bancada_testa(self, &self->slots[self->slots_ocupados - 1]);
30  }
31
```

```c
bool bancada_testa(bancada_t *self, lampada_t *lampada)
    /*TESTAR PARA VER SE NÃO EXISTE NENHUM DOS 4 DEFEITOS*/

    /*(1) se o vidro do bulbo da lâmpada está em perfeito estado;*/
    /*(2) se a lâmpada acende; */
    /*(3) se a rosca da lâmpada está em perfeito estado;*/
    /*(4) e se a marca e informações da lâmpada estão impressas no bulbo*/
{
    /* TODO: Adicionar código nesta função se necessário! */

    /* Simula um tempo aleatório de teste da lâmpada. NÃO REMOVER! */
    msleep(rand() % config.tempo_max_teste);

    self->total_testadas++;

    /* Testa se a lâmpada possui algum defeito. */
    if (lampada->bulbo == FALHA || lampada->luz == FALHA || lampada->marca == FALHA || lampada->rosca == FALHA) {
        lampada->resultado_teste = REPROVADA;
        self->total_reprovadas++;
        plog("[bancada] Resultado do teste da lâmpada %u: REPROVADA!\n", lampada->id);
        self->contador_lampadas++;
        return false;
    }

    self->total_aprovadas++;
    lampada->resultado_teste = APROVADA;
    plog("[bancada] Resultado do teste da lâmpada %u: APROVADA!\n", lampada->id);
    self->contador_lampadas++;
    return true;
}
```

```c
lampada_t * bancada_remove(bancada_t *self)
{
    /* TODO: Adicionar código nesta função se necessário! */

    /* Decrementa a quantidade de slots ocupados na bancada. */

    lampada_t *lampada_remove;
    // ESSA CONDIÇÃO ERA PRA GARANTIR QUE TERÁ LAMPADAS TESTADAS E ENTRARIA NO IF DO FOR
    while (self->contador_lampadas == 0) {}
    for (int i = 0; i < config.capacidade_bancada; i++) {
        if (self->slots[i].resultado_teste == APROVADA ||
            self->slots[i].resultado_teste == REPROVADA) {
            lampada_remove = &self->slots[i];
            self->slots[i] = self->slots[self->slots_ocupados - 1];
            self->slots_ocupados--;
            self->contador_lampadas--;
            return(lampada_remove);
        }
    }


    return(NULL);
}
```

Na bancada realizamos primeiro uma verificação para aguardar caso tenha mais que a quantidade da bancada, a qual que sempre vai inserir apenas a quantidade permitida pela config da bancada. No método lâmpada remove verificamos se há alguma lâmpada já testada na variável contador_lampadas, se sim permite a passagem para o laço que procura no array alguma lâmpada já testada. Ele salva ela e insere a última no seu lugar para não ter problemas com perdas de lâmpadas.

Buffer:

```c
void buffer_inicializa(buffer_t *self)
{

    pthread_mutex_init(&GERAL_BUFFER, NULL);

    /* TODO: Adicionar código nesta função se necessário! */
    self->slots_ocupados = 0;
    self->slots = (lampada_t **) malloc(sizeof(lampada_t *) * config.capacidade_buffer);
    plog("[buffer] Inicializado\n");
}

void buffer_insere(buffer_t *self, lampada_t *lampada)
{
    /* TODO: Adicionar código nesta função se necessário! */
    self->slots[self->slots_ocupados] = lampada;
    /* Incrementa a quantidade de slots ocupados no buffer. */
    self->slots_ocupados++;
}

lampada_t * buffer_remove(buffer_t *self)
{
    /* TODO: Adicionar código nesta função se necessário! */
    /* Decrementa a quantidade de slots ocupados no buffer. */
    self->slots_ocupados--;
    sem_post(&para_esteira);
    // SEMAFORO PARA PARAR ESTEIRA CASO BUFFER CHEIO
    return(self->slots[self->slots_ocupados]);
}
```

O buffer foi implementado como uma pilha, onde o último a entrar é o primeiro a sair.

Esteira:

```c
sem_t para_esteira;

sem_t ultima_lampada;

void * esteira_executa(void *arg)
{
    plog("[esteira] Inicializada\n");

    /* Recupera o argumento de entrada (esteira_t). */
    esteira_t *esteira = (esteira_t *) arg;

    double fator = (double) (ESTEIRA_VEL_MAX + 1 - config.velocidade_esteira);

    /* Produz config.total_lampadas lâmpadas. */
    for (int i = 0; i < config.quantidade_lampadas; i++) {
        // DECREMENTA UMA POSIÇÃO DO SEMAFORO ATE CAPACIDADE ESTEIRA == 0 = TRAVA
        sem_wait(&para_esteira);
        esteira_insere(esteira);
        msleep((long) ESTEIRA_VEL_TEMPO * pow(2.0, fator));
    }
    pthread_exit(NULL);
}
```

