Sistemas Operacionais – Prof. Rafael R. Obelheiro

Lista de Exercícios — Comunicação Interprocessos

1. **[Silberschatz 1994, 6.5]** Uma das formas de garantir exclusão mútua em uma seção crítica é desabilitando interrupções. Entretanto, isso pode afetar o relógio do sistema. Explique os motivos pelos quais isso ocorre e o que pode ser feito para minimizar o problema.

2. **[Stallings 1998, 5.4]** A espera ocupada é sempre menos eficiente (em termos de utilização do processador) do que uma espera bloqueante? Explique.

3. Considere um programa concorrente com duas *threads*, `T1` e `T2`, mostradas no código abaixo. `F`, `A`, `J`, `Q` e `K` são instruções atômicas (indivisíveis) quaisquer.

```
void T1() {
   F;
   A;
}
```

```
void T2() {
   J;
   Q;
   K;
}
```

   Determine todas as possíveis sequências de execução das *threads* acima (mostre as diferentes sequências de instruções atômicas que podem ocorrer), considerando que essas *threads* são disparadas simultaneamente (ou seja, não é possível prever qual delas inicia começa a executar primeiro).

4. **[Oliveira 2004, 3.3]** Descreva o erro na implementação do produtor-consumidor mostrada abaixo. Crie uma sequência de eventos que termina em algum comportamento indesejado para o programa.

```
struct tipo_dado buffer[N];
int proxima_insercao = 0;
int proxima_remocao = 0;
...
semaphore exclusao_mutua = 1;
semaphore espera_vaga = N;
semaphore espera_dado = 0;
...
void produtor(void)
{  ...
   down(&exclusao_mutua);
   down(&espera_vaga);
   buffer[proxima_insercao] = dado_produzido;
   proxima_insercao = (proxima_insercao + 1) % N;
   up(&exclusao_mutua);
   up(&espera_dado);
   ...
}
...
void consumidor(void)
{  ...
   down(&exclusao_mutua);
   down(&espera_dado);
   dado_a_consumir = buffer[proxima_remocao];
   proxima_remocao = (proxima_remocao + 1) % N;
   up(&exclusao_mutua);
   up(&espera_vaga);
   ...
}
```

5. Suponha que a biblioteca do seu compilador C possui uma instrução `swap(&v1, &v2)`, que troca **atomicamente** o conteúdo de duas variáveis `v1` e `v2`. Mostre como implementar funções `enter_region()` e `leave_region()` para garantir exclusão mútua em regiões críticas usando a função `swap()` e uma variável compartilhada `lock`. Não se preocupe em eliminar espera ocupada da sua solução.

   DICA: você pode se basear na solução usando TSL para resolver este problema.

6. Considere um programa concorrente com dois processos, `X` e `Y`, mostrados abaixo. A variável `S` é um semáforo compartilhado pelos dois processos, que é inicializado em 0.

```
semaphore S = 0;     /* semáforo compartilhado entre X e Y */
```

```
void X() {
   int i;
   for (i = 0; i < 4; i++) {
      printf("a");
      down(&S);
   }
}
```

```
void Y() {
   int j;
   for (j = 0; j < 4; j++) {
      printf("b");
      up(&S);
   }
}
```

Determine quais das *strings* abaixo podem ser geradas pelo programa acima e quais delas não podem:

(a) `abbabbaa`
(b) `abbabaab`
(c) `aabbabab`
(d) `baaabbab`
(e) `bbbaaaab`
(f) `aaabbbba`

7. Considere o problema da alocação de armários de aço na biblioteca da UDESC. Existe um número fixo *NARM* de armários. Normalmente, um usuário que chega à biblioteca e deseja um armário solicita ao funcionário responsável a chave de um dos armários desocupados. Caso não haja um armário livre, o usuário espera em uma fila até que algum armário seja desocupado. Para eliminar a necessidade de um funcionário para a distribuição de chaves, um aluno de SOP desenvolveu o seguinte algoritmo, baseado em `sleep()` e `wakeup()`:

```
1  int armarios[NARM], ocupados = 0;
2
3  void usuario(int i)
4  {
5     int a;              /* número do armário ocupado pelo usuário */
6
7     if (ocupados == NARM)
8        sleep(ocupados);
9     ocupados++;
10    for (a = 0; (a < NARM) && (armarios[a] != 0); a++)
11       ;                /* percorre o vetor até achar um armário livre */
12    armarios[a] = i;
13    estuda();
14    armarios[a] = 0;
15    ocupados--;
16    if (ocupados == NARM-1)
17       wakeup(ocupados);
18 }
```

Esse algoritmo resolve o problema corretamente? Quais erros você consegue apontar nessa solução, se é que eles existem? Demonstre o(s) erro(s) mostrando sequências de execução que causem problemas.

