Computação Concorrente - UFRJ - 2021.1 - Lista 3

Gabriele Jandres Cavalcanti

DRE: 119159948

Questão 01

a Sim, essa solução garante ausência de condição de corrida e de deadlock. A ausência de condição de corrida é garantida pelo fato do semáforo do recurso ser inicializado com 1 e as threads darem *wait* nele, de modo que somente uma irá acessar o recurso naquele momento. Além disso, todas as variáveis globais estão protegidas por exclusão mútua.

Quando a thread de um tipo finaliza sua execução, ela adiciona um sinal no semáforo do recurso (faz um *post* nele), possibilitando que threads de outros tipos possam executar, o que garante a ausência de deadlock, já que todas as threads terão condições de executar em algum momento.

b Sim, essa solução pode levar as threads a um estado de *starvation*. Isso pode ocorrer devido ao fato de somente threads do mesmo tipo poderem acessar o recurso ao mesmo tempo. Se tivermos muitas threads do tipo A, por exemplo, é muito provável que o programa entre em uma sequência de execuções dessas threads, porque estão sempre chegando, enquanto as threads B e C ficam aguardando. Somente após todas as threads do tipo A executarem é que B e C teriam oportunidade, o que pode demorar muito tempo.

c No caso de o semáforo *rec* ser inicializado com 0 sinais a aplicação sofrerá deadlock porque não haverá sinais que poderão ser consumidos pelas threads. Todas as threads (A,B e C) ficarão presas esperando em $sem\_wait(\&rec)$ e não conseguirão executar.

d No caso de o semáforo *rec* ser inicializado com mais de um sinal uma das condições lógicas da aplicação será ferida. De acordo com o enunciado, somente threads do mesmo tipo podem acessar o recurso ao mesmo tempo. Mas, no caso de haverem mais sinais, mais de um tipo de thread (A,B e C) conseguirá prosseguir no $sem\_wait(\&rec)$ e com isso vão acessar o recurso simultaneamente.

Questão 02

a O erro no código está no fato de que a condicional com a variável compartilhada n não está protegida por exclusão mútua na linha 9 da função dos consumidores. Isso pode ocasionar acúmulo indevido de sinais em d.

Quando o consumidor retira o último elemento do buffer, há o decremento da variável n, que passa a valer 0, e a adição de um sinal ao semáforo s. Nesse momento, pode acontecer de a thread consumidora perder a CPU e uma thread produtora começar a executar. Com isso, ela produz um novo item, incrementa o valor de n e porque ele passa a valer 1, adiciona um sinal ao semáforo d. Quando a thread consumidora ganha CPU novamente, ela consome o item e como n será diferente de 0, a thread não vai executar $sem\_wait(\&d)$, fazendo com que d acumule sinais incorretamente. Devido a esse acúmulo indevido, o consumidor acabará consumindo lixo.

b Uma forma de resolver o problema seria fazer todas as threads consumidoras aguardarem dando *wait* na linha 9, independente do valor de n. Em compensação, sempre que um item é adicionado no buffer deve haver a adição de um sinal no semáforo d.

```
void *cons(void *args) {
  int item;
```

```
  while(1) {
    sem_wait(&d); // espera ter algo no buffer para consumir
    sem_wait(&s);
    retira_item(&item);
    n--; // decrementa o numero de itens no buffer
    sem_post(&s);
    consome_item(item);
  }
}

void *prod(void *args) {
  int item;
  while(1) {
    produz_item(&item);
    sem_wait(&s);
    insere_item(item);
    n++; // incrementa o numero de itens no buffer
    sem_post(&d);
    sem_post(&s);
  }
}
```

Questão 03

a Variáveis globais:

```
int leitores = 0; // contador de threads lendo
int escritores = 0; // contador de threads escritoras aguardando
sem_t escrita; // exclusao mutua para escritores
sem_t mutexL; // exclusao mutua para acesso a variavel leitores
sem_t mutexE; // exclusao mutua para acesso a variavel escritores
sem_t fila; // fila de leitura
```

Inicialização dos semáforos na main:

```
sem_init(&mutexE, 0, 1);
sem_init(&mutexL, 0, 1);
sem_init(&escrita, 0, 1);
sem_init(&fila, 0, 1);
```

