Iniciado em	Sunday, 8 May 2022, 12:32
Estado	Finalizada
Concluída em	Sunday, 8 May 2022, 12:33
Tempo empregado	1 minuto 38 segundos
Avaliar	**10,0** de um máximo de 10,0(**100**%)

Questão 1

Correto

Atingiu 7,5 de 7,5

1. O método *resume(boolean unpreemptive)* é chamado com *unpreemptive=true* para impedir que a thread corrente seja preemptada durante a execução do método ✓. A postergação de uma eventual preempção em *resume()* pressupõe uma invocação interna do SO por um método que já trata preempção ✓.
2. As mudanças no sistema para implementar uma versão não-preemptiva do método *resume()* implicaram em que o método respectivo da interface da classe Thread faça um forward para a o método protegido ✓.
3. O método *reschedule()* seleciona a thread com a maior prioridade na fila *Ready* e conduz sua execução através do método yield() ✓.
4. Quando uma thread é criada em estado **READY** e o sistema está configurado como **preemptivo**, uma chamada para Thread::reschedule() acontece dentro do construtor da classe de acordo com a prioridade da thread criada ✓.
5. Qual política de escalonamento é implementada pelo EPOS em seu estado atual? múlti-nível, com prioridade estática como política primária e round-robin como política secundária ✓
6. Se a versão atual do EPOS for configurada como **não-preemptiva**, uma thread executando um *loop* infinito impedirá que outras threads de menor prioridade executem ✓.
7. O método *~Thread()* invoca *resume()* sobre uma eventual *joining thread*. Caso o sistema esteja configurado como **preemptivo** e a invocação de *resume()* implique em *reschedule()*, a memória alocada para servir de pilha para a thread sendo deletada só será liberada quando a thread que chamou delete voltar a executar ✓.
8. Qual método da classe Thread deve ser alterado para impedir a ativação do *time-slicer*? Thread::init() ✓.

Questão 2

Correto

Atingiu 2,5 de 2,5

1. Sobre o código abaixo, executando sobre a arquitetura RV32/RISCV/SiFive_E single-core e com preempção habilitada, considerando a versão atual do código do EPOS, responda:

```
#include <utility/ostream.h>
#include <process.h>

using namespace EPOS;

OStream cout;

const unsigned int thread_num = 5;
Thread * t[thread_num];
int code;

int func_a() {
  cout << code-- << " " << endl;
  return 0;
}

int func_b() {
  for (unsigned int i = 0; i < thread_num; ++i) {
    t[i] = new Thread(&func_a);
    cout << code++ << " " << endl;
  }
  for (unsigned int i = 0; i < thread_num; ++i) {
    t[i]->join();
    delete t[i];
  }
  return 0;
}

int main()
{
  cout << "Simple Test" << endl;
  Thread * c = new Thread(&func_b);
  c->join();
  cout << "\nThe end!" << endl;
  delete c;
  return 0;
}
```

A saída do programa gerada pelas funções *func_b* e *func_a* é "0 -1 0 -1 0 -1 0 -1 0 -1". Aumentando a prioridade da *Thread "c"* (que executa func_b) no momento de sua criação, a saída obtida é 0 1 2 3 4 5 4 3 2 1 ✓ . Isso ocorre pois quando uma thread é criada na versão atual do EPOS configurado para ser preemptivo, temos um reavaliação da thread em execução dependente da ordem da fila de escalonamento, logo se *_running* e a primeira thread da fila tem a mesma prioridade a troca ocorre pois o método yield reinsere _running na fila de ready e a ordenação se da pela relação ">" e não ">=", o que a insere atrás de threads de igual prio ✓ .

◄ OSDI with EPOS: Preemptive Scheduling

Seguir para...

OSDI with EPOS: Timing and Alarms ►