Lista 3 – Computação Concorrente

Aluno: Carlos Cozzolino

DRE: 115086800

Exercício 1:

1. Sim, como temos semáforos para exclusão mútua que são incrementados e logo depois decrementados, não acontece *deadlocks ou condições de corrida* no programa.
2. Não, pois a próxima função só pode ser executada depois que todas as threads que participam da atual função em operação terminam suas tarefas.
3. Nenhuma função começa a operar, isso ocorre pois todas as 3 funções aguardam um sinal de ‘rec’ para começar e, se ele começar em zero, elas ficam esperando para sempre o sinal.
4. Caso o sinal de ‘rec’ seja diferente de 1, as threads de outras funções poderão entrar e pegar o recurso “SC” numa só iteração do código.

Exercício 2:

1. Da maneira como foi programado, pode acontecer a seguinte situação: depois de a thread consumidora esvaziar o buffer, liberar o semáforo ‘s’ e colocar o valor de ‘n’ valendo zero, essa thread irá terminar sua execução e vai se retirar temporariamente da operação, porém, nesse tempo uma thread produtora pode incrementar o valor do semáforo ‘d’ e da variável ‘n’ antes da thread consumidora retomar suas tarefas. Com isso, o semáforo ‘d’ pode acumular um sinal errado, permitindo que o consumidor consuma quando o buffer estiver vazio, mesmo que ‘n’ volte a valer zero.
2. Já que não podemos adicionar novos semáforos, uma solução possível seria criar uma variável local na função ‘cons’, essa variável receberia o valor de ‘n’ sempre que este fosse decrementado. Então, substituímos o condicional para decrementar o semáforo ‘d’ quando a nova variável valesse zero. A função produtora não precisa de mudanças.

Exemplo:

* 1. int n = 0;
  2. sem\_t s, d;
  3. void \*cons(){
  4. int aux = 0;
  5. sem\_wait(&d);
  6. while(1){
  7. sem\_wait(&s);
  8. retira\_item(&item);
  9. n--;
  10. aux = n;
  11. sem\_post(&s);
  12. consome\_item(item);
  13. if(m == 0){
  14. sem\_wait(&d);
  15. }}}

Exercício 3:

1. Link para o Github onde tem o código anexado no arquivo, “Quest3.c” é o código implementado pelo aluno.
2. O código funciona da seguinte maneira: a função ‘main’ é iniciada dando um sinal para os nossos 3 semáforos, com isso o semáforo de exclusão mútua ‘mutex’ é incrementado e o semáforo ‘prior’ também, assim uma thread da função leitora é desbloqueada e pode incrementar a variável ‘readers’, criando uma fila de leitores que precisam ler. Sempre que nosso semáforo ‘prior’ desbloqueia uma thread leitora, esta incrementa nossa fila ‘readers’, fazendo com que todas as threads leitoras que estão aguardando operações sejam utilizadas em algum momento. Logo, criamos um condicional que decrementa o semáforo ‘wrt’ sempre que nossa fila de ‘readers’ for igual a 1, isso é feito para que quando a thread escritora termine sua execução, o incremento feito por ele no semáforo seja o suficiente para liberar todas as threads leitoras. A mesma ideia é aplicada quando ‘readers’ for igual a 0, fazendo apenas um incremento em ‘wrt’ pode acontecer uma de duas coisas: ou as threads escritoras irão executar, ou as threads leitoras irão reiniciar. Com isso, podemos ir para o caso da função leitora, onde a primeira thread deste tipo irá decrementar o semáforo ‘prior’, depois utilizaremos os semáforos ‘mutex’ e ‘wrt’ para fechar a seção crítica da função e não permitir que mais de uma thread escritora escreva enquanto temos uma trabalhando. A fila de ‘writers’, que foi criada no início da função, é decrementada em um e quando ela chega a zero podemos incrementar novamente o semáforo ‘prior’.

A função não possui *deadlocks ou condições de corrida* já que todos os semáforos são inicializados com um sinal e o semáforo criado para exclusão mútua ‘mutex’ é sempre incrementado e decrementado.

E a função atende os requisitos do problema já que a função leitora sempre começa verificando se algum escritor entrou na fila, caso positivo, ela termina a leitura e bloqueia os leitores, dando prioridade para a função de escritores. Da parte dos escritores temos que sempre que algum deles tentar escrever, a fila de escritores será incrementada em 1, quando a escrita inicia o semáforo de prioridade é decrementado em 1 e só é incrementado novamente quando todas as threads de escrita terminam suas tarefas.

Exercício 4:

1. O semáforo ‘x’ garante a exclusão mútua das threads. Logo, precisamos iniciar ele com 1.

O semáforo ‘h’ bloqueia as threads que executam a operação ‘wait’. Logo, precisamos iniciar ele com 0.

O semáforo ‘s’ garante que os sinais cheguem nas threads que irão executar a operação ‘wait’. Logo, precisamos iniciar ele com 0.

Portanto, todos os semáforos foram iniciados corretamente.

1. A implementação está correta, a função ‘wait’ desliga até receber algum sinal de ‘notify’ ou ‘notifyAll’, a função ‘notify’ acorda uma thread de cada vez, podendo ser interrompida antes de acordar todas a sua pool de threads e a função ‘notifyAll’ acorda toda sua pool de threads quando é chamada, não podendo ser interrompida.

O código “Quest4.c” mostra o programa do exercício em prática.

1. Não, pois todos os semáforos são atualizados nas 3 funções e a variável ‘aux’ é modificada dentro da seção crítica das funções, evitando assim possíveis *deadlocks ou condições de corrida*.