Obs.: O uso de `ocupados` como parâmetro de `sleep()` e `wakeup()` não constitui um erro. Nesse caso, `ocupados` é usado para "casar" um `wakeup()` com o `sleep()` correspondente (ou seja, `wakeup(ocupados)` acorda um processo dentre os que executaram `sleep(ocupados)`).

8. Resolva o problema de alocação de armários do exercício anterior usando semáforos.

9. O programa abaixo gerencia um estacionamento de automóveis. O estacionamento possui `ANDARES` andares com `MAXVAGAS` vagas em cada um. Existem múltiplas cancelas para entrada e saída, e cada cancela executa o programa abaixo, sendo que as informações referentes às vagas de estacionamento disponíveis são compartilhadas por todas as cancelas. Um carro que chegue ao estacionamento provoca a execução da função `libera_ticket()`, que localiza uma vaga no estacionamento e fornece um tíquete ao usuário informando o andar e a vaga em que ele deverá estacionar (caso não haja vagas disponíveis, é aberta uma cancela de escape para que o carro possa sair). Quando um carro sai do estacionamento, o motorista insere seu tíquete na máquina, o que causa a execução da função `retorna_ticket()`; esta função deve então liberar no sistema a vaga que estava sendo ocupada por esse carro e abrir a cancela de saída.

   (a) Identifique as regiões críticas no código (use os números de linhas).

   (b) Mostre ao menos uma sequência de execução que pode provocar um comportamento indesejado.

   (c) Use semáforos para garantir exclusão mútua nas regiões críticas.

```
1  typedef struct { int andar, vaga; } ticket_t;
2
3  carro_t vagas[ANDARES][MAXVAGAS];        /* matriz de ocupacao de vagas */
4
5  int ocupadas = 0;                              /* no. de vagas ocupadas */
6
7  ticket_t libera_ticket(carro_t carro)
8  {
9     int a, v;
10    ticket_t ticket;
11    if (ocupadas < ANDARES*MAXVAGAS) {      /* existem vagas disponíveis? */
12       for (a = 0; a < ANDARES; a++) {      /* percorre o mapa de vagas */
13          for (v = 0; v < MAXVAGAS; v++) {
14             if (vagas[a][v] == LIVRE) {    /* se encontrou uma vaga livre... */
15                ocupadas++;                 /* contabiliza a ocupação da vaga */
16                vagas[a][v] = carro;        /* ajusta o mapa de vagas */
17                ticket.andar = a;
18                ticket.vaga = v;
19                imprime_ticket(ticket);     /* imprime o ticket para o motorista */
20                libera_entrada();           /* abre a cancela de entrada */
21                return ticket;
22             }
23          }
24       }
25    }
26    libera_escape();           /* estacionamento cheio, abre cancela de escape */
27    return ticket;
28 }
29
30 void retorna_ticket(ticket_t ticket)
31 {
32    ocupadas--;                                    /* contabiliza a liberação da vaga */
33    vagas[ticket.andar][ticket.vaga] = LIVRE; /* ajusta o mapa de vagas */
34    libera_saida();                                /* abre a cancela de saída */
35 }
```

10. Suponha um estabelecimento que realiza fotocópias. No estabelecimento existem `NA` atendentes e `NC` máquinas copiadoras. Um cliente chega no estabelecimento, espera por um atendente disponível, solicita ao atendente as suas cópias, paga pelo serviço e vai embora. Cada atendente espera a chegada de um cliente, recebe a solicitação de serviço, tira as cópias e recebe o pagamento, repetindo o procedimento para os novos clientes que chegam no estabelecimento. Todas as copiadoras são iguais, e cada atendente realiza apenas um serviço e opera apenas uma copiadora de cada vez.

   (a) Escreva um algoritmo que use semáforos para sincronizar clientes e atendentes, supondo que existe um número suficiente de copiadoras para todos os atendentes.

