MC514—Sistemas Operacionais: Teoria e Prática

Profa. Islene Calciolari Garcia

8 de abril de 2010

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, impresso ou eletrônico. Em caso de fraude, todos os envolvidos receberão nota zero. **Boa prova!**

Funções:

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

/* Operações atômicas */
int bool_cmpxchg (int *i, int old, int new);
    /* se *i == old, *i recebe new; caso contrário não altera *i;
       retorna 1 se executou a troca ou 0 caso contrário. */

int atomic_inc (int *i); /* Incrementa *i atomicamente;
                            retorna o valor de *i antes da operação. */

int atomic_dec (int *i); /* Decrementa *i atomicamente;
                            retorna o valor de *i antes da operação. */
```

1. (2.0) **Incremento módulo N atômico.** Observe o código abaixo:

```
/* Executa *i = (*i + 1) % N; retorna o valor de *i antes da operação. */
int atomic_inc_mod_N(int *i, int N) {
  if (bool_cmpxchg(i, N-1, 0))  /* Cobre o caso em que *i == N-1 */
    return N-1;
  else
    return atomic_inc(i);  /* Cobre o caso em que *i < N-1 */
}
```

Qual é o valor máximo que *i pode atingir? Justifique sua resposta. Se o valor for maior do que N-1, apresente uma função que efetue corretamente a operação atômica de incremento módulo N. Você só poderá utilizar `atomic_inc()` e/ou `bool_cmpxchg()` para implementar esta operação.

2. (2.0) **O fim das threads espertinhas?** Muitos desenvolvedores ficam revoltados com o fato de algumas implementações de `mutex_locks` permitirem que threads furem a fila, enquanto existem outras aguardando no futex (suponha que os futexes respeitam a política First In First Out perfeitamente). Tentando mudar esta situação, eles utilizaram um contador de threads esperando `n_waiters` que é incrementado atomicamente. Eles aproveitaram este contador para evitar algumas chamadas à operação `futex_wake()`. Vale ressaltar que o valor 0 no campo `val` indica lock livre e o valor 1 indica lock travado.

```
01:  typedef struct {
02:    volatile int val;
03:    volatile int n_waiters;
04:  } mutex_t;
05:
06:  void mutex_init(mutex_t *m) {
07:     m->val = m->n_waiters = 0;
08:  }
09:
10:  void mutex_lock(mutex_t *m) {
11:
12:    if (atomic_inc(n_waiters) > 0)
13:      futex_wait(&m->val, 1); /* Como já havia pelo menos uma thread
14                                  esperando, entra na fila do futex. */
15:    while (!bool_cmpxchg(&m->val, 0, 1))
16:      futex_wait(&m->val, 1);
17:  }
18:
19: void mutex_unlock(mutex_t *m) {
20:   m->val = 0;
21:   if (atomic_dec(n_waiters) > 1) /* Há mais threads na espera */
22:     futex_wake(&m->val, 1);
23: }
```

Este algoritmo garante exclusão mútua? Esta abordagem eliminou o problema das threads espertinhas? Justifique as suas respostas.

3. **O Gerente dorminhoco e as threads bajuladoras.** No código abaixo, uma thread gerente tenta controlar a entrada na região crítica de N threads da aplicação. A thread gerente conta com a ajuda das threads bajuladoras, que evitam acordá-la à toa.

```
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: volatile int pausa_gerente = 1;
06
07: /* Retorna o id (diferente de i) de uma thread interessada
08:    ou -1 se não existir nenhuma. */
09: int interessada(int i) {
10:   int k;
11:   for (k = 0; k < N; k++)
12:     if (k != i && interesse[k]) return k;
13:   return -1;
14: }
15:
16: void* gerente(void* v) {
17:   int p;
18:   while (1) {
19:     p = interessada(-1);
20:     if (p != -1) {  /* p poderá fazer acesso à região crítica */
19:       vez[p] = 1;                   /* Indica vez da thread p */
21:       futex_wake(&vez[p], 1);       /* Acorda p caso necessário */
22:     }
23:     futex_wait(&pausa_gerente, 1); /* Gerente pode descansar um pouco */
24:   }
25: }
26:
27: void* f_thread(void *v) {
28:  int p, i = *(int *) v;
29:  while (1) {
30:    regiao_nao_critica();          /* Pode demorar muito aqui... */
31:    interesse[i] = 1;              /* Manifesta interesse */
32:    if (interessada(i) == -1)   {     /* Se for a única interessada, */
33:       pausa_gerente = 0;             /*     precisará incomodar o gerente. */
34:       futex_wake(&pausa_gerente, 1);
35:    }
36:    futex_wait(&vez[i], 0);        /* Aguarda sua vez */
37:    regiao_critica();              /* Pode demorar aqui também... */
38:    vez[i] = 0;                    /* Marca saída da região crítica */
39:    interesse[i] = 0;
40:    p = interessada(i); /* faz o trabalho do gerente... */
41:     if (p != -1) {   /* p poderá fazer acesso à região crítica */
42:       vez[p] = 1;                   /* Indica vez da thread p */
43:       futex_wake(&vez[p], 1);       /* Acorda p caso necessário */
44      }
45:  }
46: }
```

(a) (1.0) Aponte uma falha na abordagem utilizada na função `interessada(i)`. Reescreva esta função de modo a eliminar o problema.

(b) (1.0) Logo no início da execução do sistema, existe a possibilidade de *deadlock*. Indique este cenário e as modificações no código necessárias para eliminar o problema.

(c) (1.0) As threads bajuladoras falharam e o gerente não está conseguindo dormir... Qual linha de código está faltando? Em que ponto do programa ela deveria ser inserida? (Indique a resposta baseando-se no código sem as modificações dos itens anteriores).

(d) (1.0) Considere que todos os problemas acima foram resolvidos. Qual abordagem é a melhor? A das threads bajuladoras ou a que o gerente é a única thread que controla a entrada na região crítica? Justifique a sua resposta.

4. (2.0) **O Jantar com vinho.** Cinco filósofos levam uma vida monótona ao redor de uma mesa: eles pensam, ficam com fome e comem. Considere que em um dia de festa, eles foram autorizados a tomar vinho. Como eles estão acostumados a compartilhar recursos, apenas dois copos foram colocados no centro da mesa. Animados com a novidade, eles primeiro disputam o copo entre todos e depois os garfos com os vizinhos. Observe o pseudo-código abaixo:

```
  sem_t garfo[5] = {1,1,1,1,1};
  sem_t copo = 2;

Filósofo i:
   while (1) {
     pensa();
     sem_wait(&copo);
     sem_wait(&garfo[i]);
     sem_wait(&garfo[(i+1)%N]);
     come();
     sem_post(&copo);
     sem_post(&garfo[i]);
     sem_post(&garfo[(i+1)%N]);
   }
```

(a) Este algoritmo está sujeito a *deadlock*? Justifique a sua resposta.

(b) Em uma mesa com cinco filósofos, apenas dois podem comer ao mesmo tempo. Isto quer dizer que a compra de mais um copo não aumentaria o paralelismo do sistema? Justifique a sua resposta.

(c) Valeria a pena colocar 5 copos na mesa?