Módulo 3 - Lista de Exercícios 3 (2021/1 REMOTO)

Carlos Bravo
119136241

Outubro 2021

Questão 1

(a) Sim.

- Condições do problema: A primeira thread que começar datá a condição de $wait(rec)$. Digamos que seja uma thread do tipo A para facilitar, então $a = 1$, $b = 0$ e $c = 0$. Se uma segunda thread do tipo A entrar, $a = 2$, então poderá realizar as tarefas sem problema. Se uma thread de outro tipo tentar entrar (digamos que uma do tipo B para facilitar), $b = 1$. Nesse caso, a thread estará travada na função $wait(rec)$, pois o sinal já foi consumido pelo primeiro A. Mesma coisa pro tipo C. Quando a última thread A acabar, $a = 0$, entrando na função $post(rec)$. Assim, irá liberar a entrada do próximo tipo de thread. Satifazendo assim a condição do problema.
- Condições de corrida: As condições de corrida ocorrem quando mais de uma thread acessa a mesma variável global ao mesmo tempo. As variáveis globais são: a, b, c, emA, emB, emC e rec. As 4 últimas são semáforos e apenas são acessadas por suas funções respectivas, não ocorrendo condições de corrida. As 3 primeiras estão protegidas pelos semáforos 4, 5 e 6, correspondentemente. Todo acesso à variável a está travado pelo semáforo emA e assim por diante. então não é possível ter condições de corrida.
- Deadlock: Cada semáforo começa com 1 sinal. Toda vez que um sinal de thread (emA, emB ou emC) é consumido, i.e. é chamada a função $wait()$, logo depois é chamada a função $post()$, devolvendo o sinal consumido. Na primeira chamada que há a chamada do $wait(rec)$ não ocorrerá deadlock pois, como tem 1 sinal de início, é garantido que a primeira thread conseguirá passar do $wait()$, e no final será chamado um $post(rec)$, notificando a próxima thread.

(b) Sim. Se o consumo do recurso é demorado e possui várias threads de um mesmo tipo, o semáforo rec estará ocupado pela primeira thread que acessá-la. O semáforo só estará liberado quando a última thread de um tipo finalizar sua função antes que outra thread do mesmo tipo começar, podendo assim impedir que os outros tipos sejam executados por longos períodos de tempo.

(c) Se o semáforo não possuir sinais, qualquer thread que começar a ser executada ficará travada logo no primeiro $wait(rec)$. Assim, a aplicação entrará em deadlock.

(d) Se o semáforo possuir mais de um sinal, as condições do problema não estarão satisfeitas, porque mais de um tipo de thread poderá rodar ao mesmo tempo. Como há mais de um sinal,

se duas threads de tipo diferentes começarem, ambas passarão do $wait(rec)$.

Questão 2

(a) O erro está no fato que a condição $if(n == 0)$ não está dentro da exclusão mútua de s, gerando assim uma condição de corrida. Se ocorrer essa ordem de eventos:

Produtora produz 1 item (d possui 1 sinal) > Consumidora consome 1 item e para na linha 10 > Produtora produz 1 item (d possui 2 sinais) > Consumidora não entra na condição $if(n == 0)$ > Consumidora consome 1 item e entra na condição (Não há mais itens, mas d possui 2 sinais, então a thread não é congelada) > Consumidora consome 1 item que não existe

(b) Isso pode ser resolvido sem o uso dessas condições adicionais. Toda vez que for chamada a função *insere_item*() em sequência deverá ser registrado um sinal em d com $sem\_post(\&d)$, removendo a condição $if(n == 1)$. E toda vez que for chamada a função *retira_item*() em sequência deverá ser consumido um sinal em d com $sem\_wait(\&d)$, removendo a condição $if(n == 0)$.

Questão 3

(a)

```
// Exclusao mutua de e, l & aux
sem_t mutexE, mutexL, mutexAux;

// Bloqueia leitores enquanto ha escritores
sem_t leitorBloq;

// Garante unicidade de escritores
sem_t bufferOcup;

int e, l, aux;

void entradaLeitor(){
  sem_wait(&mutexAux);
  sem_wait(&mutexE);
  // Confere se ha escritores
  if(e > 0){
    // Se sim, bloqueia e soma 1 na fila
    aux++;
    sem_post(&mutexE);
    sem_post(&mutexAux);
    sem_wait(&leitorBloq);
  }else{
    sem_post(&mutexE);
    sem_post(&mutexAux);
  }
  // Se chegou aqui pode ler
  // Adiciona um leitor e ocupa o buffer caso seja o primeiro
```

```
  sem_wait(&mutexL);
  l++;
  if(l == 1) sem_wait(&bufferOcup);
  sem_post(&mutexL);
}

void saidaLeitor(){
  // Remove um leitor
  // Se for o ultimo libera o buffer tambem
  sem_wait(&mutexL);
  l--;
  if(l == 0) sem_post(&bufferOcup);
  sem_post(&mutexL);
}

void entradaEscritor(){
  // Adiciona um escritor
  sem_wait(&mutexE);
  e++;
  sem_post(&mutexE);

  // E ocupa o buffer
  sem_wait(&bufferOcup);
}

void saidaEscritor(){
  // Libera o buffer
  sem_post(&bufferOcup);

  sem_wait(&mutexAux);
  sem_wait(&mutexE);
  // Remove um escritor
  e--;
  // Se for o ultimo escritor libera cada leitor bloqueado
  if(e == 0 && aux > 0){
    while(aux--) sem_post(&leitorBloq);
    aux = 0;
  }
  sem_post(&mutexE);
  sem_post(&mutexAux);
}

void *leitor(void *args){
  entradaLeitor();
  // Realiza leitura
  saidaLeitor();
}
```

```
void *escritor(void *args){
  entradaEscritor();
  // Realiza escrita
  saidaEscritor();
}
```

