MC514—Sistemas Operacionais: Teoria e Prática

Profa. Islene Calciolari Garcia

8 de maio de 2008

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 :-) e utilizar o verso das folhas para rascunho ou para completar a resolução das questões. Não é permitida consulta a qualquer material manuscrito ou impresso. Em caso de fraude, todos os envolvidos receberão nota zero.

1. (2.0 pontos) Considere que houve uma falha na implementação do algoritmo do desempate proposto por Peterson para exclusão mútua que resultou na troca da ordem das linhas T0.2 e T0.3 na thread 0. Esta implementação ainda garante exclusão mútua? Em caso afirmativo, justifique a sua resposta. Em caso negativo, indique uma ordem de execução das linhas (na forma T0.1, T1.1, ...) que geraria um acesso indevido à região crítica.

```
int ultimo = 0, interesse[2] = {false, false};
Thread_0:                              Thread_1:
(T0.1) while (true)                     (T1.1) while (true)
(T0.2)   ultimo = 0;                    (T1.2)   interesse[1] = true;
(T0.3)   interesse[0] = true;           (T1.3)   ultimo = 1;
(T0.4)   while (ultimo == 0             (T1.4)   while (ultimo == 1
           && interesse[1]) ;                         && interesse[0]) ;
(T0.5)   regiao_critica();              (T1.5)   regiao_critica();
(T0.6)   interesse[0] = false;          (T1.6)   interesse[1] = false;
```

2. (2 pontos) Observe o código abaixo, que tem uma abordagem semelhante ao algoritmo de alternância entre threads na região crítica. (a) É possível que uma thread executando na região não crítica impeça outra thread de entrar na região crítica? (b) Este código garante exclusão mútua entre as threads? Justifique suas respostas.

```
int vez = -1; /* Nenhuma thread está interessada */
Thread_i:
  while (true)
    while (vez != -1) ;
    vez = i;
    regiao_critica();
    vez = -1;
    regiao_nao_critica();
```

3. (2.0 pontos) No algoritmo da padaria proposto por Lamport, duas ou mais threads podem escolher a mesma senha e o desempate é feito por meio do identificador da thread. Um programador achou isto injusto e preferiu tentar implementar um esquema no qual cada thread escolheria garantidamente uma senha diferente. A idéia é que, durante a operação de máximo, se uma thread estiver escolhendo um número, a outra deve aguardar. Identifique um problema no código abaixo.

```
/* Variáveis globais */
escolhendo[N] = { false, false, ..., false }
num[N] = { 0, 0, ..., 0 }

Thread i:
  /* max_i e j são variáveis locais */
  escolhendo[i] = true;
  max_i = 0;
  for (j = 0; j < N; j++) {
    if (j != i)
      while (escolhendo[j]) ; /* Espera j terminar de escolher */
    if (num[j] > max_i)
        max_i = num[j];
  }
  num[i] = max_i + 1;
  escolhendo[i] = false;

  /* Espera threads com senhas menores saírem da região crítica */
  for (j = 0; j < N; j++)
    while (num[j] != 0 && num[j] < num[i]) ;

  regiao_critica();
  num[i] = 0;
```

4. (2.0 pontos) Considerando as declarações e inicializações fornecidas, implemente a função `proximo()` de maneira que a função `faz_alguma_coisa()` seja chamada exatamente uma vez para cada valor de `i` entre 0 e `NMAX` − 1 mesmo que haja múltiplas threads executando `f_thr()` simultaneamente.

```
volatile int n = 0;  /* Variável compartilhada para o controle do número
                        de execuções da função faz_alguma_coisa() */

#define NMAX 100     /* Número máximo de execuções desta função */

/* Declaração e inicialização de um lock simples para controlar o
   acesso à variável n. */
pthread_mutex_t lock_n = PTHREAD_MUTEX_INITIALIZER;

/* Função auxiliar que deve ser invocada exatamente uma vez para
   cada valor de i entre 0 e NMAX-1. */
void faz_alguma_coisa(int i) {
   sleep(random() % 5); /* vamos fingir que deu trabalho! :-) */
   printf("Fiz alguma coisa para i = %d\n", i);
}

/* Função a ser executada por um grupo de threads. */
void *f_thr(void *v) {
  int i;
  while ((i = proximo()) != NMAX) {
     faz_alguma_coisa(i);
  }
  return NULL;
}

/* Por exemplo, para NMAX = 5, uma saída válida para este programa seria:
   Fiz alguma coisa para i = 0
   Fiz alguma coisa para i = 2
   Fiz alguma coisa para i = 1
   Fiz alguma coisa para i = 4
   Fiz alguma coisa para i = 3   */

/* Implemente a função próximo(). Para adquirir ou liberar um lock:
      int pthread_mutex_lock(pthread_mutex_t *mutex);
      int pthread_mutex_unlock(pthread_mutex_t *mutex); */
int proximo() {
```

5. (2.0 pontos) Analise o código abaixo para o problema dos leitores e escritores.

```
pthread_mutex_t lock;
pthread_cond_t cond;
volatile int nl = 0; /* Número de leitores */
volatile int ne = 0; /* Número de escritores */

void *leitor(void* v) {
  pthread_mutex_lock(&lock);
  while (ne > 0)
    pthread_cond_wait(&cond, &lock);
  nl++;
  pthread_mutex_unlock(&lock);

  leitura(v);

  pthread_mutex_lock(&lock);
  nl--;
  pthread_cond_broadcast(&cond);
  pthread_mutex_unlock(&lock);

  return NULL;
}
```

```
void *escritor(void *v) {
  pthread_mutex_lock(&lock);
  while (nl > 0 || ne > 0)
    pthread_cond_wait(&cond, &lock);
  ne++;
  pthread_mutex_unlock(&lock);

  escrita(v);

  pthread_mutex_lock(&lock);
  ne--;
  pthread_cond_broadcast(&cond);
  pthread_mutex_unlock(&lock);

  return NULL;
}
```

Responda as questões abaixo, justificando-as (respostas sem justificativa adequada não serão consideradas).

(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?