João Paulo Aramuni 11/09/2013: Todos os direitos reservados.  
  
Correção dos Algoritmos  
  
\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* Algoritmo da Porta \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*  
(Incorreto)  
  
-> Fornece Exclusão Mútua? Não.  
  
Prova Direta: Para que não forneça exclusão mútua, ambas as threads devem acessar a SC, simultaneamente.

T0 LÊ OPENDOOR = TRUE E SAI DO LAÇO  
T1 LÊ OPENDOOR = TRUE E SAI DO LAÇO  
  
T0 LÊ OPENDOOR = FALSE E ENTRA NA SC  
T1 LÊ OPENDOOR = FALSE E ENTRA NA SC  
  
Como T0 e T1 estão na SC, simultaneamente, então, o algoritmo não fornece exclusão mútua.  
  
\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* Algoritmo do Cavalheirismo \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*  
(Incorreto)  
  
-> Fornece Exclusão Mútua? Sim.  
  
Prova por contradição: Suponha-se que o algoritmo não forneça exclusão mútua:  
(T0 E T1 executando simultaneamente a SC)  
  
T0 ESTÁ NA SC = WANTCS[0] = TRUE  
T1 ESTÁ NA SC = WANTCS[0] = FALSE  
  
Como não é possível que wantCS[0] seja true e false, simultaneamente, então, o algoritmo fornece exclusão mútua.

-> Fornece Progresso? Não.  
Prova Direta: Para que não forneça progresso, ambas as threads precisam estar  
bloqueadas (em espera ocupada), competindo pelo acesso a SC, que está 'vazia'.  
  
T0 ESTÁ BLOQUEADA = WANTCS[0] = TRUE E WANTCS[1] = TRUE  
T1 ESTÁ BLOQUEADA = WANTCS[1] = TRUE E WANTCS[0] = TRUE  
  
O algoritmo permite que ambas as threads estejam bloqueadas. Em determinado  
momento, as threads não progridem em seu próprio contexto.  
Logo, o algoritmo não fornece condição de progresso.  
  
\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* Algoritmo dos Turnos Alternados \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*  
(Incorreto)

-> Fornece Exclusão Mútua? Sim.  
  
Prova por contradição: Suponha-se que o algoritmo não forneça exclusão mútua, ou seja,  
que ambas as threads podem executar a SC simultaneamente.  
  
T0 ESTÁ NA SC = TURN = 0  
T1 ESTÁ NA SC = TURN = 1  
  
Como não é possível que a variável TURN assuma dois valores distintos, simultaneamente,  
então, o algoritmo fornece exclusão mútua.  
  
-> Fornece Progresso? Sim.  
  
Prova por contradição: Suponha-se que o algoritmo não forneça progresso, ou seja, que ambas  
as threads estejam bloqueadas, disputando pelo acesso a SC.  
  
T0 ESTÁ BLOQUEADA = TURN = 1  
T1 ESTÁ BLOQUEADA = TURN = 0  
  
Como não é possível que a variável TURN assuma dois valores distintos, simultaneamente,  
então, o algoritmo fornece condição de progresso.

-> O algoritmo é ausente de inanição: Não.  
  
Prova Direta: Para que o algoritmo não seja ausente de inanição, uma thread que quer acesso   
a SC tem que estar bloqueada, faminta, enquanto a outra thread está fora da SC. (Já  
terminou de executar ou não quer mais acesso a SC).  
  
T1 ESTÁ BLOQUEADA = TURN = 0  
T0 NUNCA EXECUTA  
  
Mesmo com a SC vazia, T1 fica bloqueada se T0 nunca executar a SC.  
  
\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* Algoritmo de Peterson \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*  
(Correto para 2 threads)  
  
-> Fornece Exclusão Mútua? Sim.  
  
Prova Direta:  
T0 ESTÁ NA SC : WANTCS[1] - FALSE OU TURN = 0  
  
CASO 1) WANTCS[1] == FALSE  
T0 FAZ WANTCS[0] = TRUE  
T0 FAZ TURN = 1  
T0 LÊ WANTCS[1] == FALSE E ENTRA NA SC  
  
T1 FAZ WANTCS[1] = TRUE  
T1 FAZ TURN = 0  
T1 LÊ WANTCS[0] == TRUE E BLOQUEIA  
  
CASO 2) TURN == 0  
TO FAZ WANTCS[0] = TRUE (DEPOIS OU ANTES T1 FAZ WANTCS[1] = TRUE)  
T0 FAZ TURN = 1T1 FAZ TURN = 0  
T0 LÊ TURN = 0 E ENTRA NA SC  
  
POR ÚLTIMO, T1 LÊ WANTCS[0] == TRUE E TURN = 0 E BLOQUEIA.  
  
Como visto, nos casos possíveis em que T0 pode estar na SC, T1 é bloqueado, ou seja, o algoritmo fornece exclusão mútua.  
  
-> Fornece Progresso? Sim.  
  
Prova por contradição: Suponha-se que o algoritmo não fornece progresso, ou seja,  
que ambas as threads estejam bloqueadas, competindo pelo acesso a SC, com a SC 'vazia',   
e nenhuma das threads esteja conseguindo o acesso.  
  
T0 ESTÁ BLOQUEADA = WANTCS[1] = TRUE E TURN = 1  
T1 ESTÁ BLOQUEADA = WANTCS[0] = TRUE E TURN = 0  
  
Como não é possível que TURN assuma dois valores distintos, simultaneamente, então,  
o algoritmo fornece condição de progresso.  
  
-> O algoritmo é ausente de inanição? Sim.  
  
