Sistemas Operacionais – Prof. Rafael R. Obelheiro

Lista de Exercícios — Comunicação Interprocessos — Respostas

1. A desabilitação de interrupções pode afetar o relógio do sistema porque as interrupções de relógio é que permitem ao SO perceber a passagem do tempo. As interrupções de relógio têm um período fixo, e a cada interrupção o SO incrementa um contador de *ticks*. A passagem do tempo é representada pela contagem de *ticks* multiplicada pelo período do relógio; por exemplo, se o período do relógio for de 20 ms, cinco *ticks* representam 100 ms. Se as interrupções estiverem desabilitadas quando uma interrupção de relógio ocorrer, o SO não irá incrementar o contador.

   Para minimizar a interferência no relógio, deve-se restringir a desabilitação de interrupções a intervalos curtos (muito menores que o período do relógio).

2. A espera ocupada geralmente é menos eficiente que espera bloqueada, pois consume CPU inutilmente. Além disso, o processo que espera ocupa a CPU ou compete por ela, prejudicando o processo que pode satisfazer a condição de espera, o que é especialmente ruim em sistemas monoprocessados. No entanto, quando um processo encontra a condição de espera já satisfeita, a espera ocupada acaba sendo mais rápida do que uma chamada bloqueante, que envolve o *kernel* do SO.

3. As 10 sequências de instruções possíveis são as seguintes:

   1) `FAJQK`
   2) `FJAQK`
   3) `FJQAK`
   4) `FJQKA`
   5) `JQKFA`
   6) `JQFKA`
   7) `JQFAK`
   8) `JFQKA`
   9) `JFQAK`
   10) `JFAQK`

4. O erro está na ordem de `down()` no produtor e no consumidor, que permite que um dos processos bloqueie dentro da região crítica, levando a um *deadlock*. Por exemplo, suponha que o buffer está vazio e a sequência de execução seja esta:

produtor	consumidor
	`down(&exclusao_mutua)` $(1 \rightarrow 0)$
	`down(&espera_dado)` $(0 \rightarrow -1)$ – **bloqueia**
`down(&exclusao_mutua)` $(0 \rightarrow -1)$ – **bloqueia**	

   Com os dois processos bloqueados, temos um *deadlock*.

5. A diferença para a solução usando TSL está em `enter_region()`, que usa uma variável local (`old_lock`), setada explicitamente para 1, no lugar do registrador.

```
1  void enter_region(void) {
2     int old_lock;
3     do {
4        old_lock = 1;
5        swap(&lock, &old_lock);
6     } while (old_lock == 1);
7  }
8
9  void leave_region(void) {
10    lock = 0;
11 }
```

6. O semáforo `S` garante que cada *string* tenha no máximo uma letra "`a`" a mais do que o número de letras "`b`" precedentes. Assim:

   (a) válida
   (b) válida
   (c) inválida
   (d) inválida
   (e) válida
   (f) inválida

7. Existem basicamente dois problemas com o algoritmo apresentado:

   (1) O primeiro problema está no uso de `sleep()` e `wakeup()`, que está sujeito a uma condição de disputa, como no seguinte exemplo de execução indesejável:
      I. Todos os armários estão ocupados;
      II. O usuário X chega na linha 7, encontra `ocupados == NARM`, e perde o processador entre as linhas 7 e 8;
      III. O usuário Y, que estava ocupando um armário, sai da função `estuda()` e executa as linhas 14–17 – como o usuário X ainda não executou `sleep()`, o `wakeup()` da linha 17 será perdido;
      IV. O usuário X retoma o processador e executa a linha 8, ficando bloqueado pelo menos até que todos os armários sejam novamente ocupados e um outro usuário execute a linha 17.

   (2) O segundo problema é que existe uma condição de disputa no acesso ao vetor de alocação de armários (`armarios`). Um exemplo de execução indesejável seria este:
      I. Todos os armários estão livres;
      II. O usuário 1 chega até a linha 10 e encontra `armarios[0] == 0`, mas perde o processador entre as linhas 10 e 11;
      III. O usuário 2 começa a executar o algoritmo, encontra `armarios[0] == 0` na linha 10, e ocupa o armário na linha 11 (fazendo `armarios[0] = 2`);
      IV. O usuário 1 retoma o processador e ocupa o armário 0 na linha 11 (`armarios[0] = 1`).

