Primeira Lista de Exercícios (2021/2)

Computação Concorrente (ICP-117)
Prof. Silvana Rossetto

[1]IC/UFRJ
16 de dezembro de 2021

Questão 1 (1,5 pts) Responda as questões abaixo, justificando todas as respostas:

(a) O que caracteriza que um programa é concorrente e não sequencial?
(b) Qual será a *aceleração máxima* de uma aplicação que possui 3 tarefas que consomem o mesmo tempo de processamento, das quais 2 poderão ser executadas de forma concorrente? Quais são os requisitos de execução para que a aplicação possa alcançar a aceleração calculada?
(c) O que é *seção crítica* do código em um programa concorrente?

Questão 2 (2,0 pts) Encontre e descreva um algoritmo (ou aplicação) que você já tenha usado ou estudado e que poderia ser escrito de forma concorrente. **(a)** Apresente os parâmetros de entrada do algoritmo e os valores de saída esperados. **(b)** Discuta ao menos duas formas diferentes de dividir as tarefas internas do algoritmo entre fluxos de execução distintos, mostrando se elas garantem balanceamento de carga ou não. **(c)** Aponte as demandas de sincronização que a solução concorrente trará e em quais partes do código. (Não é necessáio implementar a aplicação.)

Questão 3 (2,0 pts) Considere uma aplicação onde três (3) threads são disparadas para executar a função abaixo. A variável $saldo$ é definida no escopo global com valor inicial igual a 0. Depois de todas as threads concluirem sua execução, o valor de $saldo$ é impresso na saída padrão pelo fluxo principal da aplicação.

```
void* tarefa (void* arg) {
   for(int i=0; i<2; i++)
      saldo = saldo + 100;
}
```

Responda as questões abaixo **justificando suas respostas** (respostas sem justificativas não serão consideradas): **(a)** Pode ocorrer de um valor acima de 600 ser impresso na saída padrão? **(b)** Pode ocorrer do valor 100 ser impresso na saída padrão? **(c)** Pode ocorrer do valor 200 ser impresso na saída padrão? **(d)** Pode ocorrer do valor 500 ser impresso na saída padrão?

Questão 4 (2,0 pts) Em um trabalho de Shan Lu et al. [1] são apresentados *bugs* de concorrência encontrados em aplicações reais (MySQL, Apache, Mozilla and OpenOffice). Alguns deles estão transcritos abaixo. Para cada caso, proponha uma solução para o *bug*.

Caso 1 (bug de violação de atomicidade no MySQL): Nesse caso temos duas threads (Thread 1 e Thread 2). Como nós programadores estamos mais acostumados a pensar de forma sequencial, temos a tendência de assumir que pequenos trechos de código serão executados de forma atômica. Os programadores assumiram nesse caso que se o valor avaliado na sentença 1 (S1) é diferente de NULL, então esse mesmo valor será usado na sentença 2 (S2). Entretanto, pode ocorrer em uma execução qualquer que a sentença 3 (S3) quebre essa premissa de atomicidade, causando um erro na aplicação. **(a)** Mostre qual ordem de execução das sentenças vai gerar o erro. **(b)** Proponha uma correção no código para evitar esse erro.

```
Thread 1:
S1: if (thd->proc_info) {
  S2: fputs(thd->proc_info,...);
}
```

```
Thread 2:
S3: thd->proc_info=NULL;
...
```

[1]LU, Shan et al. "Learning from mistakes: a comprehensive study on real world concurrency bug characteristics". Proceedings of the 13th international conference on Architectural support for programming languages and operating systems. 2008. p. 329-339.

Caso 2 (bug de violação de ordem no Mozilla): Nesse caso também temos duas threads (Thread 1 e Thread 2). A thread 2 só deveria acessar a variável `mThread` depois dela ser devidamente inicializada. **(c)** Proponha uma correção no código para garantir que essa condição seja sempre satisfeita.

```
Thread 1:
void init (...) {
   mThread=PR_CreateThread (mMain,...);
   ...
}
```

```
Thread 2:
void mMain(...) {
   mState=mThread->State;
   ...
}
```

Questão 5 (2,5 pts) Uma aplicação em C dispara a Thread 1 (código definido abaixo) na sua inicialização. Deseja-se acrescentar outra thread (Thread 2) nessa aplicação para imprimir na tela o valor da variável `contador` sempre que ela alcançar um valor múltiplo de 100. A Thread 1 deverá aguardar a impressão ser concluída antes de alterar novamente o valor de `contador`. A Thread 2 deverá finalizar quando não houver mais necessidade da sua atuação. **(a)** Usando **mecanismos de sincronização**, implemente a Thread 2 de forma a atender aos requisitos colocados. Altere o código da Thread 1, caso seja necessário. **(b)** Descreva como sua solução funcionará e de que forma garantirá que os requisitos serão atendidos.

```
int contador = 0;

void *Thread1(void *arg) ) {
  for (int i=0; i<N; i++) {
     contador++;
     ...//faz processamento adicional
  }
}

void *Thread2(void *arg) ) {
...
}
```