Lista 2 de Computação Concorrente

Gabriel da Fonseca Ottoboni Pinho - DRE 119043838
Rodrigo Delpreti de Siqueira - DRE 119022353

15/05/2021

Questão 1

A

A alternância entre as execuções das threads é garantida. Após o início do programa, as threads `Bar` são imediatamente pausadas pela função `pthread_cond_wait`, enquanto que as de tipo `Foo` seguem livremente até que são pausadas depois de completar suas tarefas. Quando `M` threads `Foo` são pausadas, a função `pthread_cond_broadcast` é chamada e então as threads `Bar` são despausadas. Analogamente às threads `Bar`, as threads `Foo` são pausadas depois que terminam suas tarefas, com a última liberando as threads `Bar`, que estavam pausadas. Dessa forma, os dois tipos de thread são executados de maneira alternada.

B

As variáveis `contaFoo` e `contaBar` não sofrem de condições de corrida, pois todo acesso a essas é controlado por uma mutex. Contanto que nenhuma das threads termine, a alternância continua ocorrendo, sem a presença do *deadlock*. Caso contrário, entretanto, todas as threads restantes ficarão bloqueadas, pois `contaBar` ou `contaFoo` deixará de ser atualizado.

Questão 2

A

B

C

Questão 3

A

Uma thread `A`, com a mutex adquirida, incrementou `x` para 10 e verificou que 10 é um múltiplo de 10. Com isso, a thread `B`, que estava pausada, foi despausada. Apesar disso, a função `pthread_cond_wait` só poderá de fato retornar depois que a mutex puder ser adquirida por `B`. Temos, então, uma condição de corrida, pois após a mutex ser liberada por `A`, não necessariamente `B` será a próxima adquiri-la.

Para printar 11, após a chamada a `pthread_cond_signal` e a liberação da mutex, alguma outra thread `A` adquiriu a mutex e incrementou `x` mais uma vez. Depois, a mutex foi liberada e só então `B` conseguiu a adquirir, retornando de `pthread_cond_wait`. Por conta da aquisição tardia, o valor de

`x` já não era mais 10 e 11 foi impresso.

B

A causa da condição de corrida é a forma como o valor de `x` é passado de `A` para `B`. Sabendo disso, essa troca de informações poderia ser feita usando um modelo de produtores e consumidores. As threads `A` seriam as produtoras e a thread `B` seria a consumidora. Isso resolveria a condição de corrida, pois o valor de `x` seria copiado para um buffer, garantindo a exitência do valor até que `B` o acesse.

Como não há nenhum loop na thread `B`, apenas o primeiro múltiplo será impresso. Tirando vantagem desse comportamento, uma versão simplificada de produtores e consumidores foi implementada. A variável `print_x` faz o papel do buffer, e `A` só escreve no buffer uma única vez. A mutex `print_mutex` não era estritamente necessária, mas foi adicionada a fim de manter o modelo do código original. A função `pthread_cond_wait` exige uma mutex e não faria sentido usar a mutex da outra variável, que nem é utilizada em `B`.

```
int x = 0;
pthread_mutex_t x_mutex;
pthread_cond_t x_cond;

int print_x = 1;
pthread_mutex_t print_mutex;

void *A (void *tid) {
    for (int i=0; i<100; i++) {
        pthread_mutex_lock(&x_mutex);
        pthread_mutex_lock(&print_mutex);
        x++;
        if(!(x%10) && print_x == 1) {
            print_x = x;
            pthread_cond_signal(&x_cond);
        }
        pthread_mutex_unlock(&print_mutex);
        pthread_mutex_unlock(&x_mutex);
    }
}

void *B (void *tid) {
```

```
    pthread_mutex_lock(&print_mutex);
    if(print_x%10)
        pthread_cond_wait(&x_cond, &print_mutex);
    printf("X=%d\n", print_x);
    pthread_mutex_unlock(&print_mutex);
}
```