   Nesse momento, `armarios` e `ocupados` estão em estados inconsistentes: `ocupados` vale 2, mas apenas o armário 0 está alocado no vetor, para o usuário 2 (a alocação do usuário 1 foi perdida). Além disso, o primeiro usuário (1 ou 2) que sair da biblioteca liberará o armário 0 na linha 14.

8. O problema tem:

   (a) uma condição de sincronização: um usuário só pode ocupar um armário quando houver pelo menos um armário livre. Para garantir essa condição, é usado um semáforo contador `livres`, que representa o número de armários livres.

   (b) necessidade de exclusão mútua no acesso ao estado compartilhado, que no caso é o vetor `armarios`. No algoritmo mostrado no exercício anterior, esse acesso ocorre nas linhas 10–12 (primeira região crítica) e na linha 14 (segunda região crítica). Para garantir a exclusão mútua, é usado um semáforo binário `mutex` para guardar o acesso às regiões críticas.

```
1  int armarios[NARM] = 0;
2  semaphore livres = NARM;
3  semaphore mutex = 1;
4
5  void usuario(int i)
6  {
7     int a;              /* número do armário ocupado pelo usuário */
8
9     down(&livres);      /* espera por um armário livre */
10    down(&mutex);       /* acessa a região crítica */
11    for (a = 0; (a < NARM) && (armarios[a] != 0); a++)
12       ;                /* percorre o vetor até achar um armário livre */
13    armarios[a] = i;
14    up(&mutex);         /* libera a região crítica */
15    estuda();
16    down(&mutex);       /* acessa a região crítica */
17    armarios[a] = 0;
18    up(&mutex);         /* libera a região crítica */
19    up(&livres);
20 }
```

9. (a) O estado compartilhado é representado pelas variáveis `vagas` e `ocupadas`. Assim, existem duas regiões críticas no código, a primeira nas linhas 11–16 e a segunda nas linhas 32–33.

   (b) Uma sequência indesejável de execução seria esta, onde `MAX == ANDARES*MAXVAGAS`:

   I. Num momento em que existe apenas uma vaga livre (i.e., `ocupadas == MAX-1`), dois usuários X e Y chegam simultaneamente em duas cancelas;

   II. Na execução do código, o usuário X chega até a linha 14, encontra a vaga livre, mas perde o processador entre as linhas 14 e 15;

   III. O usuário Y chega até a linha 14, encontra a mesma vaga livre e executa as linhas 15 e 16, ocupando a vaga e fazendo `ocupadas == MAX`;

   IV. O usuário X retoma o processador e executa as linhas 15 e 16, sobrescrevendo a alocação da vaga para o usuário Y e fazendo `ocupadas == MAX+1`.

   O resultado é que os dois usuários obtêm tíquetes para a mesma vaga, que á a única livre no estacionamento. Isso não vai terminar bem.

   (c) A solução passa por usar um semáforo binário para guardar o acesso às regiões críticas. A solução completa está mostrada abaixo. É importante não esquecer de liberar o acesso à região crítica não só quando uma vaga for encontrada (linha 17), mas também quando o estacionamento estiver cheio (linha 27).

```
1  typedef struct { int andar, vaga; } ticket_t;
2  carro_t vagas[ANDARES][MAXVAGAS];        /* matriz de ocupação de vagas */
3  int ocupadas = 0;                        /* no. de vagas ocupadas */
4  semaphore mutex = 1;                     /* semáforo para exclusão mútua */
5
6  ticket_t libera_ticket(carro_t carro)
7  {
8     int a, v;
9     ticket_t ticket;
10    down(&mutex);                          /* bloqueia RC */
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                up(&mutex);               /* libera RC */
18                ticket.andar = a;
19                ticket.vaga = v;
20                imprime_ticket(ticket);   /* imprime o ticket para o motorista */
21                libera_entrada();         /* abre a cancela de entrada */
22                return ticket;
23             }
24          }
25       }
26    }
27    up(&mutex);                 /* libera RC */
28    libera_escape();            /* estacionamento cheio, abre cancela de escape */
29    return ticket;
30 }
31
32 void retorna_ticket(ticket_t ticket)
33 {
34    down(&mutex);                           /* bloqueia RC */
35    ocupadas--;                             /* contabiliza a liberação da vaga */
36    vagas[ticket.andar][ticket.vaga] = LIVRE;   /* ajusta o mapa de vagas */
37    up(&mutex);                             /* libera RC */
38    libera_saida();                         /* abre a cancela de saída */
39 }
```

