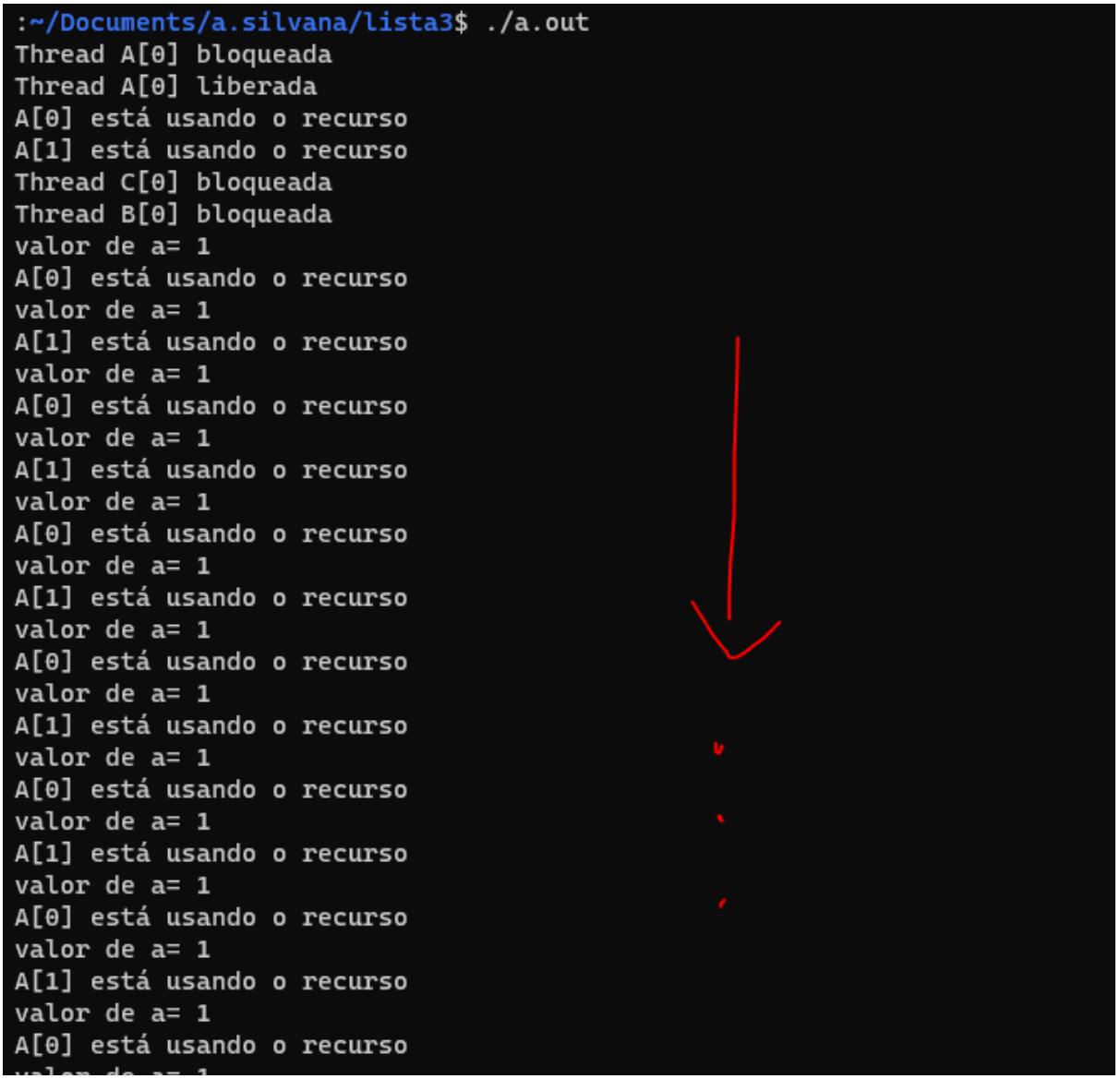
Assim como nas outras listas, os arquivos estão disponíveis no github