(b) O código pode ser separado em 4 partes: *entradaLeitor*, *saidaLeitor*, *entradaEscritor* e *saidaEscritor*. Ele funciona da seguinte forma:

- *entradaLeitor*: Confere se há escritores. Se sim, a variável *aux*, que é responsável por contabilizar os leitores bloqueados pela presença de escritores, será somada em 1 e o leitor correspondente será bloqueado. Se não houver escritores ou for desbloqueada, irá adicionar 1 a quantidade de leitores. Se for o primeiro leitor, irá bloquear o buffer para impedir a entrada de outros escritores enquanto estiver sendo feita a leitura.
- *saidaLeitor*: Subtrai 1 da quantidade de leitores. Se for o último leitor, libera o buffer para entrada de futuros leitores ou escritores em espera.
- *entradaEscritor*: Soma 1 a quantidade de escritores e tenta acessar o buffer, caso esteja desbloqueado.
- *saidaEscritor*: Subtrai 1 da quantidade de escritores. Se for o último escritor e houver leitores esperando, libera cada um deles, com o auxílio da variável *aux*.

Os requisitos são:

- (i) Mais de um leitor pode ler ao mesmo tempo: Válido. Caso um leitor tente acessar o buffer enquanto outro leitor estiver lendo, não haverá restrições com o semáforo $bufferOcup$.
- (ii) Apenas um escritor pode escrever de cada vez: Válido. Cada escritor que acessar o buffer irá bloqueá-lo instantaneamente, proibindo a entrada de outros escritores até que a escrita termine
- (iii) Nenhum leitor pode ler enquanto um escritor escrever: Válido. Como o escritor bloqueia o buffer, o primeiro leitor que entrar não poderá acessá-lo.

Não há condições de corrida no código pois cada variável global está sendo travada por um semáforo com a finalidade de exclusão mútua. As variáveis são e, l e aux, e os semáforos são $mutexE$, $mutexL$ e $mutexAux$ respectivamente.

Não é possível que a aplicação se encontre em deadlock, olhando função a função:

- *entradaLeitor*: Os sinais únicos dos semáforos $mutexAux$ e $mutexE$ são consumidos, e depois são produzidos independente do resultado da condição. Depois $mutexL$ é consumido e produzido em sequência. A única forma de continuar travado é se o buffer estiver ocupado, mas mais pra frente, na função que o liberar, é possível ver que o $mutexL$ não é necessário, podendo continuar travado.
- *saidaLeitor*: $mutexL$ é consumido e produzido em sequência, não havendo possibilidade de ficar em espera. Por depender de $mutexL$, caso uma thread leitora tenha sido bloqueada anteriormente, não poderá ser liberada por essa. No entanto, é impossível que as 2 condições aconteçam ao mesmo tempo, pois para a primeira é necessário que não haja leitores lendo, enquanto para a segunda é necessário que o último leitor esteja saindo.

- $entradaEscritor$: $mutexE$ é consumido e produzido logo em sequência. $bufferOcup$ é garantido de estar ocupado.
- $saidaEscritor$: O $bufferOcup$ consumido anteriormente é produzido para repor. Essa produção também é responsável pela condição em $entradaLeitor$ que, como não depende de $mutexL$, não estará travado ao mesmo tempo. Em seguida os semáforos $mutexE$ e $mutexAux$ são consumidos e produzidos logo em sequência

Assim, não é possível que entre em deadlock, pois em cada caso que se utiliza um semáforo, é garantido a "estabilidade"de sinais.

Questão 4

(a)

- Semáforo s: Garantir que todas as threads notificadas tenham sido liberadas antes de permitir novas threads a serem chamadas no wait.
- Semáforo x: Exclusão mútua para a variável aux.
- Semáforo h: Responsável por travar as threads ao entrarem no wait, sendo liberadas somenete com notify ou notifyAll

Foram inicializados corretamente. O semáforo x precisa começar com 1 para realizar a exclusão mútua. Os outros dois semáforos precisam começar com 0 para travar qualquer thread que tente acessá-los.

(b) Sim. Qualquer thread que chamar a função $wait()$ estará bloqueada. Quando chamar a função $notify()$ irá desbloquear uma thread do wait caso haja threads para serem desbloqueadas. Quando chamar a função $notifyAll()$ irá desbloquear todas threads bloqueadas e esperar para que todas saiam do bloqueio, para assim permitir que novas threads possam ser bloqueadas.

(c) Olhando para cada caso:

- Semáforo x: Não. Na função $wait$ é consumido e produzido um sinal em sequência, voltando a estaca zero. No $notify$ e $notifyAll$ há uma espera por s, podendo travar o semáforo. Mas o consumo de s é controlado pela variável aux, assim, todo s criado será consumido e a aplicação não terá problemas nessa parte.
- Semáforo h: Cada sinal de espera é contado na variável aux. Ela é usada depois para liberação de sinais, então todo sinal criado será consumido, retornando a 0 sinais.
- Semáforo s: O controle de sinais desse semáforo é feito inversamente proporcinal ao semáforo h. Pois toda vez que um é criado o outro é consumido e vice-versa. Igualmente, não é possível ter acúmulo de sinais pela variável aux.