Sistemas Operacionais – Prof. Rafael R. Obelheiro

Exercícios — IPC no Linux

1. O código abaixo possui ao menos uma condição de disputa.

   (a) Execute o código várias vezes para comprovar a existência da(s) condição(ões) de disputa. Dicas:

      i. No Linux, você pode fazer isso usando o terminal:

      ```
      $ for i in {1..10} ; do ./prog ; done | sort | uniq -c
      ```

      A linha acima executa o programa `./prog` 10 vezes, e conta quantas vezes é gerada cada saída distinta. (Você pode ter de executar mais de 10 vezes para encontrar uma saída errônea.)

      ii. Por padrão, o programa cria `nthr` = 100 *threads*. Esse número pode ser alterado passando um argumento na linha de comando. Por exemplo, para criar 200 *threads*, execute o código com `./prog 200`.

   (b) Use as condições de Bernstein para identificar a(s) região(ões) crítica(s) no código.

   (c) Elimine a(s) condição(ões) de disputa usando mutex, de modo que o valor final de `done` seja sempre igual ao de `nthr`.

```c
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/types.h>
#include <pthread.h>
#include <assert.h>

int done = 0;

void f1(void *arg) {
    done++;
    pthread_exit(NULL);
}

int main(int argc, char *argv[]) {
    pthread_t *thr;
    int rc, i, nthr;

    nthr = (argc > 1 ? atoi(argv[1]) : 100);
    thr = calloc(nthr, sizeof(pthread_t));
    assert(thr != NULL);
    for (i = 0; i < nthr; i++) {
        rc = pthread_create(&thr[i], NULL, (void *)f1, NULL);
        assert(rc == 0);
    }
    for (i = 0; i < nthr; i++) {
        rc = pthread_join(thr[i], NULL);
        assert(rc == 0);
    }
    printf("done=%d\n", done);
    return 0;
}
```

2. O código abaixo cria `nthr` *threads* que manipulam de forma concorrente uma base com `ncontas` contas bancárias. Cada *thread* realiza `ntransf` transferências entre duas contas; as contas de origem e destino e o valor transferido em cada operação são escolhidos ao acaso. Os valores *default* para `nthr`, `ncontas` e `ntransf` são 10, 100 e 10, respectivamente, e podem ser modificados passando argumentos na linha de comando.

   (a) Como as *threads* apenas transferem valores de um conta para outra, espera-se que o somatório dos saldos finais de todas as contas seja igual ao somatório de saldos iniciais. Verifique se isso acontece ao executar o programa.

   (b) Use um mutex para eliminar a condição de disputa no código.

   (c) A sua solução do item anterior provavelmente força a realização de uma única transferência de cada vez. No entanto, se uma *thread* quiser transferir um valor da conta 1 para a conta 2 e outra *thread* quiser transferir da conta 3 para a conta 4, não há problema em realizar essas operações simultaneamente. Modifique sua solução do item anterior para permitir que transferências entre pares distintos de contas possam ocorrer em paralelo. Tome cuidado para garantir que seu código não fique passível de *deadlock*.

```c
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/types.h>
#include <pthread.h>
#include <assert.h>

unsigned int ncontas, maxvalor = 50;
long *cta;
long saldo_inicial = 100;

/* Transfere 'valor' da conta 'c1' para a conta 'c2' */
void transfere(unsigned int c1, unsigned int c2, long valor) {
    if ((c1 < ncontas-1) && (c2 < ncontas-1) && (c1 != c2)) {
        cta[c1] -= valor;
        cta[c2] += valor;
    }
}

void thread(void *arg) {
    unsigned int c1, c2, valor;
    unsigned long i, ntransf = (unsigned long)arg;
    for (i = 0; i < ntransf; i++) {
        c1 = random() % ncontas;
        do {
            c2 = random() % ncontas;
        } while (c1 == c2);
        valor = (random() % maxvalor) + 1;
        transfere(c1, c2, valor);
    }
    pthread_exit(NULL);
}

int main(int argc, char *argv[]) {
    pthread_t *thr;
    int rc;
    unsigned int i, nthr;
    unsigned long ntransf;
    unsigned long long total_inicial, total_final;

    nthr = (argc > 1 ? atoi(argv[1]) : 100);
    thr = calloc(nthr, sizeof(pthread_t));
```

```
43      assert(thr != NULL);
44      ncontas = (argc > 2 ? atoi(argv[2]) : 10);
45      cta = calloc(ncontas, sizeof(long));
46      assert(cta != NULL);
47      for (i = 0; i < ncontas; i++)
48          cta[i] = saldo_inicial;
49      total_inicial = ncontas * saldo_inicial;
50      ntransf = (argc > 3 ? atoi(argv[3]) : 100);
51      for (i = 0; i < nthr; i++) {
52          rc = pthread_create(&thr[i], NULL, (void *)thread,
53                              (void *)ntransf);
54          assert(rc == 0);
55      }
56      for (i = 0; i < nthr; i++) {
57          rc = pthread_join(thr[i], NULL);
58          assert(rc == 0);
59      }
60      total_final = 0;
61      for (i = 0; i < ncontas; i++)
62          total_final += cta[i];
63      if (total_inicial == total_final)
64          printf("OK");
65      else
66          printf("ERRO");
67      printf(" -- total inicial=%llu, total final=%llu\n", total_inicial,
68            total_final);
69      return 0;
70  }
```