Questão 1

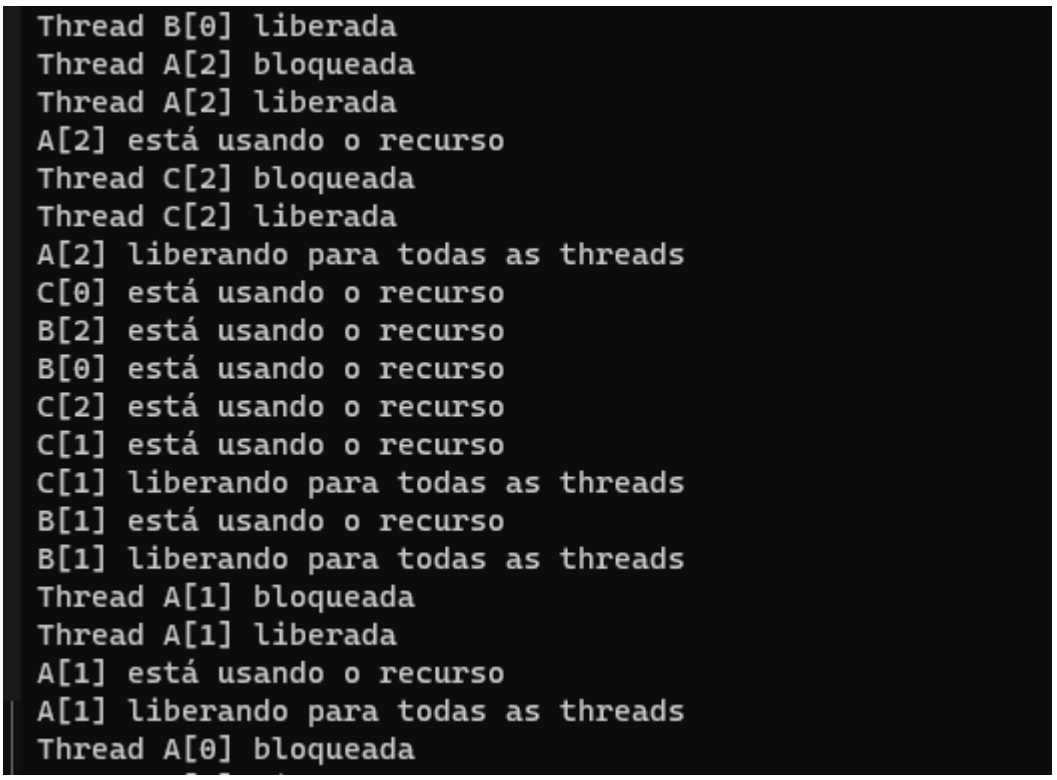
```
12  void *A (void *i) {
 1     long int id = (long int) i;
 2     while(1) {
 3        sem_wait(&emA);
 4        a++;
 5        if(a==1) {
 6           printf("Thread A[%ld] bloqueada\n",id);
 7              sem_wait(&rec);
 8           printf("Thread A[%ld] liberada\n",id);
 9        }
10        sem_post(&emA);
11        printf("A[%ld] está usando o recurso\n",id);
12        //SC: usa o recurso
13        sleep(1);
14        sem_wait(&emA);
15        a--;
16        printf("valor de a= %d\n",a);
17        if(a==0){
18           sem_post(&rec);
19           printf("A[%ld] liberando para todas as threads\n",id);
20        }
21        sem_post(&emA);
22        //sleep(1);
23     }
24  }
```

(a) Sim. As threads de mesmo tipo não podem acessar simultaneamente as partes dentro dos semáforos, (linha 3 a 10 e 14 a 21). Isso indica também que, quando uma thread libera para outras threads, todas elas estão necessariamente na linha 3 e a próxima thread irá encontrar sem_wait(&rec) e ficar travada.

(b) Sim. As threads podem esperar tempo indeterminado dependendo do recurso utilizado na memória compartilhada. Colocar um sleep como indicado na linha 13 ocasiona isso. Durante os 30 segundos rodados, resultou numa fileira de threads do mesmo tipo, como pode ser visto na imagem abaixo. No entanto, a fila do &rec garante oportunidades iguais a todos os tipos de threads, já que existe um thread de cada tipo em cada posição da fila. No entanto, isso só não ocorre aqui porque o “a” nunca chega a 0, como também pode ser visto na imagem. Retirar o sleep dessa posição anula a starvation.
Isso indica que um algoritmo muito pesado nesse local pode impedir as outras threads, provavelmente porque o tempo para executar o sleep é gigantesco em relação ao resto do loop, fazendo a = numero_de_As - 1, já que todas as outras threads estão dormindo.
Nesse caso numero_de_As = 2, fazendo a=1 sempre.



