MC504/MC514—Sistemas Operacionais

Profa. Islene Calciolari Garcia

15 de outubro de 2014

Questão	Nota
1	
2	
3	
4	
5	
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!**

1. (2.0) **O Jantar com vinho e suspeita de roubo.** Cinco filósofos levam uma vida monótona ao redor de uma mesa: eles pensam, ficam com fome e comem. Para comer, eles precisam de dois garfos. No algoritmo tradicional, cada garfo é compartilhado por dois filósofos. Em um dia especial, eles foram autorizados a beber vinho. Apenas **duas** taças foram colocadas no centro da mesa controladas pelo semáforo `sem_taca`. No entanto, neste mesmo dia um dos garfos desapareceu. Para não interromper a festa, eles resolveram deixar os quatro garfos restantes no centro da mesa, que seriam controlados por um único semáforo `sem_garfo` e compartilhados por todos. (a) Analise o código abaixo e descreva um cenário de *deadlock*. (b) Proponha uma alteração no código que garanta ausência de *deadlock*.

```
01: sem_t sem_garfo = 4;
02: sem_t sem_taca = 2;
03:
04: F0, F1, F2, F3, F4:
05:   while (1) {
06:     pensa();
07:     sem_wait(&sem_garfo);
08:     sem_wait(&sem_garfo);
09:     sem_wait(&sem_taca);
10:     come();
11:     sem_post(&sem_garfo);
12:     sem_post(&sem_garfo);
13:     sem_post(&sem_taca);
14:   }
```

2. (2.0) Um programa deve ler o nome de um arquivo de entrada e depois fazer várias operações muito demoradas sobre este arquivo. Preocupado com o tempo total de execução, um desenvolvedor impaciente resolveu emitir mensagens para apressar o usuário a digitar o nome do arquivo. (a) Você acha que o usuário vê as mensagens apenas quando está demorando para digitar? (b) Como você faria para não aborrecer o usuário se ele já tiver terminado de digitar? (c) Sua abordagem garante que os dados escritos pelo usuário não aparecerão misturados a uma mensagem? Justifique suas respostas.

```
01: void trata_SIGALRM(int signum) {
02:   printf("Digite rápido, por favor!\n");
03:   alarm(5); /* Programa um sinal para daqui a 5 segundos. */
05: }
06:
07: int main() {
08:   char nome_arq[100];
09:
10:   signal(SIGALRM, trata_SIGALRM); /* Instala um tratador para SIGALRM */
11:   alarm(5);  /* Programa um sinal para daqui a 5 segundos. */
12:   printf("Digite o nome do arquivo de entrada: ");
13:   scanf("%s", nome_arq);
14:   trabalha_muito_com_o_arquivo(nome_arq);
15:   return 0;
16: }
```

3. (2.0) Barreiras são primitivas para sincronização de várias threads em algum ponto do programa. Em uma abordagem didática, o livro The Little Book on Semaphores, de Allen B. Downey, apresenta inicialmente várias implementações erradas antes de apresentar uma versão correta e mais abrangente para a implementação de barreiras.

   A versão abaixo é uma variante das soluções propostas por Downey baseada em operações atômicas de incremento e decremento. Suponha que `b->tamanho+1` threads executam a função `barrier_wait` de maneira concorrente. Há garantia que `b->tamanho` threads passarão e 1 thread ficará presa na barreira? Justifique sua resposta.

```
01:  typedef struct {
02:    int c, tamanho;
03:    sem_t sem_barreira = 0;
04:  } barrier_t;
05:
06:  /* tamanho: número de threads que devem se sincronizar via barrier_wait() */
07:  void barrier_init(barrier_t b, int tamanho) {
08:    b->c = 0;
09:    b->tamanho = tamanho;
10:    sem_init(&b->sem_barreira, 0);
11:  }
12:
13:  void barrier_wait (barrier_t *b) {
14:     atomic_inc(&b->c);
15:     if (b->c == b->tamanho)
16:       sem_post(&b->sem_barreira);
17:     sem_wait(&b->sem_barreira);
18:     atomic_dec(&b->c);
19:     if (b->c > 0)
20:       sem_post(&b->sem_barreira);
21:    }
22:
23:  }
```

4. (2.0) 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.

```
01: /* Operações com futexes */
02: int futex_wait(void *addr, int val1);  /* Bloqueia se *addr == val1 */
03: int futex_wake(void *addr, int n);     /* Acorda até n threads */
04:
05: typedef struct {
06:   volatile int val = 0;
07: } mutex_t;
08:
09: void mutex_lock(mutex_t *m) {
10:   int c;
11:   while ((c = atomic_inc(&m->val)) != 0)
12:     futex_wait(&m->val, c+1);
13: }
14:
15: void mutex_unlock(mutex_t *m) {
16:   m->val = 0;
17:   futex_wake(&m->val, 1);
18: }
```

5. (2.0) Analise o código abaixo:

```
#define N_THR 5

void* f_thread(void *v) {
  int thr_id;
  thr_id = *(int *) v;
  printf("Thread %d. ", thr_id);
  return NULL;
}

int main() {
  pthread_t thr[N_THR];
  int i;

  for (i = 0; i < N_THR; i++)
    pthread_create(&thr[i], NULL, f_thread, (void*) &i);

  for (i = 0; i < N_THR; i++)
    pthread_join(thr[i], NULL);

  return 0;
}
```

A seguinte saída seria possível? Em caso afirmativo, apresente resumidamente uma seqüência de execução das threads que levaria a esta saída. Em caso negativo, explique o motivo.

```
Thread 0. Thread 0. Thread 0. Thread 0. Thread 4.
```