Sistemas Operacionais – Prof. Rafael R. Obelheiro

Exercícios — IPC no Linux

1. O código abaixo* possui uma condição de disputa. Elimine essa condição de disputa do código usando mutex, de modo que o programa sempre imprima `n=0`.

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

#define MAX 2000

int n;

void f1(void *argp) {
    int i, temp;
    for (i = 0; i < MAX; i++) {
        temp = n;
        temp++;
        n = temp;
    }
}

void f2(void *argp) {
    int i, temp;
    for (i = 0; i < MAX; i++) {
        temp = n;
        temp--;
        n = temp;
    }
}

int main(void) {
    pthread_t t1, t2;
    int rc;
    n = 0;
    rc = pthread_create(&t1, NULL, (void *)f1, NULL);
    rc = pthread_create(&t2, NULL, (void *)f2, NULL);
    rc = pthread_join(t1, NULL);
    rc = pthread_join(t2, NULL);
    printf("n=%d\n", n);
    return 0;
}
```

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

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

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

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

```c
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.

*Exercício 6 da lista de Pthreads, disponível como `pth-6.c` no arquivo de exemplos de IPC no Linux.

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

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

5. 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?

6. 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).

7. Resolva o exercício 1 usando semáforos POSIX em vez de mutex.

8. Resolva o exercício 2 usando semáforos POSIX em vez de variáveis de condição.

9. 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?

10. Resolva o exercício 2 com processos no lugar de *threads*, usando `fork()`, memória compartilhada e semáforos.

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

12. Modifique a sua solução do exercício 4 para usar semáforos POSIX no lugar de mutex e variáveis de condição.