Função das threads leitoras:

```
void *leitor(void * arg) {
  int *id = (int *) arg;
  while(1) {
    printf("Thread leitora %d tentando executar...\n", *id);
    sem_wait(&fila); // entra em espera para executar se tiver escritor
        aguardando

    sem_wait(&mutexL); // exclusao mutua para acesso a variavel leitores
    leitores++; // incrementa o numero de threads leitoras executando
    if (leitores == 1) { // primeiro leitor verifica se nao ha alguem
        escrevendo
      sem_wait(&escrita); // entra em espera se tiver escritor escrevendo
    }
    sem_post(&mutexL);

    sem_post(&fila); // incrementa o semaforo de leitura para liberar
        outras threads leitoras simultaneamente
    printf("Thread leitora %d lendo...\n", *id);
```

```
    for (int i = 0; i < 100000000; i++) {;}

    sem_wait(&mutexL);
    leitores--; // decrementa o numero de threads leitoras executando
    printf("Thread leitora %d terminou de ler \n", *id);
    if (leitores == 0) { // ultimo leitor sinaliza que escritores podem
        escrever
      sem_post(&escrita);
    }
    sem_post(&mutexL);
  }
}
```

Função das threads escritoras:

```
void *escritor(void * arg) {
  int *id = (int *) arg;
  while (1) {
    sem_wait(&mutexE); // exclusao mutua para acesso a variavel escritores
    escritores++; // incrementa o numero de threads escritoras aguardando
    printf("%d escritor(es) aguardando \n", escritores);
    if (escritores == 1) { // se for o primeiro escritor aguardando,
        consome o semaforo de leitura para bloquear leitores
      sem_wait(&fila);
    }
    sem_post(&mutexE);

    sem_wait(&escrita); // esclusao mutua para escritores
    printf("Thread escritora %d escrevendo...\n", *id);
    sem_post(&escrita);

    sem_wait(&mutexE);
    escritores--;
    printf("Thread escritora %d terminou de escrever \n", *id);
    if (escritores == 0) { // quando nao tiverem mais escritores aguardando
         incrementa o semaforo da fila de leitura
      printf("Thread escritora %d vai liberar a fila de leitura porque nao
          ha escritores aguardando \n", *id);
      sem_post(&fila);
    }
    sem_post(&mutexE);
  }
}
```

b Além das características originais do padrão leitor/escritor (vários leitores podem ler ao mesmo tempo e apenas um escritor pode escrever ao mesmo tempo), o código funciona de modo a priorizar a escrita.

O código conta com as seguintes variáveis globais:

- **leitores**: contador de threads lendo
- **escritores**: contador de threads escritoras aguardando
- **escrita**: semáforo para garantir exclusão mútua para escritores, de modo que somente um escritor escreva por vez
- **fila**: semáforo para fila de leitura, utilizado para que as threads leitoras aguardem, dando prioridade aos escritores que chegaram
- **mutexL**: semáforo para garantir exclusão mútua para acesso à variável global "leitores"
- **mutexE**: semáforo para garantir a exclusão mútua para acesso a variável global "escritores"

Basicamente, os requisitos são atendido porque as threads leitoras fazem *wait* no semáforo da fila de leitura e quando existem novos escritores aguardando para escrever, não haverá sinal para ser consumido e a thread leitora entra em espera. Caso não haja escritores aguardando, a thread leitora consome o sinal e prossegue normalmente. Antes de começar a realizar a leitura em si, há um incremento de sinal do semáforo de fila de leitura ($sem\_post(\&fila)$), possibilitando que outras threads leitoras possam executar simultaneamente, caso não fiquem bloqueadas por algum escritor tentando escrever. Finalmente, quando o número de threads leitoras for 0 quer dizer que a leitura foi finalizada e o semáforo de escrita é incrementado para possibilitar que escritores executem.