(c) Não haveria espaço para nenhuma thread passar do primeiro sem_wait(&rec), fazendo todas elas travarem.
(d) Apenas uma thread de cada tipo entra na fila do rec, quando a==0, ou b==0, etc. Colocar tamanho N no sinal &rec permitirá até N tipo de threads simultâneas no recurso compartilhado, como pode ser visto na imagem: 3 threads de cada tipo e &rec inicializado com 2:



Questão 2

O semáforo que serve como mutex entre os consumidores, e o semáforo que serve como mutex entre os produtores são a mesma variável &s. Para consertar isso basta criar dois tipos diferentes, e.g.: &s_prod e s_cons. A solução nem precisa disso, pois há apenas 1 de cada thread.

O semáforo que deve ser compartilhado entre esses dois tipos (produtor/cosumidor) está com uma condição desnecessária para contagem (n) e um dos sem_wait está fora do loop. O “n” pode ser removido e substituído por outro semáforo inicializado com 0. Que checa se o buffer fica vazio.

(b) Para a solução:

```
13 void * consumidor(void * arg){
12    int *id = (int *) arg;
11    int item;
10    while(1){
 9       sem_wait(&d); //&slot_cheio
 8       retira_item(&item); ;
 7       printf("C[%d] consumiu %d\n",*id,item);
 6       sleep(1);
 5    }
 4
 3    free(arg);
 2    pthread_exit(NULL);
 1 }
4
 1 void * produtor(void * arg){
 2    int *id = (int *) arg;
 3    printf("%d",*id);
 4    while(1){
 5       insere_item(*id);
 6       sem_post(&d); //&slot_cheio
 7       sleep(1);
 8    }
 9    free(arg);
10    pthread_exit(NULL);
11 }
```

O buffer é inicializado com 0 e o produtor sempre insere 1. Logo, se o consumidor retirar 0 então ocorreu um erro:
Observe que o erro nunca ocorre:

```
C[2] consumiu 1
C[2] consumiu 1
C[2] consumiu 1
C[2] consumiu 1
C[2] consumiu 1
C[2] consumiu 1
C[2] consumiu 1
C[2] consumiu 1
C[2] consumiu 1
C[2] consumiu 1
C[2] consumiu 1
C[2] consumiu 1
```

Questão 3

Observando o código, percebemos duas variáveis de contagem das threads: leit e escr. Para tentar priorizar o escritor ao leitor, a variável “escr” foi aumentada antes de bloquear o escritor:

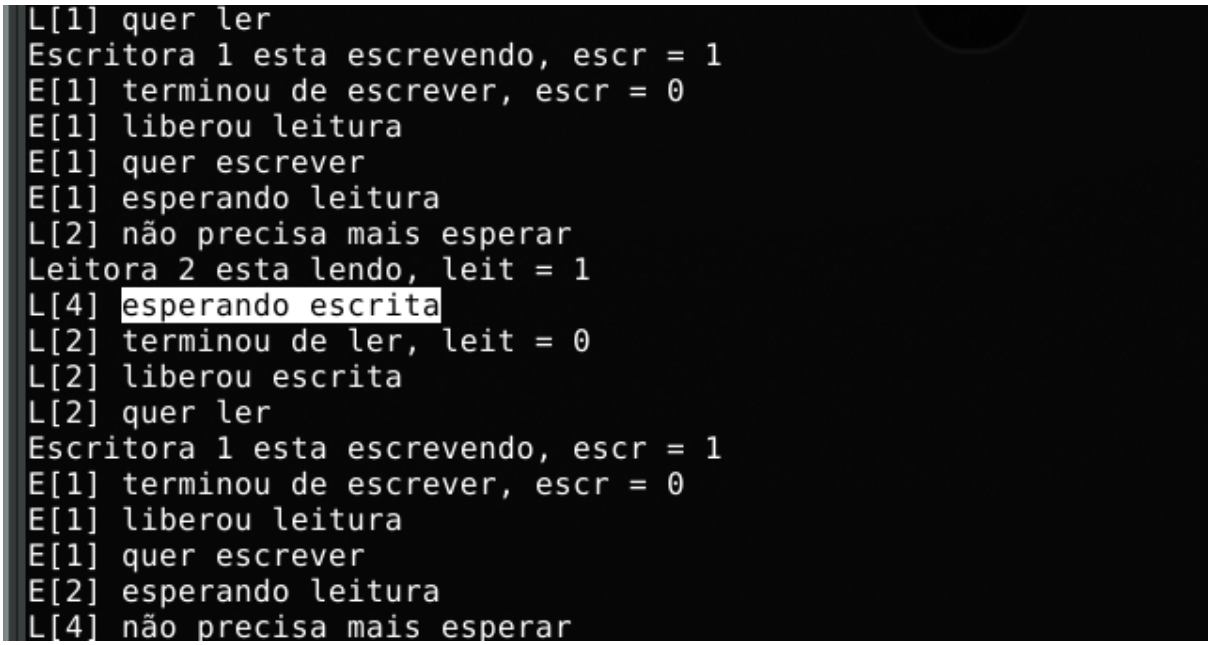
```
//entrada escrita
void InicEscr (int id) {
   printf("E[%d] quer escrever\n", id);
   sem_wait(&escr_mutex);
   escr++;// avisa aos leitores, aumenta prioridade do escritor
   if(leit>0){// espera qualquer escritor ocupado
      printf("E[%d] esperando leitura\n", id);
      sem_wait(&priority);
   }
}
```

