

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_inserir(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
```

```

32  bool bancada_testa(bancada_t *self, lampada_t *lampada)
33      /*TESTAR PARA VER SE NÃO EXISTE NENHUM DOS 4 DEFEITOS*/
34
35      /*(1) se o vidro do bulbo da lâmpada está em perfeito estado;*/
36      /*(2) se a lâmpada acende; */
37      /*(3) se a rosca da lâmpada está em perfeito estado;*/
38      /*(4) e se a marca e informações da lâmpada estão impressas no bulbo*/
39  {
40      /* TODO: Adicionar código nesta função se necessário! */
41
42      /* Simula um tempo aleatório de teste da lâmpada. NÃO REMOVER! */
43      msleep(rand() % config.tempo_max_teste);
44
45      self->total_testadas++;
46
47      /* Testa se a lâmpada possui algum defeito. */
48      if (lampada->bulbo == FALHA || lampada->luz == FALHA || lampada->marca == FALHA || lampada->rosca == FALHA) {
49          lampada->resultado_teste = REPROVADA;
50          self->total_reprovadas++;
51          plog("[bancada] Resultado do teste da lâmpada %u: REPROVADA!\n", lampada->id);
52          self->contador_lampadas++;
53          return false;
54      }
55
56      self->total_aprovadas++;
57      lampada->resultado_teste = APROVADA;
58      plog("[bancada] Resultado do teste da lâmpada %u: APROVADA!\n", lampada->id);
59      self->contador_lampadas++;
60      return true;
61  }

```

```

64  lampada_t * bancada_remove(bancada_t *self)
65  {
66      /* TODO: Adicionar código nesta função se necessário! */
67
68      /* Decrementa a quantidade de slots ocupados na bancada. */
69
70      lampada_t *lampada_remove;
71      // ESSA CONDIÇÃO ERA PRA GARANTIR QUE TERÁ LAMPADAS TESTADAS E ENTRARIA NO IF DO FOR
72      while (self->contador_lampadas == 0) {}
73      for (int i = 0; i < config.capacidade_bancada; i++) {
74          if (self->slots[i].resultado_teste == APROVADA ||
75              self->slots[i].resultado_teste == REPROVADA) {
76              lampada_remove = &self->slots[i];
77              self->slots[i] = self->slots[self->slots_ocupados - 1];
78              self->slots_ocupados--;
79              self->contador_lampadas--;
80              return(lampada_remove);
81          }
82      }
83
84
85      return(NULL);
86  }
87

```

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:

```
12 void buffer_inicializa(buffer_t *self)
13 {
14
15     pthread_mutex_init(&GERAL_BUFFER, NULL);
16
17     /* TODO: Adicionar código nesta função se necessário! */
18     self->slots_ocupados = 0;
19     self->slots = (lampada_t **) malloc(sizeof(lampada_t *) * config.capacidade_buffer);
20     plog("[buffer] Inicializado\n");
21 }
22
23 void buffer_insere(buffer_t *self, lampada_t *lampada)
24 {
25     /* TODO: Adicionar código nesta função se necessário! */
26     self->slots[self->slots_ocupados] = lampada;
27     /* Incrementa a quantidade de slots ocupados no buffer. */
28     self->slots_ocupados++;
29 }
30
31 lampada_t * buffer_remove(buffer_t *self)
32 {
33     /* TODO: Adicionar código nesta função se necessário! */
34     /* Decrementa a quantidade de slots ocupados no buffer. */
35     self->slots_ocupados--;
36     sem_post(&para_esteira);
37     // SEMAFORO PARA PARAR ESTEIRA CASO BUFFER CHEIO
38     return(self->slots[self->slots_ocupados]);
39 }
40
```

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

Esteira:

```
10 sem_t para_esteira;
11
12 sem_t ultima_lampada;
13
14 void * esteira_executa(void *arg)
15 {
16     plog("[esteira] Inicializada\n");
17
18     /* Recupera o argumento de entrada (esteira_t). */
19     esteira_t *esteira = (esteira_t *) arg;
20
21     double fator = (double) (ESTEIRA_VEL_MAX + 1 - config.velocidade_esteira);
22
23     /* Produz config.total_lampadas lâmpadas. */
24     for (int i = 0; i < config.quantidade_lampadas; i++) {
25         // DECREMENTA UMA POSIÇÃO DO SEMAFORO ATE CAPACIDADE ESTEIRA == 0 = TRAVA
26         sem_wait(&para_esteira);
27         esteira_insere(esteira);
28         msleep((long) ESTEIRA_VEL_TEMPO * pow(2.0, fator));
29     }
30     pthread_exit(NULL);
31 }
```

```

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");
    }
}

```

```

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 UTILIZADA 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_pegar_lampada(robo);
85
86         robo_coloca_lampada(robo, lampada_teste);
87
88         contador_lampadas++;
89     }
90
91     pthread_exit(NULL);
92 }
```

```

133         break;
134     case BANCADA:
135         //bancada = (bancada_t *) self->equipamento_esquerda;
136         // REGIAO CRITICA
137
138         sem_wait(&bancada_full);
139         pthread_mutex_lock(&bancada_mutex);
140         lampada = bancada_remove((bancada_t *) self->equipamento_esquerda);
141         plog("[robô %u] Lâmpada %u REMOVIDA da BANCADA\n", self->id, lampada->id);
142         pthread_mutex_unlock(&bancada_mutex);
143         sem_post(&bancada_empty);
144
145         break;
146     default:
147         plog("[robô %u] Erro ao recuperar equipamento da esquerda\n", self->id);
148     }
149
150     return(lampada);
151 }

```

