MC504/MC514—Sistemas Operacionais

Profa. Islene Calciolari Garcia

Prova 1

29 de março de 2016

Questão	Nota
1	
2	
3	
4	
5	
Total	

Nome:

RA:

Instruções: Você pode fazer a prova a lápis e utilizar o verso das folhas para completar suas respostas. Não é permitida consulta a qualquer material manuscrito, impresso ou em formato eletrônico. Em caso de fraude, todos os envolvidos receberão nota zero.

1. (2.0) Analise o código abaixo. Considere um sistema GNU/Linux similar ao que você utiliza nos laboratório do IC e descreva o erro que irá ocorrer. Qual é a saída esperada do programa?

```
#include <stdio.h>
void f() {
  int v_f[5];
  int i;
  printf("Função f!\n");
  for (i = 0; i < 100; i++)
     v_f[i] = i;
  f();
}

int main() {
  int v_main[100];
  for (i = 0; i < 100; i++)
     v_main[i] = i;
  f();
  return 0;
}
```

2. (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];
  volatile int i, j;

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

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

  return 0;
}
```

As seguintes saídas seriam possíveis? Em caso afirmativo, apresente resumidamente uma seqüência de execução das *threads* que levaria à saída. Em caso negativo, explique o motivo.

(a) Thread 0. Thread 0. Thread 0. Thread 0. Thread 4.

(b) Thread 0. Thread 5. Thread 5. Thread 5. Thread 5.

3. (2.0) **O Jantar com vinho e um canhoto.** 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 **três** taças foram colocadas no centro da mesa. Animados com a novidade, os filósofos F0, F1, F2 e F3 resolveram primeiro disputar a taça e depois os garfos com os vizinhos. O filósofo F4, que também é o único canhoto do grupo, resolveu pegar primeiro os garfos na ordem inversa dos seus vizinhos e depois disputar a taça. Este algoritmo baseado em semáforos está sujeito a *deadlock*? Justifique a sua resposta.

```
01: sem_t garfo[5] = {1,1,1,1,1};
02: sem_t taca = 3;
03:
04: F0, F1, F2, F3:
05:   while (1) {
06:     pensa();
07:     sem_wait(&taca);
08:     sem_wait(&garfo[i]);
09:     sem_wait(&garfo[(i+1)%N]);
10:     come();
11:     sem_post(&taca);
12:     sem_post(&garfo[i]);
13:     sem_post(&garfo[(i+1)%N]);
14:   }
```

```
F4:
   while (1) {
     pensa();
     sem_wait(&garfo[(i+1)%N]);
     sem_wait(&garfo[i]);
     sem_wait(&taca);
     come();
     sem_post(&garfo[(i+1)%N]);
     sem_post(&garfo[i]);
     sem_post(&taca);
   }
```

4. (2.0) **A padaria com atendimento preferencial** Um desenvolvedor pensou em uma maneira simples de alterar o algoritmo da padaria proposto por Lamport para atender *threads* com prioridade. As *threads* prioritárias poderiam escolher números em uma faixa mais baixa do que as *threads* não prioritárias, que deveriam respeitar um número mínimo de 500. Analise o pseudocódigo abaixo e responda as questões a seguir, justificando-as:

   (a) Há garantia de que as senhas das *threads* prioritárias e das não prioritárias ficarão separadas ao longo do tempo?

   (b) As *threads* não prioritárias poderão sofrer *starvation*?

   (c) Essa estratégia garante exclusão mútua?

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

Thread i:
  escolhendo[i] = true;
  if (Thread i é prioritária)
     num[i] = max (num[j] tal que thread j é prioritária) + 1;
  else {
     num[i] = max (num[j] tal que thread j não é prioritária) + 1;
     if (num[i] < 500) num[i] = 500; /* Senha mínima não prioritária */
  }
  escolhendo[i] = false;

 /* Espera threads com senhas menores saírem da região crítica.
    Em caso de empate, o desempate é feito pelo identificador da thread. */
  for (j = 0; j < N; j++) {
    while (escolhendo[j]) ;
    while (num[j] != 0 &&
          (num[j] < num[i] || num[i] == num[j] && j < i));

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

5. (2.0) **A thread gerente e o atendimento preferencial** Vendo que a alteração da padaria apresentava algumas falhas, o mesmo desenvolvedor resolveu implementar filas preferenciais alterando o código da *thread* gerente visto em aula. Ele criou um vetor separado para *threads* com prioridade e deu a cada entrada do vetor `interesse` o seguinte significado:

```
interesse[i] = 0 => thread i não quer fazer acesso a região crítica
interesse[i] = 1 => thread i quer fazer acesso a região crítica
interesse[i] = 2 => thread i foi autorizada pelo gerente a entrar na região crítica

#define N 10
#define N_PRIO 5

volatile int interesse[N]; /* Controla threads não prioritárias */
volatile int interesse_prio[N_PRIO]; /* Controla threads prioritárias */

void* f_gerente(void *v) {
  int i,j;

  while (1)
    for (i = 0; i < N; i++) {
      for (j = 0; j < N_PRIO; j++)  /* Atende todas as prioritárias */
        if (interesse_prio[j]) {
           { interesse_prio[j] = 2; while (interesse_prio[j] != 0); }
      if (interesse[i])  /* Atende uma não prioritária */
        { interesse[i] = 2; while (interesse[i] != 0); }
    }
  return NULL;
}

void* f_thread(void *v) {
  int thr_id = *(int*)v;

  while (1)    {
    /* Declara interesse */
    interesse[thr_id] = 1;
    /* Espera ser sua vez */
    while (interesse[thr_id] != 2);
    regiao_critica();
    interesse[thr_id] = 0;
    regiao_nao_critica();
  }
  return NULL;
}

void* f_thread_prio(void *v) {
  int thr_id = *(int*)v;

  while (1) {
    /* Declara interesse */
    interesse_prio[thr_id] = 1;
    /* Espera ser sua vez */
    while (interesse_prio[thr_id] != 2);
    regiao_critica();
    interesse_prio[thr_id] = 0;
    regiao_nao_critica();
  }
  return NULL;
}
```

Suponha a existência de uma função `main()` que cria todas *threads* e

(a) Descreva um cenário de *deadlock* que pode ocorrer com este código (use o verso desta folha se necessário)

(b) Indique alterações no código que irão eliminar a possibilidade de *deadlock* e garantir exclusão mútua.