Segunda Lista de Exercícios (2021/2)

Computação Concorrente (ICP-117)
Prof. Silvana Rossetto

[1]IC/UFRJ
22 de fevereiro de 2022

Questão 1 (2,0 pts) O código abaixo[1] propõe uma implementação de *variáveis de condição* e suas operações básicas `wait`, `notify` e `notifyAll` fazendo uso de semáforos. A semântica dessa implementação é similar àquela que vimos em Java: o objeto de lock e a variável de condição são os mesmos e estão implícitos, ou seja, não são passados como argumentos para as funções. A operação `wait` deve liberar o lock atualmente detido pela thread e bloquear essa thread. A operação `notify` deve verificar se há alguma thread bloqueada na variável de condição implícita e desbloqueá-la. A operação `notifyAll` deve verificar se há threads bloqueadas na variável de condição implícita e desbloquear todas elas.

Tarefa Considerando o que foi exposto, examine esse algoritmo e responda, **justificando suas respostas** (respostas sem justificativas não serão consideradas): (a) Qual é a finalidade dos semáforos s, x e h dentro do código? (b) Essa implementação está correta? Ela garante que a semântica das operações $wait$, $notify$ e $notifyAll$ está sendo atendida plenamente? (c) Existe a possibilidade de acúmulo indevido de sinais nos semáforos s, x e h? (d) Esse algoritmo pode chegar a um estado de *deadlock*?

```
//variaveis internas
sem_t  s, x, h;    int aux = 0;
pthread_mutex_t m = PTHREAD_MUTEX_INITIALIZER;
//inicializacoes feitas na funcao principal
sem_init(&s,0,0); sem_init(&x,0,1); sem_init(&h,0,0);

void wait() {
   //pre-condicao: a thread corrente detem o lock de 'm'
   sem_wait(&x);
   aux++;
   sem_post(&x)
   pthread_mutex_unlock(&m);
   sem_wait(&h);
   sem_post(&s);
   pthread_mutex_lock(&m);
}
void notify() {
   sem_wait(&x);
   if (aux > 0) {
      aux--;
      sem_post(&h);
      sem_wait(&s);
   }
   sem_post(&x)
}
void notifyAll() {
   sem_wait(&x);
   for (int i = 0; i < aux; i++)
      sem_post(&h);
   while (aux > 0) {
      aux--;
      sem_wait(&s);
   }
   sem_post(&x);
}
```

[1]Adaptado de Birrell, Andrew D. "Implementing condition variables with semaphores." Computer Systems. Springer, New York, NY, 2004. 29-37.

Questão 2 (2,0 pts) O programa Java mostrado abaixo [2] implementa a classe `Account` que armazena no atributo `balance` o saldo de uma conta. A classe oferece métodos para: obter o saldo (`getBalance`), alterar o saldo (`setBalance`), incrementar o saldo (`increaseBalanceBy`) e decrementar o saldo (`decreaseBalanceBy`). No programa principal, são disparadas duas threads distintas que compartilham uma única instância de `Account`, com um saldo inicial de 500. Uma destas threads é uma instância da classe `RunThread` que irá verificar se o saldo da conta é superior a uma quantia dada e, caso seja, decrementará esta quantia da conta. Esta verificação existe para evitar que a conta fique com o saldo negativo, que seria um estado errôneo. A outra thread é uma instância da classe `ModifierThread`, e tem como função configurar o saldo da conta para um novo valor. Ao final da execução do programa, a thread principal imprime o valor final do saldo na saída padrão.

Tarefa Avalie o programa apresentado e responda as questões abaixo **justificando suas respostas** (respostas sem justificativas não serão consideradas). (a) Em uma execução do programa, o saldo final da conta ficou negativo (situação indesejada). Por que isso aconteceu? (b) Que correções precisam ser feitas no programa para garantir que o saldo da conta não fique negativo e o programa não chegue a um estado de *deadlock*?

```
class Account {
  private double balance;
  Account (double balance) {
    this.balance = balance;
  }
  double getBalance () {
    return this.balance;
  }
  void setBalance (double balance) {
    this.balance = balance;
  }
  void increaseBalanceBy (double amount) {
    balance += amount;
  }
  void decreaseBalanceBy (double amount) {
    balance -= amount;
  }
}
class ModifierThread extends Thread {
  Account account;
  public ModifierThread (Account account) {
    this.account = account;
  }
  public void run() {
    account.setBalance (10);
  } }
class RunThread extends Thread {
  Account account;
  public RunThread (Account account) {
    this.account = account;
  }
  public void run () {
    double amount = 100;
    if (account.getBalance() >= amount) {
      account.decreaseBalanceBy (amount);
    }
  } }
public class Test {
  public static void main (...) { //argumentos omitidos
    Account account = new Account (500);
    Thread a = new RunThread (account);
    Thread b = new ModifierThread (account);
    a.start(); b.start(); a.join(); b.join();
    System.out.println (account.getBalance ());
  } }
```