3. Considere um programa concorrente com três *threads*, `X`, `Y` e `Z`, mostradas abaixo.

```
int n = 1;     /* variável compartilhada entre X, Y e Z */
```

```
void X() {
   n = n * 16;
}
```

```
void Y() {
   n = n / 7;
}
```

```
void Z() {
   n = n + 40;
}
```

   Implemente esse programa usando Pthreads, e use variáveis de condição para garantir que o resultado final de `n` seja sempre 8.

4. Modifique a sua solução do exercício 6 da lista de Pthreads (contagem até 2^{31}), usando uma barreira para que todas as *N threads* iniciem juntas a contagem.

5. Analise sua solução para o exercício 7 de lista de Pthreads (busca do menor e do menor valores em um vetor de números aleatórios), verificando a existência de condições de disputa. Caso identifique uma ou mais condições de disputa, elimine-a(s) usando mutexes, variáveis de condição e/ou semáforos POSIX.

6. Generalize o código do produtor-consumidor com Pthreads visto em aula (solução do Tanenbaum) para usar um buffer circular com *N* posições.

7. Escreva um programa *multithread* em que N *threads* usem repetidas vezes uma barreira para sincronização. Cada *thread i* repete 10 vezes os seguintes passos:

   (1) Incrementa uma variável inteira 10^9 vezes (se necessário, ajuste o número de incrementos para que cada rodada demore alguns segundos);

   (2) Insere o valor i no final de uma fila que armazena a sequência de *threads* que terminaram o laço interno;

   (3) Se for a última *thread* a terminar, imprime a fila, mostrando assim a ordem em que as *threads* concluíram os incrementos a cada rodada;

   (4) Espera em uma barreira até que todas as *threads* tenham concluído a rodada antes de iniciar a próxima rodada.

   O número N de *threads* a serem criadas deve ser um parâmetro informado via linha de comando (`argv`).

   Verifique se pode ser identificada alguma tendência na ordem de conclusão das *threads* quando N for menor, igual ou maior do que o número de núcleos do seu processador. Considere questões como: a ordem é sempre a mesma em todas as rodadas? A primeira ou última *thread* é sempre a mesma? Existe alguma ordem que aparece com muito mais frequência que as demais? O comportamento se repete em múltiplas execuções do programa?

8. Escreva um programa *multithread* para encontrar o menor elemento em uma matriz $N \times N$ de inteiros sem sinal. O número T de *threads* a serem criadas deve ser um parâmetro informado via linha de comando (`argv`). A matriz deve ser logicamente particionada entre as *threads* (você pode supor que N é divisível por T): cada *thread* deve primeiro encontrar o menor elemento em sua partição, e depois reportar quantas *threads* encontraram elementos menores que o seu. O programa principal (`main()`) deve reportar o menor elemento da matriz. Exemplo de execução:

```
$ ./barreira 4
Thread 1: menor elemento=0, threads com elementos menores=0
Thread 2: menor elemento=61, threads com elementos menores=3
Thread 0: menor elemento=58, threads com elementos menores=2
Thread 3: menor elemento=18, threads com elementos menores=1
Menor elemento da matriz: 0
```

   A matriz pode ser preenchida com valores aleatórios. Lembre-se que, para que uma *thread* possa determinar quantas *threads* encontraram elementos menores que o seu mínimo, é necessário que todas as *threads* tenham encontrado o seu menor (use uma barreira para garantir isso).

9. Resolva o exercício 1 usando semáforos POSIX em vez de mutexes.

10. Resolva o exercício 2 usando semáforos POSIX em vez de mutexes.

11. Resolva o exercício 3 usando semáforos POSIX em vez de variáveis de condição.

12. O que acontece caso um processo tente usar uma região de memória compartilhada maior do que a alocada? Por exemplo, caso seja alocado espaço para um vetor de 1000 inteiros, o que acontece se um processo tentar usar 4000 inteiros?

13. Resolva o exercício 3 com processos no lugar de *threads*, usando `fork()`, memória compartilhada e semáforos.

14. Generalize o código do produtor-consumidor com memória compartilhada e semáforos visto em aula para usar um buffer circular com N posições.

15. Modifique a sua solução do exercício 6 para usar semáforos POSIX no lugar de mutexes e variáveis de condição.