   (b) Modifique o algoritmo anterior para tratar a situação em que o número de copiadoras pode ser inferior ao número de atendentes, ou seja, um atendente pode ter que esperar até que uma copiadora esteja livre.

11. **[Oliveira 2004, 3.8]** O problema dos leitores/escritores consiste de um texto que pode ser lido ou escrito por vários processos. Considerando o código abaixo, responda justificando:

    (a) É possível vários leitores lerem ao mesmo tempo?

    (b) É possível vários escritores escreverem ao mesmo tempo?

    (c) É possível postergação indefinida de um leitor? (A postergação indefinida de um processo ocorre quando ele pode ser impedido de entrar em uma seção crítica por um tempo ilimitado, enquanto outros processos continuam entrando e saindo dessa seção crítica.)

    (d) É possível postergação indefinida de um escritor?

    (e) É possível leitores e escritores acessarem ao mesmo tempo?

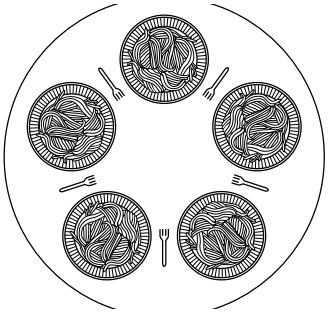
```
int nl = 0;
semaphore tipo = 1;
semaphore exclusivo = 1;

void leitor(void)
{  ...
   down(&exclusivo);
   if (nl > 0) ++nl;
   else {
      down(&tipo);
      nl = 1;
   }
   up(&exclusivo);
   acessa_texto();
   down(&exclusivo);
   --nl;
   if (nl == 0) up(&tipo);
   up(&exclusivo);
   ...
}
```

```
void escritor(void)
{  ...
   down(&tipo);
   acessa_texto();
   up(&tipo);
   ...
}
```

12. Com base no código mostrado no exercício anterior, resolva o problema dos leitores/escritores eliminando postergação indefinida.

13. Um dos problemas de sincronização mais conhecidos é o "jantar dos filósofos", proposto por Dijkstra em 1965. Nesse problema, cinco filósofos sentam-se ao redor de uma mesa circular. Cada filósofo tem diante de si um prato de macarrão, e entre cada par de pratos existe um garfo (veja a figura abaixo). Um filósofo pode estar meditando ou comendo. Quando um filósofo deseja comer, ele tenta pegar dois garfos (é impossível comer o macarrão com um único garfo), o que está à sua direita e o que está à sua esquerda, em qualquer ordem, um de cada vez (não é possível pegar os dois garfos ao mesmo tempo). Se o filósofo consegue pegar os garfos, ele come durante um tempo e depois larga os garfos na mesa. O problema consiste em escrever um programa (**diferente do apresentado no livro do Tanenbaum**) que represente o comportamento dos filósofos, e que não entre em impasse (*deadlock*). (Dica: considere a possibilidade de limitar o número de filósofos que tentam comer ao mesmo tempo.)



14. **[Stevens 1998, 5.3]** Algumas linguagens de programação oferecem suporte a concorrência por meio de um par de comandos `parbegin`...`parend`. Seja, por exemplo, o trecho de código abaixo:

```
parbegin
  P1;
  P2;
  P3;
parend
```

O uso de `parbegin` e `parend` indica que `P1`, `P2` e `P3` executam em paralelo, e podem ser escalonados em qualquer ordem (ou seja, `P3` pode começar a executar antes de `P1` e/ou de `P2`).

Com base nisso, considere o programa abaixo:

```
#define N 50

int soma;

void total(void) {
   int cont;
   for (cont = 1; cont <= N; cont++)
      soma++;
}

main() {
   soma = 0;
   parbegin
      total(); total();
   parend
   printf("A soma eh %d\n", soma);
}
```

(a) Determine os limites inferior e superior para o valor final da variável compartilhada `soma` exibido por esse programa concorrente. Suponha que os processos podem executar em velocidades relativas quaisquer e que uma operação de incremento `X++` é implementada por três instruções de máquina:

```
MOVE X, ACC      ; copia X para o acumulador
INC  ACC         ; incrementa o acumulador
MOVE ACC, X      ; copia o acumulador para X
```

(b) Suponha que um número arbitrário desses processos podem executar segundo as premissas da parte (a). Qual o efeito de tal modificação na faixa de valores finais de `soma`?