10. (a) Como o problema é de sincronização, são usados dois semáforos contadores. O semáforo a conta o número de atendentes livres, e o semáforo `cli` conta o número de clientes esperando para serem atendidos.

```
semaphore a = NA;
semaphore cli = 0;

void cliente() {
   up(&cli);
   down(&a);
   solicita_copias();
   paga_servico();
   up(&a);
}

void atendente() {
   while (TRUE) {
      down(&cli);
      recebe_pedido();
      faz_copias();
      recebe_pagamento();
   }
}
```

(b) Nesta solução é acrescentado o semáforo `c`, que conta o número de copiadoras livres.

```
semaphore a = NA;
semaphore c = NC;
semaphore cli = 0;

void cliente() {
   up(&cli);
   down(&a);
   solicita_copias();
   paga_servico();
   up(&a);
}

void atendente() {
   while (TRUE) {
      down(&cli);
      recebe_pedido();
      down(&c);
      faz_copias();
      up(&c);
      recebe_pagamento();
   }
}
```

11. (a) Sim. O primeiro leitor irá decrementar `tipo` de 1 para 0, e os leitores seguintes apenas irão incrementar `nl`, sem correr o risco de ficarem bloqueados no semáforo.

(b) Não. O primeiro irá escritor decrementar `tipo` de 1 para 0, e os escritores seguintes ficarão bloqueados no semáforo até que o primeiro escritor o libere (`up(&tipo)`).

(c) Não. Um leitor pode ser bloqueado por um período indeterminado de tempo por um escritor usando o semáforo `tipo`, situação em que outros escritores também ficariam bloqueados. Como o escritor em algum momento irá liberar `tipo`, o leitor poderá acessar a região crítica do texto (`acessa_texto()`).

Um leitor também pode ser bloqueado por outro leitor usando o semáforo `exclusivo`. Nesse caso, a região crítica do texto pode estar livre, e o leitor que detém `exclusivo` está apenas atualizando as variáveis, e logo após irá liberar o acesso ao texto. Se a região crítica do texto estiver ocupada, o leitor que detém `exclusivo` está esperando por um escritor que detém `tipo`, e, pelo explicado no parágrafo anterior, terminará por conseguir acessar o texto, trazendo consigo todos os leitores que porventura estejam bloqueados em `exclusivo`.

(d) Sim. Um escritor pode ser bloqueado por um tempo indeterminado pelo semáforo `tipo`. Se esse semáforo tiver sido bloqueado por um leitor, outros leitores poderão acessar a região crítica do texto sem liberá-lo; caso ocorra uma chegada ininterrupta de leitores, o escritor sofrerá postergação indefinida.

(e) Não. O semáforo `tipo` implementa exclusão mútua entre leitores e escritores: caso um leitor e um escritor tentem acessar o texto "ao mesmo tempo" quando `tipo` estiver em 1, quem executa `down(&tipo)` primeiro decrementa o semáforo de 1 para 0 e prossegue sua execução, enquanto o segundo decrementa o semáforo de 0 para -1, ficando bloqueado até que `up(&tipo)` seja executado.

