Lista de Exercícios - Sistemas Operacionais
Aula 5: *Comunicação entre Processos*

Professores: Valmir C. Barbosa e Felipe M. G. França
Assistente: Alexandre H. L. Porto

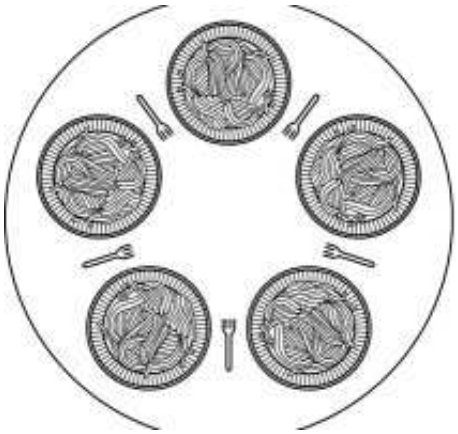
1. Considere um computador que não tenha uma instrução TEST AND SET LOCK (TSL) mas tenha uma instrução para comutar o conteúdo de um registrador e de uma palavra da memória em uma única ação indivisível. É possível utilizar essa instrução para escrever uma rotina **ENTER_REGION** como a reproduzida a seguir (vista na Aula 5)?

```
ENTER_REGION:
    TSL REGISTER, LOCK
    CMP REGISTER, #0
    JNE ENTER_REGION
    RET
```

Resp.: Sim, pois podemos usar a variável LOCK como um sinalizador, com dois possíveis valores: 0, indicando que o processo pode executar a sua seção crítica, e 1, indicando que o processo não pode executar a sua seção crítica. Com isso, bastará que o registrador, no caso o REGISTER, seja igual a 1 antes de executarmos a instrução, que poderia se chamar, por exemplo, SWAP, como no código dado a seguir. Assim como antes, o processo, ao sair da seção crítica, deverá executar o procedimento **LEAVE_REGION**, que colocará um valor 0 no sinalizador, para permitir que outros processos possam executar a sua seção crítica.

```
ENTER_REGION:
    MOVE REGISTER, #1
    SWAP REGISTER, LOCK
    CMP REGISTER, #0
    JNE ENTER_REGION
    RET
```

2. Um problema clássico da comunicação de processos é o problema dos filósofos. Neste problema, os filósofos estão dispostos em torno de uma mesa circular e cada filósofo passa por períodos alternados de comer e de pensar. Sobre a mesa existe um prato de espaguete à frente de cada filósofo e um garfo entre dois pratos consecutivos, como podemos ver na figura a seguir, em que temos 5 filósofos (e 5 garfos e pratos de espaguete):



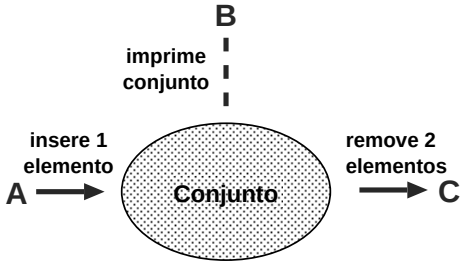
Quando um filósofo está com fome, ele tenta pegar os garfos à sua esquerda e à sua direita, pois o espaguete está muito escorregadio e por isso são necessários dois garfos para comê-lo. Ao conseguir pegar os garfos, o filósofo come por um tempo um pouco do espaguete e depois recoloca os garfos na mesa e volta a pensar.

A seguir é dada uma solução para o problema com n filósofos baseada em n semáforos binários, sendo que o semáforo $fork[i]$ está associado ao garfo i, $1 \leq i \leq n$. Os semáforos são todos inicializados com o valor 1. Esta solução funcionará se os processos que implementam os filósofos executarem a função *philosopher* dada a seguir, sendo que cada um destes processos está associado a um identificador i, $1 \leq i \leq n$? Justifique a sua resposta.

```
void philosopher(int i)
{
  while (1) {
    pensar();
    P(fork[i]);
    P(fork[(i+1) % n]);
    comer();
    V(fork[i]);
    V(fork[(i+1) % n]);
  }
}
```

Resp.: A solução não funcionará, pois podemos ter um impasse, como veremos a seguir. Suponha que todos os processos não são bloqueados ao executarem a primeira operação **P** do procedimento *philosopher*, isto é, a segunda operação **P** somente será executada após todos os processos executarem a primeira operação **P**. Neste caso, depois de cada processo executar a primeira operação **P**, o valor de cada um dos semáforos binários $fork[i]$, $1 \leq i \leq n$, será igual a 0. Com isso, todos os processos serão bloqueados ao executarem a segunda operação **P**, e nunca mais serão desbloqueados, pois nenhum deles poderá executar a operação **V** sobre um dos semáforos binários, e somente a execução desta operação poderá desbloquear um dos processos.

3. O conjunto dado na figura a seguir, que pode armazenar até n elementos, é compartilhado por três processos, A, B e C. O processo A sempre coloca um elemento no conjunto, o processo B sempre imprime o conteúdo atual do conjunto e o processo C sempre remove dois elementos do conjunto. Supondo que a estrutura de dados que representa o conjunto somente armazene os elementos do conjunto, como os semáforos podem ser usados para garantir o correto funcionamento dos processos A, B e C? Justifique a sua resposta.



