MC514—Sistemas Operacionais: Teoria e Prática

Profa. Islene Calciolari Garcia

28 de abril de 2009

Questão	Nota
1	
2	
3	
4	
Total	

Nome:	RA:

Instruções: Você pode fazer a prova a lápis, desde que o resultado final seja legível. Não é permitida consulta a qualquer material manuscrito ou impresso. Em caso de fraude, todos os envolvidos receberão nota zero. **Boa prova!**

Funções:

```
int pthread_mutex_lock(pthread_mutex_t *mutex);
int pthread_mutex_unlock(pthread_mutex_t *mutex);

int pthread_cond_signal(pthread_cond_t *cond);
int pthread_cond_wait(pthread_cond_t *cond, pthread_mutex_t *mutex);

int futex_wait(void *addr, int val1);  /* Bloqueia se *addr == val1 */
int futex_wake(void *addr, int n);     /* Acorda até n threads */
```

1. (2.0) Um programador estava escrevendo um tratador para um sinal assíncrono (SIGUSR1) que deveria fazer acesso a uma estrutura de dados também utilizada por uma função `f()` do seu programa. Preocupado com possíveis erros de consistência, este programador pensou em utilizar *mutex locks*, da maneira como está esquematizado abaixo. (a) Por que esta não seria uma boa solução para o problema? (b) Locks recursivos seriam mais indicados? Justifique todas as respostas.

```
1:  /* Tratador invocado quando um sinal tipo SIGUSR1 é recebido */
2:  void trata_SIGUSR1(int signum) {
3:    pthread_mutex_lock(&mutex);
4:   /* Acesso a dados */
5:   pthread_mutex_unlock(&mutex);
6:  }
7:  /* Função do programa */
8:  void f() {
9:    pthread_mutex_lock(&mutex);
10:   /* Acesso a dados */
11:   pthread_mutex_unlock(&mutex);
12: }
```

2. No código abaixo, uma thread gerente tenta controlar a entrada na região crítica de N threads da aplicação.

```
01: #define N 100
02: volatile int interesse[N]; /* interesse[i]: indica se a thread i está ou não
03:                               interessada em fazer acesso à região crítica */
04: volatile int vez[N];       /* vez[i]: indica se é ou não a vez da thread i */
05:
06: /* Retorna o id de uma thread interessada ou -1 se não encontrar nenhuma. */
07: int interessada() {
08:   int k;
09:   for (k = 0; k < N; k++)
10:     if (interesse[k]) return k;
11:   return -1;
12: }
13:
14: void* gerente(void* v) {
15:   int p;
16:   while (1) {
17:     p = interessada();
18:     if (p != -1) {  /* p poderá fazer acesso à região crítica */
19:       vez[p] = 1;                   /* Indica vez da thread p */
20:       futex_wake(&vez[p], 1);       /* Acorda p caso necessário */
21:       futex_wait(&interesse[p], 1); /* Aguarda p terminar */
22:       vez[p] = 0;                   /* Retira vez da thread p */
23     }
24:    else
25:      futex_wait(&pausa_gerente, 1); /* Gerente pode descansar um pouco */
26:   }
27: }
28:
29: void* f_thread(void *v) {
30:  int i = *(int *) v;
31:  while (1) {
32:    regiao_nao_critica();          /* Pode demorar muito aqui... */
33:    interesse[i] = 1;              /* Manifesta interesse */
34:    futex_wake(&pausa_gerente, 1); /* Gerente não pode descansar agora */
35:    futex_wait(&vez[i], 0);        /* Aguarda sua vez */
36:    regiao_critica();              /* Pode demorar aqui também... */
37:    interesse[i] = 0;              /* Encerra o acesso à região crítica */
38:    futex_wake(&interesse[i], 1);  /* Acorda o gerente */
39:  }
40: }
```

(a) (1.5) Descreva um cenário de *starvation* e reescreva a função `interessada()` de modo a eliminar o problema.

(b) (1.5) Descreva um cenário de exclusão mútua e indique uma alteração simples no código que eliminaria o problema.

(c) (1.5) Descreva um cenário de *deadlock*. Você consegue indicar alguma alteração simples no código para resolver este problema? Em caso afirmativo, apresente a modificação. Caso contrário, comente quais são as dificuldades computacionais envolvidas na solução do problema.

3. (1.5) O algoritmo abaixo tenta implementar `mutex_locks` a partir de futexes e operações atômicas de incremento. Quando igual a 0, o campo `val` indica lock livre, qualquer outro valor indica lock travado. A operação `atomic_inc` retorna o valor da variável imediatamente antes do incremento atômico. Este algoritmo fez parte da biblioteca NPTL (Native POSIX Thread Library) durante vários meses e foi responsável por perdas de desempenho e picos de uso de CPU. Descreva o cenário que levava a este problema.

```
1:  typedef struct {
2:    volatile int val = 0;
3:  } mutex_t;
4:
5:  void mutex_lock(mutex_t *m) {
6:    int c;
7:    while ((c = atomic_inc(&m->val)) != 0)
8:      futex_wait(&m->val, c+1);
9:  }
10:
11: void mutex_unlock(mutex_t *m) {
12:   m->val = 0;
13:   futex_wake(&m->val, 1);
14: }
```

4. (2.0) Durante uma aula no semestre passado, um aluno de pós-graduação propôs a seguinte solução para o problema dos leitores e escritores. Neste problema, várias threads compartilham dados, sendo que as operações de escrita necessitam de acesso exclusivo, mas operações de leitura podem ser concorrentes.

```
volatile int nl; /* Número de leitores */
mutex_t mutex_dados, mutex_nl;
cond_t cond;
```

```
     /***** Leitura *****/
L1:  mutex_lock(&mutex_dados);
L2:  mutex_lock(&mutex_nl);
L3:  nl++;
L4:  mutex_unlock(&mutex_nl);
L5:  mutex_unlock(&mutex_dados);
L6:  le_dados();
L7:  mutex_lock(&mutex_nl);
L8:  nl--;
L9:  if (nl == 0)
L10:   cond_signal(&cond);
L11: mutex_unlock(&mutex_nl);
```

```
     /***** Escrita *****/
E1: mutex_lock(&mutex_dados);
E2: mutex_lock(&mutex_nl);
E3: while (nl > 0)
E4:   cond_wait(&cond, &mutex_nl);
E5: escreve_dados();
E6: mutex_unlock(&mutex_nl);
E7: mutex_unlock(&mutex_dados);
```

Responda as questões abaixo, **justificando-as**.

(a) Há garantia de que apenas um escritor executa a operação de escrita em um dado instante?

(b) Vários leitores podem executar simultaneamente a operação de leitura em um dado instante?

(c) Há possibilidade de *starvation* de leitores?

(d) Há possibilidade de *starvation* de escritores?