```

94 lampada_t * robo_pegar_lampada(robo_t *self)
95 {
96
97
98     lampada_t *lampada = NULL;
99
100     esteira_t *esteira = NULL;
101
102     switch(self->tipo_equipamento_esquerda) {
103     case ESTEIRA:
104         esteira = (esteira_t *) self->equipamento_esquerda;
105         //lampadas produzidas - consumir da esteira tem que ser diferente de zero pra ele pegar
106         while ((esteira->lampadas_produzidas - esteira->lampadas_consumidas) == 0) {}
107
108         lampada = esteira_remove((esteira_t *) self->equipamento_esquerda);
109         plog("[robô %u] Lâmpada %u REMOVIDA da ESTEIRA\n", self->id, lampada->id);
110         break;
111
112     case BUFFER:
113         //buffer = (buffer_t *) self->equipamento_esquerda;
114         // REGIAO CRITICA
115
116         sem_wait(&buffer_full);
117         pthread_mutex_lock(&buffer_mutex);
118         lampada = buffer_remove((buffer_t *) self->equipamento_esquerda);
119         plog("[robô %u] Lâmpada %u REMOVIDA do BUFFER\n", self->id, lampada->id);
120         pthread_mutex_unlock(&buffer_mutex);
121         sem_post(&buffer_empty);
122
123
124
125
126         break;
127     case BANCADA:
128         //bancada = (bancada_t *) self->equipamento_esquerda;
129         // REGIAO CRITICA
130
131         sem_wait(&bancada_full);
132         pthread_mutex_lock(&bancada_mutex);
133         lampada = bancada_remove((bancada_t *) self->equipamento_esquerda);
134         plog("[robô %u] Lâmpada %u REMOVIDA da BANCADA\n", self->id, lampada->id);
135         pthread_mutex_unlock(&bancada_mutex);
136         sem_post(&bancada_empty);
137
138         break;

```



```

147 void robo_coloca_lampada(robo_t *self, lampada_t *lampada)
148 {
149     controle_agvs_t * controle = NULL;
150     agv_t *agv = NULL;
151
152     /* TODO: Adicionar código nesta função se necessário! */
153
154     switch(self->tipo_equipamento_direita) {
155         case BUFFER:
156
157             sem_wait(&buffer_empty);
158             pthread_mutex_lock(&buffer_mutex);
159             buffer_insere((buffer_t *) self->equipamento_direita, lampada);
160             plog("[robô %u] Lâmpada %u INSERIDA no BUFFER\n", self->id, lampada->id);
161             pthread_mutex_unlock(&buffer_mutex);
162             sem_post(&buffer_full);
163
164
165             break;
166         case BANCADA:
167
168             sem_wait(&bancada_empty);
169             pthread_mutex_lock(&bancada_mutex);
170             bancada_insere((bancada_t *) self->equipamento_direita, lampada);
171             plog("[robô %u] Lâmpada %u INSERIDA na BANCADA\n", self->id, lampada->id);
172             pthread_mutex_unlock(&bancada_mutex);
173             sem_post(&bancada_full);
174
175             break;
176         case AGVS:
177
178             controle = (controle_agvs_t *) self->equipamento_direita;
179             if (lampada->resultado_teste == DESCONHECIDO) {
180                 /* Devolve lâmpada para bancada de teste */
181                 bancada_insere((bancada_t *) self->equipamento_esquerda, lampada);
182                 plog("[robô %u] Lâmpada %u DEVOLVIDA para a BANCADA\n", self->id, lampada->id);
183                 break;
184             } else if (lampada->resultado_teste == APROVADA) {
185                 while(agv == NULL)
186                     agv = controle_retorna_agv(controle, false); // AGV com lâmpadas aprovadas
187
188             } else {                                     // Lâmpada reprovada no teste
189                 while(agv == NULL)
190                     agv = controle_retorna_agv(controle, true); // AGV com lâmpadas para reciclar
191
192             }

```

```

194         // FAZER ALGUMA CONDICAO DE VERIFICACAO AQUI
195         pthread_mutex_lock(&agv_coloca);
196         if (config.lampadas_no_agv == config.quantidade_lampadas - 1) {
197             config.lampada_final = true;
198             plog("-----ULTIMA LAMPADA DETECTADA-----\n");
199         }
200         agv_insere(agv, lampada);
201         config.lampadas_no_agv++;
202         plog("[robô %u] Lâmpada %u INSERIDA no AGV %u\n", self->id, lampada->id, agv->id);
203         pthread_mutex_unlock(&agv_coloca);
204         break;
205
206
207         default:
208             plog("[robô %u] Erro ao recuperar equipamento da direita\n", self->id);
209     }
210 }
211

```

```

224 void robo_finaliza(robo_t *self)
225 {
226     /* TODO: Adicionar código nesta função se necessário! */
227
228     pthread_join(self->thread, NULL);
229
230     /* DESTRUIÇÃO DOS MUTEXES E SEMAFOROS*/
231
232     sem_destroy(&bancada_empty);
233     sem_destroy(&bancada_full);
234     sem_destroy(&buffer_empty);
235     sem_destroy(&buffer_full);
236     pthread_mutex_destroy(&buffer_mutex);
237     pthread_mutex_destroy(&bancada_mutex);
238
239     plog("[robô %u] Finalizado\n", self->id);
240 }

```

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. Em algumas tentativas o programa funciona e em outras não, provavelmente tem alguma corrida acontecendo entre as threads que não identificamos que resulta em um deadlock nos agvs.