[2] Extraído da monografia de TCC intitulada "ANÁLISE DE FERRAMENTAS DE TESTE NO CONTEXTO DE APRENDIZADO DE PROGRAMAÇÃO CONCORRENTE", de Mayara Martins Poim Fernandes, 2021.

Questão 3 (2,0 pts) O código abaixo implementa uma aplicação com dois tipos de threads (A e B) que acessam um recurso compartilhado com as seguintes restrições: (i) threads do mesmo tipo podem acessar o recurso ao mesmo tempo; (ii) se há threads do tipo oposto acessando o recurso, é necessário aguardar até que todas terminem de acessar o recurso para então acessá-lo.

Tarefa Avalie o código apresentado e responda as questões abaixo **justificando suas respostas** (respostas sem justificativas não serão consideradas): (a) O que acontecerá se $quatro$ threads B tentarem acessar o recurso no momento em que já houver $duas$ threads A acessando o recurso? (b) O que acontecerá quando as $duas$ threads A terminarem de acessar o recurso? As threads B conseguirão acessá-lo? (c) O que acontecerá se em seguida chegar (logo após a saída das duas primeiras thread A) uma nova thread A e tentar acessar o recurso? (d) Há possibilidade de ocorrência de *starvation* neste código?

```
//variaveis globais
sem_t em, dorme, e; //semaforos para sincronizacao
int cont=0, n_A=0, n_B=0; //contadores de threads
//inicializacoes feitas na main...
sem_init(&em,0,1); sem_init(&dorme,0,0); sem_init(&e,0,1);
//funcao auxiliar para bloqueio das threads
void espera() {
   sem_wait(&em); cont++; sem_post(&em);
   sem_wait(&dorme);
   cont--;
   if(cont>0) sem_post(&dorme);
   else sem_post(&em);
}
//funcao auxiliar para desbloqueio das threads
void libera() {
  sem_wait(&em);
  if(cont>0) sem_post(&dorme);
  else sem_post(&em);
}
//codigo das threads A
void* A (void *args) {
   while(1) {
      sem_wait(&e);
      while(n_B > 0) {
         sem_post(&e);
         espera();
         sem_wait(&e);
      }
      n_A++;  sem_post(&e);
      //...aqui código para acessar o recurso
      sem_wait(&e);
      n_A--;
      if(n_A == 0) libera();
      sem_post(&e);
      //...aqui codigo que nao acessa o recurso
   }
}
//codigo das threads B
void* B (void *args) {
   while(1) {
      sem_wait(&e);
      while(n_A > 0) {
         sem_post(&e);
         espera();
         sem_wait(&e);
      }
      n_B++;  sem_post(&e);
      //...aqui código para acessar o recurso
      sem_wait(&e);
      n_B--;
      if(n_B == 0) libera();
      sem_post(&e);
      //...aqui codigo que nao acessa o recurso
   } }
```

Questão 4 (2,0 pts) Projete um **algoritmo concorrente para gerenciar uma fila de impressão** (a solução pode ser pensada para C ou Java). A thread que irá gerenciar a impressora deverá aguardar por novas requisições de impressão, selecionar uma das requisições pendentes, executar a impressão e voltar a aguardar por novas requisições. As requisições devem ser atendidas na ordem em que foram recebidas. As threads que gerarão as requisições de impressão deverão depositar as requisições de impressão na fila de impressão. **A fila de impressão terá um tamanho limitado**, então caso a thread encontre a fila cheia, ele deverá ficar retida até conseguir depositar a sua requisição na fila, e só então prosseguir com a sua execução normal. **Comente a solução proposta, justificando as principais decisões tomadas**.

Questão 5 (2,0 pts) Estenda a solução da Questão 4, permitindo que a fila de impressão possa ser consultada para saber quantas requisições estão aguardando para serem atendidas. **Comente a solução proposta, justificando as principais decisões tomadas**.