12. A solução envolve um semáforo adicional, chamado de `roleta`. Enquanto não houver escritores tentando acessar a região crítica, todos os leitores passam pela roleta (linhas 7 e 8). A chegada de um escritor trava a roleta (linha 25), fazendo com que os leitores seguintes fiquem bloqueados na linha 7. A roleta será liberada na linha 28, depois que o escritor tiver acessado a região crítica, o só irá ocorrer quando o último leitor liberar `tipo` (linhas 19 e 26). Caso haja outros escritores bloqueados na roleta (linha 25), o escalonador é que irá determinar se um leitor ou um escritor será desbloqueado.

```
1  int nl = 0;
2  semaphore tipo = 1;
3  semaphore exclusivo = 1;
4  semaphore roleta = 1;
```

```
5  void leitor(void)
6  {  ...
7     down(&roleta);
8     up(&roleta);
9     down(&exclusivo);
10    if (nl > 0) ++nl;
11    else {
12       down(&tipo);
13       nl = 1;
14    }
15    up(&exclusivo);
16    acessa_texto();
17    down(&exclusivo);
18    --nl;
19    if (nl == 0) up(&tipo);
20    up(&exclusivo);
21    ...
22 }
```

```
23 void escritor(void)
24 {  ...
25    down(&roleta);
26    down(&tipo);
27    acessa_texto();
28    up(&roleta);
29    up(&tipo);
30    ...
31 }
```

13. A solução mais intuitiva para o problema, em que cada filósofo, ao resolver comer, simplesmente pega o garfo direito e depois o esquerdo (ou vice-versa), está sujeita a *deadlock*: se todos os filósofos pegarem o garfo direito, todos ficarão bloqueados esperando pelo garfo esquerdo. Seguindo a dica, se no máximo quatro filósofos puderem tentar pegar os garfos, ao menos um deles terá sucesso em pegar ambos, e não ocorrerá *deadlock*. Essa solução é mostrada a seguir. Os filósofos e garfos são numerados de 0 a 4; os garfos do filósofo i são i e $(i + 1) \mod N$, como representado por `LEFT` e `RIGHT`. O semáforo `hungry` controla quantos filósofos ainda podem tentar pegar garfos, e o semáforo `forks` controla a alocação de cada garfo. Caso todos os filósofos tentem comer ao mesmo tempo, um deles ficará bloqueado por `hungry` (linha 11), o que permitirá que um dos demais consiga pegar seus dois garfos.

```
1  #define N     5
2  #define LEFT  i
3  #define RIGHT (i+1)%N
4
5  semaphore forks[N] = 1;
6  semaphore hungry = N-1;
7
8  void philosopher(int i) {
9     while (TRUE) {
10       think();
11       down(&hungry);
12       down(&forks[RIGHT]);
13       down(&forks[LEFT]);
14       eat();
15       up(&forks[RIGHT]);
16       uo(&forks[LEFT]);
17       up(&hungry);
18    }
19 }
```

14. (a) À primeira vista, parece que `soma` será sempre 100. Uma segunda inspeção revela que, como não há exclusão mútua entre os processos na manipulação da variável, `soma` ficará no intervalo $50 \leq$ `soma` ≤ 100, já que de 0 a 50 incrementos podem ser perdidos. No entanto, existe uma sequência de execução que produz um resultado ainda menos óbvio:

    I. O processo A carrega o valor de `soma`, faz o incremento e perde o processador (o registrador `ACC` conterá 1, mas esse valor não foi armazenado em `soma`);

    II. O processo B carrega o valor de `soma` (ainda zero) e realiza 49 incrementos completos, perdendo o processador após armazenar 49 em `soma`;

    III. O processo A retoma o processador, armazena o valor de `ACC` (1) em `soma`, e perde novamente o processador;

    IV. O processo B retoma o processador, carrega o valor de `soma` (1) em `ACC` e perde novamente o processador;

    V. O processo A retoma o processador e efetua seus 49 incrementos restantes, armazenando o resultado (50) em `soma`;

    VI. O processo B retoma o processador, incrementa `ACC` (que vale 1) e armazena o seu valor (2) em `soma`.

    Portanto, o intervalo correto de possíveis valores é $2 \leq$ `soma` ≤ 100.

    (b) No caso de N processos, o intervalo de possíveis valores para `soma` será $2 \leq$ `soma` $\leq 50N$, pois todos os outros processos poderão executar por completo no passo V antes que o processo B desfaça tudo ao terminar por último (passo VI).