Universidade do Vale do Itajaí
Escola Politécnica
Curso: Engenharia de Computação
Professor: Felipe Viel
Disciplina: Sistemas de Tempo Real
Exercícios Sistemas de Tempo Real – Execução e captura de tempo e revisão de threads

O código de exemplo está em: Link

Outros código de exemplo e que também pode ser usados para os exercícios podem encontrados em: Link nas pastas Process, Threads e Time Execution

1) Para o código de exemplo, compile o mesmo e o execute para que o laço de repetição tenha apenas uma execução. Qual o tempo de execução informado ao fim da execução?

2) Para o mesmo código, implemente um laço de repetição com 100 interações para as 5 tarefas. Quanto é o tempo de execução de total para o software?

3) Repita os dois exercícios anteriores para os diferentes níveis de otimização dos códigos (O0, O1, O2, O3).

4) Repita o exercício 2) agora obtendo o tempo para executar cada tarefa. Utilize as bibliotecas time.h e intrin.h para o obter os valores de tempo de execução.

   Usando a função clock():

```
#include <time.h>
….
clock_t begin, end;
double time_spent;
….
begin = clock();
//função a ser observada
end = clock();
…..
time_spentf = (double)(end - begin)/CLOCKS_PER_SEC;
print(time_spent)
```

   Usando a função time():

```
#include <time.h>
….
time_t begin, end;
double time_spent;
….
begin = time(NULL);
//função a ser observada
end = time(NULL);
```

```
…..
time_spent = (end - begin);
printf(time_spent)
```

Usando a função clock_gettime():

```
#include <time.h>
….
#define BILLION  1000000000.0
….
struct timespec start, end;
double time_spent;
….
clock_gettime(CLOCK_REALTIME, &start); // vale a pena a leitura
//função a ser observada
clock_gettime(CLOCK_REALTIME, &end);
…..
time_spent = (end.tv_sec - start.tv_sec) +
                    (end.tv_nsec - start.tv_nsec) / BILLION;
printf(time_spent)
```

Usando a função intrin.h:

```
#include <intrin.h>
#include <stdint.h>
…..
//  Windows
#ifdef _WIN32
#include <intrin.h>
#include <stdint.h>
uint64_t rdtsc(){
   return __rdtsc();
}
//  Linux/GCC
#else
uint64_t rdtsc(){
   unsigned int lo,hi;
   __asm__ __volatile__ ("rdtsc" : "=a" (lo), "=d" (hi));
   return ((uint64_t)hi << 32) | lo;
}
#endif
......
uint64_t begin, end, cycles_spent;
......
begin = rdtsc();
//função a ser observada
end = rdtsc();
…..
cycles_spent = (end - begin);
```

printf(cycles_spent)

5) Escolha uma das bibliotecas citadas anteriormente para avaliar o conceito de tempo de execução pela média. Para obter a média, obtenha 100 amostras da execução e calcule a média e a exiba. Também armazene e exiba o melhor tempo e o pior tempo de execução observados. Qual a função mais lenta e qual a função mais "rápida".

6) Em um programa concorrente existem várias tarefas do tipo "calcula" e várias tarefas do tipo "mostra". As tarefas do tipo "calcula" iniciam sua execução no início do programa. Já as tarefas do tipo "mostra" precisam esperar autorização das tarefas "calcula" para começar. Além disto, cada três tarefas "calcula" permitem que exatamente uma tarefa "mostra" execute. Implemente uma solução para este problema usando semáforos. A solução consiste de 2 rotinas:

/* tarefa "calcula" usa para informar que está autorizando, ela mesmo nunca fica bloqueada */ void autoriza (void);

/* tarefa "mostra" usa para ficar bloqueada até que pelo menos três autorizações sejam feitas, quando então ela é liberada, não importa quando as autorizações foram feitas */

void espera_3_autorizacoes(void);

Cada 3 autorizações permite a liberação de apenas uma tarefa "mostra". Não importa se as autorizações aconteceram antes ou depois da tarefa "mostra" chamar a rotina "espera_3_autorizacoes()". Não é permitido deixar uma tarefa "mostra" bloqueada no caso de já terem acontecido 3 chamadas da rotina "autoriza" que não foram usadas. A tarefa "calcula" nunca bloqueia.

5) Um simulador de tráfego urbano foi implementado como um programa concorrente para melhor aproveitar os computadores com arquitetura multicore. Neste simulador cada carro é representado por uma thread. Todas as vias são de sentido único. Cada cruzamento é representado por um monitor.

Quando um carro deseja passar por um cruzamento, a thread que o representa chama a rotina "pedePassar()" do monitor e passa como parâmetro o sentido desejado. A thread então fica bloqueada até que a passagem seja autorizada. Depois que o carro passou pelo cruzamento, a thread que o representa chama a rotina "jaPassei()" com mutex. A gerência do recurso cruzamento é feita da seguinte forma:

- Se o carro pede para passar, e o cruzamento está livre, ele pode passar;

- Se o carro pede para passar, e estão passando carros no mesmo sentido que ele, ele pode passar;

- Se o carro pede para passar, e estão passando carros no outro sentido, ele fica em espera.

Implemente o mutex utilizando as funções da biblioteca de pthreads, sem postergação indefinida e sem deadlock. O monitor em questão possui duas rotinas de acesso:

```
void pedePassar( int sentido);// 1 é norte-sul, -1 é leste-oeste

void jaPassei( void);
```

de Oliveira, Rômulo Silva. Fundamentos dos Sistemas de Tempo Real: Segunda Edição (Portuguese Edition) (p. 290). Edição do Kindle.