Já as threads escritoras assim que começam a executar incrementam a quantidade de escritores aguardando e caso a thread seja o primeiro escritor aguardando, ela consome o semáforo de fila de leitura para bloquear a entrada de novos leitores. Em seguida, há consumo do semáforo de exclusão mútua para evitar dois escritores escrevendo ao mesmo tempo e a escrita em si. Quando a thread termina de escrever, decrementa o número de threads escritoras aguardando e caso não existam outros escritores esperando, há incremento do semáforo de fila de leitura para liberar a leitura (apesar de ser um único sinal, o post que ocorre nas threads leitoras garante que todas que estiverem esperando serão sinalizadas pelo fato que a thread leitora também emite sinalização para a fila).

Não ocorre condições de corrida indesejadas porque as threads leitoras sempre se bloqueiam caso haja escritor aguardando e as variáveis globais de leitores e escritores são manipuladas em seções críticas protegidas pelos semáforos de exclusão mútua. Não ocorre deadlock porque sempre que o número de threads escritoras aguardando for igual a 0, haverá incremento do semáforo que vai desencadear que a fila de leitoras possa "andar", isso é, haverá um sinal ali que será consumido por uma thread leitora que por sua vez vai disponibilizar outro sinal para eventuais outras threads leitoras que possam estar aguardando.

Questão 04

a O semáforo x serve para garantir a exclusão mútua para acesso à variável global *aux* quando as threads têm que manipulá-la.

O semáforo h serve para manter uma fila de espera, porque dentro da função *wait*, ocorre um $sem\_wait(\&h)$, de forma que se não houverem sinais para serem consumidos a thread entra em espera, e na função $notify$ e $notifyAll$ ocorre $sem\_post(\&h)$ para *uma* thread no primeiro caso, ou para *aux* threads, no segundo caso. Ele serve como a variável de condição implícita citada no enunciado.

O semáforo s serve para controle das threads em espera e controle das execuções de $notify$ e $notifyAll$, uma vez que na função *wait*, temos um $sem\_post(\&s)$, indicando que existem threads em espera e nas funções de notificação temos $sem\_wait(\&s)$, de modo que elas só notificam se houverem sinais indicando que existem threads em espera, caso contrário elas aguardam até que a função *wait* sinalize que alguma thread entrou para a fila. Esse semáforo é que garante que novas threads chamando *wait* ultrapassem outras threads que já estavam na fila, já que após adicionar um sinal no semáforo h, fica aguardando até que a thread que estava esperando consuma o sinal.

Todos os semáforos foram inicializados corretamente, porque a inicialização de x com 1 (semáforo binário) possibilita que somente uma thread consiga prosseguir ao dar *wait*, garantindo a exclusão mútua. A inicialização de h com 0 também está correta porque isso vai fazer com que a primeira thread que execute *wait* também fique em espera, porque não existirão sinais a serem consumidos. E por fim a inicialização de s com 0 também está correta porque isso fará com que as funções de notificação só prossigam se a função de *wait* adicionar um sinal no semáforo indicando que existem threads em espera.

b Sim, essa implementação está correta e garante que a semântica das operações *wait*, $notify$ e $notifyAll$ está sendo atendida. Na função *wait*, é garantido que o lock detido pela thread é liberado por meio do *unlock* do mutex e é garantido que a thread é bloqueada ao chegar no $sem\_wait(\&h)$, já que o semáforo h é inicializado com 0 sinais. Na função $notify$ e $notifyAll$ é garantido que as

threads que eventualmente estejam bloqueadas sejam desbloqueadas por meio do $sem\_post(\&h)$, seja 1 thread no caso de $notify$, ou aux threads, no caso de $notifyAll$.

c No semáforo x essa possibilidade não existe pelo fato de ser um semáforo usado para exclusão mútua e todas as funções iniciam dando *wait* e finalizam dando *post*, para aguardarem caso seja necessário e liberarem assim que finalizarem a seção crítica.

Nos semáforos s e h também não ocorre a possibilidade de acúmulo indevido de sinais porque as operações de *wait* e *post* nesses semáforos são condicionadas pela variável aux, uma vez que a fila de espera de threads cresce com o aumento de aux e diminui com sua diminuição. Como a manipulação de aux está protegida por exclusão mútua, há consumo e adição de sinais no semáforo quando necessário, sem o risco de a variável aux ser manipulada e causar incoerências.