Iniciado em	Tuesday, 26 Apr 2022, 22:12
Estado	Finalizada
Concluída em	Tuesday, 26 Apr 2022, 22:17
Tempo empregado	4 minutos 30 segundos
Avaliar	**9,9** de um máximo de 10,0(**99**%)

Questão 1

Correto

Atingiu 2,5 de 2,5

1. A instrução de máquina *tsl* (*test and set lock*) é uma instrução [atômica] e é utilizada para implementar mecanismos de sincronização a nível de sistema. Ela faz uso de mecanismos de *bus locking* para ler o valor de [uma posição da memória] para [um registrador] e seta o valor [da posição da memória] para [um valor diferente de 0 (zero)]. O valor [do registrador] é então é comparado e caso seja [igual a 0 (zero)] dize-se que o *lock* foi obtido. Ela é utilizada, por exemplo, para implementar-se mecanismos de sincronização de processos como [Mutex].

2. Um mecanismo para implementar-se relações de [composição ou agregação (tem um)] em C++ é [a herança privada]. Neste tipo de relação, os elementos da classe base são reutilizados sem implicar em uma relação de [tipos] entre as partes. Os elementos reutilizados são [herdados com visibilidade privada], fazendo com que os métodos públicos da classe base [não integrem] interface da classe derivada.

3. A macro [assert] pode ser utilizada para reforçar contratos de [interface] em linguagens que não possuem mecanismos explícitos de pré e pós-condições. Ela pode tanto ser usada no contexto de ferramentas [de verificação formal] para especificar tais condições, quanto no contexto do projeto de sistemas durante a fase de [desenvolvimento].

Questão 2

Correto

Atingiu 2,5 de 2,5

1 - A função de espera pelo término de uma *thread*, *join()*, é uma [member function] da classe [Thread]. Ela foi projetada para ser utilizada por [apenas uma thread] e portanto acarretou na inclusão do atributo [_joining] na classe *Thread*, o qual é [um ponteiro para a thread em espera]. Entretanto, esta implementação [falha um assert] caso seja invocada mais de uma vez sobre o mesmo objeto.

2 - Na função *Thread::exit()*, verifica-se o ponteiro *_joining* a fim de [acordar uma eventual thread em espera], mas não verifica-se o estado da fila *_waiting* porque [a thread ainda não foi deletada] neste ponto.

Questão 3

Correto

Atingiu 2,5 de 2,5

1. Na primeira versão do código de Semaphore::p() e Semaphore::v():

```
void Semaphore::p()
{
  db(TRC) << "Semaphore::p(this=" << this << ",value=" << _value << ")" << endl;
  fdec(_value);
  while(_value < 0)
    sleep();
}

void Semaphore::v()
{
  db(TRC) << "Semaphore::v(this=" << this << ",value=" << _value << ")" << endl;
  finc(_value);
  if(_value < 1)
    wakeup();
}
```

o semáforo [funcionava corretamente se utilizado como um mutex] ✓ .

2. Na implementação de Semaphore::p() a seguir:

```
begin_atomic();
if(fdec(_value) < 1)
    sleep(); // implicit end_atomic()
else
    end_atomic();
```

sendo "value" na hora que a *thread* entra no "if" igual a 1, a *thread* [consegue] ✓ acesso ao recurso. Ainda sobre a operação de *finc()* e *fdec()*, se o "if" for falso, o incremento ou decremento [ocorre] ✓ .

Questão 4

Parcialmente correto

Atingiu 2,4 de 2,5

1. A implementação de filas no EPOS exige que os objetos a serem inseridos contenham [uma estrutura de ligação de dados (i.e. link)] .

Esta opção de implementação [reduz] o *overhead* de operações como *insert()* e *remove()*, uma vez que o Sistema Operacional [não realiza operações de alocação e liberação de memória neste caso] . Utilizando esta implementação, [o próprio objeto] fica responsável por manter controle de quantas listas podem conter o Objeto em questão em um determinado instante.

2. Na implementação de co-rotinas, uma *thread* ao concluir uma etapa de sua execução atual utiliza do método [yield()] para liberar a CPU para outras tarefas. Neste caso a tarefa em execução (que quer liberar a CPU) entra para a fila de [ready] .

3. A implementação usando a ideia de co-rotinas do método *Thread::suspend()* utiliza a seguinte verificação antes de trocar a tarefa em execução:

```
if((_running == this) && !_ready.empty())
```

Caso a verificação seja falsa, o método *Thread::idle()* é chamado, o qual, por sua vez, invoca [CPU::halt()] . Nesse caso, ao suspender-se a tarefa em execução [a próxima interrupção do timer fará com que ela volte a executar mesmo com _state = SUSPENDED] , o que [deve ser considerado um bug durante a transição até um modelo completamente bloqueante] .

4. Na implementação bloqueantes do método *Thread::join()*, um ponteiro (*_joining*) guarda o endereço da *thread* [que estava em execução no momento da chamada] , suspendendo a *thread* [em execução] até que a thread [sobre a qual o método foi aplicado] conclua sua execução. Desta forma o ponteiro *_joining* de uma *thread T* indica a *thread* [que está aguardando o término de T] .

5. No método *Thread::exit()* a verificação:

```
if(!_suspended.empty())
```

antes do trecho de código:

```
while(_ready.empty())
    idle(); // implicit unlock();
```

[impede que o sistema entre em looping] uma vez que [não surgirão novas threads prontas para executar] .

6. Em relação ao estado atual do código do nosso EPOS didático (com sincronização bloqueante), o método [sleep()] *da classe Thread é utilizado pelos mecanismos de sincronização para bloquear threads. Neste método, a Thread muda seus estado para* [WAITING] *e só volta a executar após a chamada do método* [wakeup()] .

7. O método Thread::sleep() tem como parâmetro o ponteiro para uma fila, a thread [em execução] *é inserida na fila, seu estado é setado para WAITING e o atributo* [_waiting] *da thread recebe o ponteiro da fila.*

8. Para implementação de espera bloqueante, ao entrar-se em uma fila de espera, a thread é inserida na fila e um ponteiro para fila é guardado no atributo [_joining] *da thread. Neste contexto, guardar em um atributo a fila em que a tarefa está ajuda a evitar que* [seja feito wakeup() sobre uma tarefa excluída] .

9. Qual o tamanho da pilha da função main()? [16384] *bytes. E da função philosopher()?* [16384] *bytes.*

◄ *OSDI with EPOS: Blocking Synchronization*

Seguir para...

OSDI with EPOS: Idle Thread ►