Exercícios de coordenação

Exercícios de coordenação usando *threads* Posix e semáforos (ou *mutexes*). A página de exemplos de exclusão mútua contém exemplos de código que podem ser usados como ponto de partida.

Ordem estrita

Considerando 5 *threads* lançadas simultaneamente com suas respectivas funções definidas abaixo, use semáforos para garantir que a saída na tela seja sempre `U O I E A`, nessa ordem:

```
Task A          Task E          Task I          Task O          Task U
{               {               {               {               {
  printf("A")     printf("E")     printf("I")     printf("O")     printf("U")
  task_exit()     task_exit()     task_exit()     task_exit()     task_exit()
}               }               }               }               }
```

Robôs Simpsons

Suponha três robôs (*Bart*, *Lisa*, *Maggie*), cada um controlado por sua própria *thread*. Escreva o código dessas *threads*, usando semáforos para garantir que os robôs avancem sempre na mesma sequência:

```
Bart
    Lisa
        Maggie
    Lisa
Bart
    Lisa
        Maggie
    Lisa
...
```

Evite situações de espera ocupada (*busy wait*).

Barreiras

Uma barreira é um operador de sincronização forte entre N tarefas, em que elas esperam até que todas as tarefas cheguem à barreira. Barreiras são muito usadas em ambientes de programação paralela, como OpenMP.

O exemplo a seguir ilustra o uso de uma barreira:

```
barrier_init (&b, 3) ; // define uma barreira para 3 tarefas

Task A
{
  printf ("A antes\n") ;
  barrier_wait (&b) ;  // espera na barreira
  printf ("A depois\n") ;
}
Task B
{
```

```
  printf ("B antes\n") ;
  barrier_wait (&b) ;  // espera na barreira
  printf ("B depois\n") ;
}

Task C
{
  printf ("C antes\n") ;
  barrier_wait (&b) ;  // espera na barreira
  printf ("C depois\n") ;
}
```

A execução do código deve gerar uma saída na qual todas as operações “antes” ocorrem antes das operações “depois”. Por outro lado, não há nenhuma restrição na ordem entre as operações “A”, “B” e “C”. Uma saída possível seria a indicada abaixo:

```
A antes
C antes
B antes
-- (barreira)
C depois
B depois
A depois
```

Defina `struct barrier_t`, implemente as operações `barrier_init (barrier_t *b, int num)` e `barrier_wait (barrier_t *b)` e use-as para implementar o exemplo acima. Use semáforos ou mutexes e evite soluções que usem espera ocupada (*busy wait*).

A solução correta deste problema exige **dois semáforos**: um para “entrar” na barreira e outro para “sair” dela“.

Banheiro Unissex

Este problema foi adaptado do Pequeno Livro de Semáforos (que tem a solução, mas tente resolver sem consultá-la!).

Em um departamento de informática com apenas um banheiro disponível[1)], a chefia decidiu torná-lo multiusuário e unissex, com as seguintes restrições:

- Homens e mulheres não podem usar simultaneamente o banheiro.
- Não pode haver mais de 3 pessoas usando o banheiro ao mesmo tempo.
- Obviamente, não podem ocorrer impasses.

Usando *threads* e semáforos, escreva o código necessário para garantir essas restrições. As *threads* têm o seguinte comportamento (sem a parte de coordenação):

```
task homem
{
  while (true)
  {
    trabalhar () ;
    usar_banheiro_h () ;
  }
```

```
}
task mulher
{
  while (true)
  {
    trabalhar () ;
    usar_banheiro_m () ;
  }
}
```

Jantar dos selvagens

Este problema foi adaptado do Pequeno Livro de Semáforos (que tem a solução, mas tente resolver sem consultá-la!).

No jantar comunal de uma tribo remota, um cozinheiro serve ensopado em um caldeirão, que tem capacidade para *N* porções. Cada convidado se serve de uma porção de ensopado; caso o caldeirão fique vazio, o cozinheiro deve ser chamado para enchê-lo novamente, enquanto os convidados esperam para se servir. Enquanto não tiver o que fazer, o cozinheiro dorme.

Código do cozinheiro (sem a parte de coordenação):

```
task cozinheiro()
{
  while (true)
    enche_caldeirao(N) ;
}
```

Código dos convidados (idem):

```
task convidado()
{
  while (true)
  {
    serve_porção() ;
    come() ;
  }
}
```

A granja de ovos

Em uma granja, 30 galinhas produzem ovos. Cada galinha bota um ou dois ovos por vez; todos os ovos produzidos pelas galinhas caem diretamente em uma cesta de ovos com capacidade infinita (nenhum ovo quebra).

Os ovos que caem na cesta são coletadas por 5 fazendeiros. Cada fazendeiro pega uma bandeja com capacidade para 6 ovos e espera. Quando houver ovos suficientes na cesta, o fazendeiro pega os ovos e enche sua bandeja; em seguida, pega uma nova bandeja vazia e recomeça o ciclo.

Modelando cada galinha e cada fazendeiro com sua própria *thread*, escreva o código de sincronização para NG galinhas, NF fazendeiros e bandejas com capacidade para NB ovos cada.

Sugestão de pseudocódigo das galinhas e dos fazendeiros:

```
task galinha ()
{
  while (true)
  {
    // coloca um ou dois ovos na cesta
    novos_ovos = 1 + random() % 2 ;
    ovos_cesta += novos_ovos ;

    // se há NB ou mais ovos na cesta, avisa um fazendeiro
    if (ovos_cesta >= NB)
      sinaliza condição "cesta tem ovos"

    sleep (1) ;
  }
}

task fazendeiro ()
{
  while (true)
  {
    // se não houver ovos suficientes na cesta, espera
    while (ovos_cesta < NB)
      espera condição "cesta tem ovos"

    // retira NB ovos da cesta
    ovos_cesta -= NB ;

    sleep (1)
  }
}
```

A solução usando variáveis de condição é **bem mais simples** que usando semáforos.

[1)]
Esta é uma situação totalmente hipotética; qualquer semelhança com situações reais ~~não~~ é mera coincidência...