```c
void esteira_inicializa(esteira_t *self)
{

    sem_init(&para_esteira, 0, config.capacidade_buffer);
    sem_init(&ultima_lampada, 0, 0);


    self->tipo = ESTEIRA;
    self->lampadas_produzidas = 0;
    self->lampadas_consumidas = 0;
    self->lampadas = (lampada_t *) malloc(sizeof(lampada_t) * config.quantidade_lampadas);

    if (pthread_create(&self->thread, NULL, esteira_executa, (void *) self) != 0) {
        plog("[esteira] Erro ao inicializar esteira\n");
    }
}
```

```c
75  void esteira_finaliza(esteira_t *self)
76  {
77      sem_wait(&ultima_lampada);
78      pthread_join(self->thread, NULL);
79      free(self->lampadas);
80      plog("[esteira] Finalizada\n");
81  }
82
```

Na esteira, foi feito a parte de parar a esteira quando o buffer estiver cheio da seguinte forma: inicializado um semáforo com a capacidade de início do buffer, e sempre que insere algo decrementa uma posição do buffer, e no buffer sempre que removemos algo é incrementado uma posição para esteira inserir algo. No esteira finaliza temos um semáforo que tem o papel de aguardar que todo o processo da fábrica seja terminado e isso acontece quando a última lâmpada é transportada, no agv incrementamos o semáforo deixando essa linha ser executada.

Robô:

```
void robo_inicializa(robo_t *self, unsigned int id,
                     void *equipamento_esquerda, equipamento_t tipo_equipamento_esquerda,
                     void *equipamento_direita, equipamento_t tipo_equipamento_direita)
{
    /* TODO: Adicionar código nesta função se necessário! */

    // INICIALIZAÇÃO DOS MUTEXES E SEMAFOROS
    inicializa = true;
    if (inicializa) {
        inicializa = false;
        config.lampadas_no_agv = 0;
        config.lampada_final = false;

        // SEMAFORO QUE PERMITE A ENTRADA DE NO MAXIMO CAPACIDADE BUFFER
        sem_init(&buffer_empty, 0, config.capacidade_buffer);
        sem_init(&buffer_full, 0, 0);

        // SEMAFORO QUE CONSISTE NA CONDIÇÃO DE PARADA DA BANCADA
        sem_init(&bancada_empty, 0, config.capacidade_bancada);
        sem_init(&bancada_full, 0, 0);

        // EXCLUSAO MUTUA ULTILIZADA NO CONTROLE DOS BRAÇOS PARA ELES NÃO COLIDIREM
        pthread_mutex_init(&agv_coloca, NULL);
        pthread_mutex_init(&bancada_mutex, NULL);
        pthread_mutex_init(&buffer_mutex, NULL);

    }
```

```
76  void * robo_executa(void *arg)
77  {
78
79      int contador_lampadas = 0;
80      while (contador_lampadas < config.quantidade_lampadas) {
81
82          robo_t *robo = (robo_t *) arg;
83
84          lampada_t *lampada_teste = robo_pega_lampada(robo);
85
86          robo_coloca_lampada(robo, lampada_teste);
87
88          contador_lampadas++;
89      }
90
91      pthread_exit(NULL);
92  }
```

```
            break;
        case BANCADA:
            //bancada = (bancada_t *) self->equipamento_esquerda;
            // REGIAO CRITICA

            sem_wait(&bancada_full);
            pthread_mutex_lock(&bancada_mutex);
            lampada = bancada_remove((bancada_t *) self->equipamento_esquerda);
            plog("[robô %u] Lâmpada %u REMOVIDA da BANCADA\n", self->id, lampada->id);
            pthread_mutex_unlock(&bancada_mutex);
            sem_post(&bancada_empty);

            break;
        default:
            plog("[robô %u] Erro ao recuperar equipamento da esquerda\n", self->id);
    }

    return(lampada);
}
```

```
lampada_t * robo_pega_lampada(robo_t *self)
{


    lampada_t *lampada = NULL;

    esteira_t *esteira = NULL;

    switch(self->tipo_equipamento_esquerda) {
        case ESTEIRA:
            esteira = (esteira_t *) self->equipamento_esquerda;
            //lampadas produzidas - consumidar da esteira tem que ser diferente de zero pra ele pegar
            while ((esteira->lampadas_produzidas - esteira->lampadas_consumidas) == 0) {}

            lampada = esteira_remove((esteira_t *) self->equipamento_esquerda);
            plog("[robô %u] Lâmpada %u REMOVIDA da ESTEIRA\n", self->id, lampada->id);
            break;

        case BUFFER:
            //buffer = (buffer_t *) self->equipamento_esquerda;
            // REGIAO CRITICA

            sem_wait(&buffer_full);
            pthread_mutex_lock(&buffer_mutex);
            lampada = buffer_remove((buffer_t *) self->equipamento_esquerda);
            plog("[robô %u] Lâmpada %u REMOVIDA do BUFFER\n", self->id, lampada->id);
            pthread_mutex_unlock(&buffer_mutex);
            sem_post(&buffer_empty);