Prova por contradição: Suponha-se que o algoritmo não seja ausente de inanição, uma thread que quer acesso   
a SC, tem que estar bloqueada, faminta, enquanto a outra thread está fora da SC. (Já terminou de   
executar ou não quer mais acesso a SC).  
  
T0 ESTÁ BLOQUEADA = WANTCS[1] = TRUE E TURN = 1  
T1 ESTÁ FORA DA SC = WANTCS[1] = FALSE  
  
Como não é possível que WANTCS[1] assuma dois valores distintos, simultaneamente, então,  
o algoritmo é ausente de inanição.  
  
\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* Algoritmo de Dekker \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*  
(Correto para 2 threads)  
  
-> Fornece Exclusão Mútua? Sim.  
  
Prova Direta:  
  
CASO 1) T0 NA SC  
TURN = 1  
T0 SOLICITA ACESSO A SC  
J = 1 - 0 = 1  
wantCS[ 0 ] = true  
T0 TESTA wantCS[ 1 ] = false  
T0 ENTRA NA SC  
  
T1 SOLICITA ACESSO A SC  
J = 1 - 1 = 0  
wantCS[ 1 ] = true  
T1 TESTA wantCS [ 0 ] = true e entra no laço  
T1 TESTA TURN = J que retorna falso (1 == 0)  
T1 é bloqueada em esperado ocupada.  
  
CASO 2) T1 NA SC  
TURN = 1  
T1 SOLICITA ACESSO A SC  
J = 1 - 1 = 0  
wantCS[ 1 ] = true  
T1 TESTA wantCS[ 0 ] = false e sai do laço  
T1 ENTRA NA SC  
  
T0 SOLICITA ACESSO A SC  
J = 1 - 0 = 1  
wantCS [ 0 ] = true  
T0 TESTA wantCS[ 1 ] = true e entra no laço   
T0 TESTA TURN = J = 1 e entra no bloco do IF  
wantCS [ 0 ] = false  
T0 TESTA TURN == J e é bloqueada em espera ocupada  
  
Como visto, sempre que uma thread entrar na SC, a outra thread será bloqueada. Logo, o algoritmo fornece exclusão mútua.  
  
-> Fornece Progresso? Sim.  
  
Prova por contradição: Suponha-se que o algoritmo não forneça progresso, ou seja,  
ambas as threads querem acesso a sessão crítica wantCS[0] == true e wantCS[1] == true, porém,   
ficam bloqueadas e nenhuma delas consegue acesso a SC.  
  
T0 ESTÁ BLOQUEADA => TURN = 0 E WANTCS[1] == TRUE  
T1 ESTÁ BLOQUEADA => TURN = 1 E WANTCS[0] == TRUE  
  
Como não é possível que a variável turn assuma dois valores distintos, simultaneamente, então,   
o algoritmo fornece condição de progresso.   
  
Para que as 2 threads ficassem bloqueadas disputando a sessão crítica, o teste if(turn ==j) teria que retornar falso em ambas execuções. Para isso, a variável turn teria que valer 0 durante a execução de T0 (0 == 1) e 1 durante a execução de T1 (1 == 0), simultaneamente, o que é impossível.  
  
-> O algoritmo é ausente de inanição? Sim.  
  
Prova por contradição: Suponha-se que o algoritmo não seja ausente de inanição, mesmo que uma thread T1 não queira mais acesso a sessão critica, ou simplesmente tenha terminado sua execução, uma thread T0 fica inibida de acessar a   
sessão crítica. T0 permanece bloqueada e por isso, fica faminta:  
  
T0 ESTÁ BLOQUEADA = WANTCS[1] == TRUE  
T1 ESTÁ FORA DA SC = WANTCS[1] == FALSE  
  
Para que T0 fique em espera ocupada, bloqueada e faminta, primeiramente a condição do primeiro laço,   
while (wantCS[ j ] == true) deve ser atendida. Porém, wantCS[ j ] será falso, uma vez que a thread T1   
não quer acesso a sessão crítica. Como não é possível wantCS[ 1 ] ser true e falso simultaneamente, então,   
o algoritmo é ausente de inanição. T0 não fica faminta e consegue acessar sua sessão crítica.  
  
\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* Algoritmo de Lamport (da Padaria) \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*  
(Correto para N threads)  
  
Primeiro algoritmo correto para múltiplas threads.  
As provas necessitam dos lemas encontrados no material didático.  
  
DOORWAY (PORTAL) 1ºParte  
N -> quantidade de threads  
Tem 2 loops espera ocupada. Busy wait.  
Nj < Ni ou (Nj - Ni e j < 1)  
Notação: (Nj, j) < (Ni, i)  
  
A thread i vence a thread j se  
Nj = 0 ou (Ni, i) < (Nj, j)  
[2ºcaso] [1º caso Vitória legítima]  
  
2º caso; Não está interessada na sessão crítica mas pode ficar.  
Se ficar interessada, entra o papel do choosing, que define um bloco  
true e false.  
  
Timestamps = senhas  
  
Nesse algoritmo, timestamps podem crescer sem limites.  
No caso de variáveis inteiras, corre o risco de estourar o espaço máximo reservado.  
  
Prova que fornece exclusão mútua:  
Lemas: 1,5,6  
  
Por contradição:  
Suponha-se por um absurdo, que duas threads estejam na sessão crítica.  
(Ni, i) < Nj, j) e (Nj, j) < (Ni, i)  
Isso deve ocorrer para que as threads estejam simultaneamente na sc.   
Como isso é impossível, então, fornece exclusão mútua.  
  
Provar que permite progresso:  
Lemas 1 e 4  
Provar a ausência de inanição  
Lema 3 [prova direta]  
  
Toda thread T, competindo pelo acesso a SC, terá o menor número de senha.  
  
\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*