Em comparação, o leit foi colocado depois, para apenas ser aumentado caso não haja escritor. Se um leitor cair nesta 3ª linha, a continuidade dos leitores será impedida:

```
3 //entrada leitura
2 void InicLeit (int id) {
1    printf("L[%d] quer ler\n", id);
18   sem_wait(&leit_mutex);
1    if(escr>0){ // espera escritor
2       printf("L[%d] esperando escrita\n",id);
3       sem_wait(&rec);//bloqueia a si mesmo
4       printf("L[%d] não precisa mais esperar\n",id);
5    }else{ // "else" pois último escritor já aumenta leit
6       leit++;//colocar depois perimite prioridade menor
7    }
8    sem_post(&leit_mutex);
9 }
10
```

Assim, como podemos observar na figura abaixo, quando um escritor quer escrever, qualquer leitor dá o sinal de “esperando escrita”, mesmo se não houver escritores ativos.

Observe como L[4], teve que esperar mesmo tendo apenas L[2] lendo.



O único problema é que isso impede o leitor de executar simultaneamente, como pode ser parcialmente observado na imagem. Pois basta ter 1 escritor querendo entrar que a fila do leitor é bloqueada.

Questão 4

(a) “h” é o semáforo utilizado para fila de threads em espera. Toda vez que o wait é chamado, o sem_wait(&h) é alcançado e interrompe a thread.
“x” é o semáforo usado para impedir a condição de corrida na variável aux, serve como mutex simples entre as threads.
“s” é o semáforo usado para garantir a continuidade da thread liberada, dentro da wait(). Isso excluiria, por exemplo, a necessidade de colocar o leit++ dentro do FimEscr na questão 1.

(b) Sim, em todos os meus testes é o que ficou aparente. Mas eu acredito que o mutex do &m esteja trocado, embora não tenha feito diferença dado que eu usei apenas 1 wait().

```c
void * A() {
    printf("eu ");
    sem_post(&a);
    wait();
    printf(" de chocolote ");
    notify();
    notifyAll();
    pthread_exit(NULL);
}
void * B() {
    sem_wait(&a);
    printf("gosto");
    notify();
    pthread_exit(NULL);
}
```

output:

```
eu gosto de chocolote :~/Documents/a.silvana/lista3$ ./a.out
eu gosto de chocolote :~/Documents/a.silvana/lista3$ ./a.out
eu gosto de chocolote :~/Documents/a.silvana/lista3$ ./a.out
eu gosto de chocolote :~/Documents/a.silvana/lista3$ ./a.out
eu gosto de chocolote :~/Documents/a.silvana/lista3$ ./a.out
eu gosto de chocolote :~/Documents/a.silvana/lista3$ ./a.out
eu gosto de chocolote :~/Documents/a.silvana/lista3$ ./a.out
eu gosto de chocolote :~/Documents/a.silvana/lista3$ ./a.out
eu gosto de chocolote :~/Documents/a.silvana/lista3$ ./a.out
eu gosto de chocolote :~/Documents/a.silvana/lista3$ ./a.out
eu gosto de chocolote :~/Documents/a.silvana/lista3$ ./a.out
eu gosto de chocolote :~/Documents/a.silvana/lista3$ ./a.out
eu gosto de chocolote :~/Documents/a.silvana/lista3$ ./a.out
eu gosto de chocolote :~/Documents/a.silvana/lista3$ ./a.out
eu gosto de chocolote :~/Documents/a.silvana/lista3$ ./a.out
eu gosto de chocolote :~/Documents/a.silvana/lista3$ ./a.out
```

(c) Não. O aux sempre garante que os semáforos dos “notifies” sejam executados quando existem sinais em espera. Mesmo que algum notify consiga chegar no sinal &h antes do wait(), ele estará bloqueado pelo s. Permitindo que o wait() alcance-o facilmente.