Resp.: Para garantir o correto funcionamento dos processos A, B e C, isto é, a sua correta sincronização, vamos precisar de três semáforos, sendo que dois deles, *vazio* e *cheio*, são de contagem, e o terceiro, *acesso*, é binário. O semáforo binário *acesso* é usado para garantir o acesso exclusivo ao conjunto, e é inicializado com o valor 1, pois nenhum processo ainda está executando. O semáforo *vazio* é usado para contar o espaço disponível no conjunto para a inserção de novos elementos, e é usado para bloquear o processo A se o conjunto estiver cheio, isto é, quando o conjunto não possuir espaço disponível para mais elementos. O semáforo *cheio* conta o número de elementos no conjunto, e é usado para bloquear o processo C caso o conjunto esteja vazio, isto é, se não possuir elementos. Os valores iniciais dos semáforos *vazio* e *cheio* dependem do número de elementos no conjunto antes da execução dos processos A e C. Se o número de elementos for m, então temos um espaço disponível de $n - m$ no conjunto e, portanto, o valor inicial do semáforo *cheio* é m e o valor inicial do semáforo *vazio* é $n - m$. O processo A deverá usar a função $InserirElemento(e)$ dada a seguir para inserir o elemento e no conjunto. Já o processo B deverá usar a função $ImprimirConjunto(\textbf{void})$ a seguir para imprimir os elementos do conjunto. Finalmente, o processo C deverá usar a função $RemoverElementos(e_1, e_2)$ dada a seguir para remover dois elementos e_1 e e_2 do conjunto. Como o processo C precisará de dois elementos do conjunto note, no código para a função *RemoverElementos*, que precisaremos executar duas vezes a operação **P** sobre *cheio*, pois removeremos dois elementos. Além disso, precisaremos executar a operação **V** também duas vezes sobre *vazio*, pois duas novas posições estarão disponíveis após a remoção destes elementos. Observe também que a função *ImprimeConjunto* somente precisará usar o semáforo *acesso*, pois somente imprime os elementos do conjunto.

```
void InserirElemento(e)
{
  P(vazio);
  P(acesso);
  // Código para inserir o elemento e no conjunto.
  V(acesso);
  V(cheio);
}
```

void *ImprimirConjunto*(**void**)
{
  P(*acesso*);
  // Código para imprimir todos os elementos no conjunto.
  V(*acesso*);
}

void *RemoverElementos*(e_1, e_2)
{
  P(*cheio*);
  P(*cheio*);
  P(*acesso*);
  // Código para remover dois elementos do conjunto e depois
  // colocá-los em e_1 e e_2.
  V(*acesso*);
  V(*vazio*);
  V(*vazio*);
}

4. Suponha que dois processos, A e B, compartilhem uma pilha, usada para armazenar números. Suponha ainda que esta pilha, inicialmente vazia, possa armazenar até n números, e que não exista uma variável para contabilizar a quantidade de números armazenados na pilha. O processo A continuamente insere números na pilha. Já o processo B continuamente remove dois números da pilha, calcula a soma destes números, e depois insere o resultado da soma na pilha. Como os semáforos devem ser usados para garantir a correta execução dos processos A e B? Justifique a sua resposta.

   Resp.: Como não existem variáveis para contabilizar a quantidade de números armazenados na pilha, precisaremos de um semáforo de contagem *cheia*, que contará a quantidade de números na pilha e será usado para bloquear o processo B quando a pilha estiver vazia. Também precisaremos de um semáforo de contagem *vazia*, que contará a quantidade de números que ainda podem ser inseridos na pilha e será usado para bloquear o processo A quando a pilha estiver cheia. E também precisaremos de um semáforo binário *acesso*, usado para garantir o acesso exclusivo à pilha. Como inicialmente a pilha está vazia e os processos não estão em execução, então o valor inicial de *cheia* será 0, o valor inicial de *vazia* será n e o valor inicial de *acesso* será 1. A seguir

mostramos dois possíveis procedimentos que podem ser executados pelos processos A e B. Os procedimentos da sua resposta não precisam ser exatamente iguais aos dados a seguir, mas precisam, para estarem corretos, de: (i) possuírem um laço infinito como os destes procedimentos; (ii) garantirem o acesso exclusivo à pilha; e (iii) garantirem que o processo A não coloque números em uma pilha cheia e que o processo B não retire números de uma pilha vazia.

```
void ProcessoA()
{
  while (1)
  {
    P(vazia);
    P(acesso);
    // Código para colocar um número na pilha.
    V(acesso);
    V(cheia);
  }
}

void ProcessoB()
{
  while (1)
  {
    P(cheia);
    P(cheia);
    P(acesso);
    // Código para remover dois números da pilha, calcular a sua
    // soma, e depois colocar esta soma na pilha.
    V(acesso);
    // Note que somente uma operação V sobre vazia é necessária,
    // porque colocamos a soma dos números na pilha.
    V(vazia);
    V(cheia);
  }
}
```