            break;
        case BANCADA:
            //bancada = (bancada_t *) self->equipamento_esquerda;
            // REGIAO CRITICA

            sem_wait(&bancada_full);
            pthread_mutex_lock(&bancada_mutex);
            lampada = bancada_remove((bancada_t *) self->equipamento_esquerda);
            plog("[robô %u] Lâmpada %u REMOVIDA da BANCADA\n", self->id, lampada->id);
            pthread_mutex_unlock(&bancada_mutex);
            sem_post(&bancada_empty);

            break;
```

```c
void robo_coloca_lampada(robo_t *self, lampada_t *lampada)
{
    controle_agvs_t * controle = NULL;
    agv_t *agv = NULL;

    /* TODO: Adicionar código nesta função se necessário! */

    switch(self->tipo_equipamento_direita) {
        case BUFFER:

            sem_wait(&buffer_empty);
            pthread_mutex_lock(&buffer_mutex);
            buffer_insere((buffer_t *) self->equipamento_direita, lampada);
            plog("[robô %u] Lâmpada %u INSERIDA no BUFFER\n", self->id, lampada->id);
            pthread_mutex_unlock(&buffer_mutex);
            sem_post(&buffer_full);


            break;
        case BANCADA:

            sem_wait(&bancada_empty);
            pthread_mutex_lock(&bancada_mutex);
            bancada_insere((bancada_t *) self->equipamento_direita, lampada);
            plog("[robô %u] Lâmpada %u INSERIDA na BANCADA\n", self->id, lampada->id);
            pthread_mutex_unlock(&bancada_mutex);
            sem_post(&bancada_full);

            break;
        case AGVS:

            controle = (controle_agvs_t *) self->equipamento_direita;
            if (lampada->resultado_teste == DESCONHECIDO) {
                /* Devolve lâmpada para bancada de teste */
                bancada_insere((bancada_t *) self->equipamento_esquerda, lampada);
                plog("[robô %u] Lâmpada %u DEVOLVIDA para a BANCADA\n", self->id, lampada->id);
                break;
            } else if (lampada->resultado_teste == APROVADA) {
                while(agv == NULL)
                    agv = controle_retorna_agv(controle, false); // AGV com lâmpadas aprovadas

            } else {                                        // Lâmpada reprovada no teste
                while(agv == NULL)
                    agv = controle_retorna_agv(controle, true);  // AGV com lâmpadas para reciclar

            }
```

```c
              // FAZER ALGUMA CONDICAO DE VERIFICACAO AQUI
              pthread_mutex_lock(&agv_coloca);
              if (config.lampadas_no_agv == config.quantidade_lampadas - 1) {
                  config.lampada_final = true;
                  plog("---------ULTIMA LAMPADA DETECTADA--------------\n");
              }
              agv_insere(agv, lampada);
              config.lampadas_no_agv++;
              plog("[robô %u] Lâmpada %u INSERIDA no AGV %u\n", self->id, lampada->id, agv->id);
              pthread_mutex_unlock(&agv_coloca);
              break;


          default:
              plog("[robô %u] Erro ao recuperar equipamento da direita\n", self->id);
      }
  }
```

```c
void robo_finaliza(robo_t *self)
{
    /* TODO: Adicionar código nesta função se necessário! */

    pthread_join(self->thread, NULL);

    /* DESTRUIÇÃO DOS MUTEXES E SEMAFOROS*/

    sem_destroy(&bancada_empty);
    sem_destroy(&bancada_full);
    sem_destroy(&buffer_empty);
    sem_destroy(&buffer_full);
    pthread_mutex_destroy(&buffer_mutex);
    pthread_mutex_destroy(&bancada_mutex);

    plog("[robô %u] Finalizado\n", self->id);
}
```

No robô que está a sincronização para “Os braços não se chocarem” implementamos da forma em que temos um produto consumidor realizado com dois semáforos (4 no caso, dois para bancada e 2 para o buffer) que fica verificando quantas vagas tem em cada um e travando o acesso quando insere/remove, no primeiro método de pegar lâmpada no caso esteira, temos um while que verifica se tem alguma lâmpada já produzida para não pegar lixo de memória, no demais é só verificação dos semáforos de sincronização. Já no case do equipamento direita, além das sincronizações citadas anteriormente, no método do insere agv, em cima dele temos uma verificação que diz se é a última lâmpada a ser inserida se for, ele seta para true uma variável que manda o agv transportar, e incrementa um semáforo na esteira para terminar o programa.

Problemas relacionados à solução entregue:

Anteriormente em alguns testes causava deadlocks por conta de um semáforo que estava fora do escopo do while true do agv, mas resolvemos colocando dentro do while true que resultou no programa funcionando corretamente, mas em alguns casos ocorre deadlocks que não conseguimos resolver.