UNIVERSIDADE FEDERAL DE SANTA CATARINA
CENTRO TECNOLÓGICO DE JOINVILLE

Exercício 5 - EMB 5632 - Sistemas Operacionais

Lucas Gabriel Bernardino - 22101935

1 - Introdução

Esse relatório busca propor uma explicação do funcionamento do meu código para o quinto exercício de sistemas operacionais. A avaliação proposta exigia que o aluno utilizasse o conceito de semáforos com produtores, consumidores e *threads* para realizar operações em uma lista, feita no exercício 5.

2 - Desenvolvimento

2.1 - Explicando o produtor

A função de produção é a responsável por inserir elementos no fim da lista. Assim como a função de consumir, ela será chamada por várias *threads* que tentaram realizar operações na mesma lista.

Figura 1 - Função responsável pela produção.

```
void *produzir(void *arg) {

  for (;;) {
    my_list *p = (my_list *)arg;
    pthread_mutex_lock(mutex: &lock);
    if (elem_id == 20) {
      printf(format: "Alcançou limite de producao. Thread [%d] produtora saindo.\n",
             p->thread_id);
      canFinish = true;
      pthread_mutex_unlock(mutex: &lock);
      return NULL;
    }
    int meu_elemento = elem_id;
    elem_id++;
    pthread_mutex_unlock(mutex: &lock);
    sem_wait(sem: &sem_prod);
    printf(format: "[%d]Produzindo %d ...\n", p->thread_id, meu_elemento);
    usleep(useconds: 500000);
    p->value = meu_elemento;
    insert_last(arg: p);
    printf(format: "[%d]Produzido\n", p->thread_id);
    sem_post(sem: &sem_cons);
  }
}
```

Fonte: Elaborado pelo autor (2024).

A função inicialmente utiliza o mutex para verificar se *elem_id* é igual a 20, ou seja, se todos os elementos já foram produzidos. Se isso for verdade, então ela altera a variável *canFinish* para true, a qual será utilizada pela função de consumir para terminar a *thread*. A operação de inserir no final da lista está sendo protegida pelos semáforos, com as funções *sem_wait* e *sem_post.*

2.2 - Explicando o consumidor

A função responsável por consumir os elementos da lista precisou ser refeita várias vezes. Inicialmente, não estava sendo utilizada uma *flag* para verificar se a *thread* já poderia ser encerrada, de modo que se estava apenas verificando se o *elem_id* fosse igual a 20. Porém, embora o código não apresentasse nenhum erro ao utilizar a *-fsanitize=thread*, ele não estava funcionando de modo correto, visto que algumas

vezes o consumo era correto e outras vezes uma *thread* não era encerrada corretamente ou não eram consumidos elementos corretos.

Figura 2 - Função responsável por consumir.

```
2  void *consumir(void *arg) {
3    for (;;) {
4      my_list *p = (my_list *)arg;
5      pthread_mutex_lock(mutex: &lock);
6      if (canFinish) {
7        printf(format: "Alcançou limite de producao. Thread [%d] consumidora saindo.\n",
8               p->thread_id);
9        count_consm++;
10       pthread_mutex_unlock(mutex: &lock);
11       break;
12     }
13     pthread_mutex_unlock(mutex: &lock);
14     printf(format: "[%d]Consumindo ... \n", p->thread_id);
15     sem_wait(sem: &sem_cons);
16     pthread_mutex_lock(mutex: &lock);
17     if (lastThread) {
18       printf(format: "Alcançou limite de producao. Thread [%d] consumidora saindo.\n",
19              p->thread_id);
20       return NULL;
21     }
22     pthread_mutex_unlock(mutex: &lock);
23     usleep(useconds: 333000);
24     remove_first(arg: p);
25     printf(format: "[%d]Consumido %f\n", p->thread_id, p->value);
26     sem_post(sem: &sem_prod);
27   }
28   pthread_mutex_lock(mutex: &lock);
29   if (count_consm == (CONSUMIDORAS - 1)) {
30     sem_post(sem: &sem_cons);
31     lastThread = true;
32   }
33   pthread_mutex_unlock(mutex: &lock);
34   return NULL;
35 }
```

Fonte: Elaborado pelo autor (2024).

Conforme pode ser visto pela figura, a lógica da função ficou relativamente complexa, e diferente do que foi inicialmente pensado. Inicialmente, a função verifica se a *flag* alterada pela função produzir é verdadeira, e caso seja essa *thread* se encerra. O código funcionava corretamente somente com esse *if*, porém ao utilizar três consumidores ao invés de dois, de modo que o número de consumidores agora era diferente do número de produtores, o código passou a apresentar falha em uma das *threads*, que acabava nunca sendo encerrada. A razão pela qual acredito que essa *thread* não se encerrava era porque ela ficava esperando um sinal da variável *sem_cons*, porém isso não acontecia nunca e então o programa ficava em um loop infinito esperando a *thread* se encerrar. Assim, foi necessário utilizar uma lógica especial para esse caso de dois produtores e três consumidores, na qual existe um contador que é incrementado cada vez que uma thread consumidora se encerra, e quando a condição mostrada na linha 29 é verdadeira, acontece um sinal na variável *sem_cons* e uma mudança na *flag* lastThread, permitindo assim que a última thread possa ser encerrada corretamente.

Mais detalhes da compilação e execução do programa podem ser vistos no vídeo que foi enviado junto com esse relatório