MC504—Sistemas Operacionais
MC514—Sistemas Operacionais: Teoria e Prática

Profa. Islene Calciolari Garcia

18 de setembro de 2013

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. (1.5) **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 **três** copos foram colocados no centro da mesa. Animados com a novidade, os filósofos F0, F1, F2 resolveram primeiro disputar o copo e depois os garfos com os vizinhos. Os filósofos F3 e F4 resolveram primeiro disputar os garfos e depois o copo. 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 copo = 3;
03:
04: F0, F1, F2:
05:   while (1) {
06:     pensa();
07:     sem_wait(&copo);
08:     sem_wait(&garfo[i]);
09:     sem_wait(&garfo[(i+1)%N]);
10:     come();
11:     sem_post(&copo);
12:     sem_post(&garfo[i]);
13:     sem_post(&garfo[(i+1)%N]);
14:   }
```

```
F3, F4:
  while (1) {
    pensa();
    sem_wait(&garfo[i]);
    sem_wait(&garfo[(i+1)%N]);
    sem_wait(&copo);
    come();
    sem_post(&copo);
    sem_post(&garfo[i]);
    sem_post(&garfo[(i+1)%N]);
  }
```

2. (2.5) 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 existam múltiplas threads executando `f_thr()` simultaneamente. Você não deve fazer outras alterações no código.

```
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. Para adquirir ou liberar o lock, considere as funções:
      int pthread_mutex_lock(pthread_mutex_t *mutex);
      int pthread_mutex_unlock(pthread_mutex_t *mutex);
*/
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) {
   /* ... */
}

/* 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;
}

/* Função auxiliar para controle da execução de faz_alguma_coisa() */
int proximo() {
```

3. (3.0) **O fim das threads espertinhas?** Muitos desenvolvedores ficam revoltados com o fato de algumas implementações de `mutex_lock` permitirem que threads furem a fila, enquanto existem outras aguardando no futex (suponha para a resolução desta questão que os futexes respeitam a política First In First Out perfeitamente). Tentando mudar esta situação eles pensaram em um esquema de distribuição de senhas via um contador atômico incrementado no início da função `mutex_lock`. A variável `proxima` é incrementada na função `mutex_unlock` e indica o número da senha da próxima thread que poderá obter o lock.

   Avalie se esta função garante ou não: (a) exclusão mútua, (b) ausência de deadlock e (c) ausência de starvation. Justifique suas respostas.

```
01:  int atomic_inc (int *i); /* Incrementa *i atomicamente;
02:                           retorna o valor de *i depois da operação. */
03:   int futex_wait(void *addr, int val1);  /* Bloqueia se *addr == val1 */
04:   int futex_wake(void *addr, int n);     /* Acorda até n threads */
05:
06:  typedef struct {
07:    volatile int senha;
08:    volatile int proxima;
09:  } mutex_t;
10:
11:  void mutex_init(mutex_t *m) {
12:     m->senha = 0;
13:     m->proxima = 1;
14:  }
15:
16:  void mutex_lock(mutex_t *m) {
17:    int minha_senha = atomic_inc(m->senha);
18:    int prox = m->proxima;
19:
20:    while (prox != minha_senha)
21:        futex_wait(&m->proxima, prox);
22:  }
23:
24: void mutex_unlock(mutex_t *m) {
25:   m->proxima++;
26:   if (m->senha >= m->proxima)
27:     futex_wake(&m->proxima, 1);
28: }
```

4. (1.5) Barreiras são primitivas para sincronização de várias threads em algum ponto do programa. O livro The Little Book on Semaphores, de Allen B. Downey, em uma abordagem didática, apresenta inicialmente várias implementações erradas antes de apresentar uma uma versão correta e mais abrangente para a implementação de barreiras. Para a versão abaixo, descreva se pode ocorrer um cenário em que uma thread fica presa na barreira enquanto todas as outras conseguem sair.

```
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:     sem_post(&b->sem_barreira);
19:     atomic_dec(&b->c);
20:     if (b->c == 0)
21:       sem_wait(&b->sem_barreira);
22:  }
```

5. (1.5) Suponha que um processo pai invoca a função `fork()` e imediatamente depois vai dormir com o comando `pause()`, que interrompe a execução de um processo até que este receba um sinal. Quando começa a executar, o filho envia um sinal SIGALRM para acordar o pai que deve escrever a mensagem antes de prosseguir a sua execução.

```
01: void trata_SIGALRM(int signum) {
02:   printf("Ai que sono! Queria dormir mais...\n");
03: }
04:
05: int main() {
06:
07:   if ((pid = fork()) != 0) {
08:      signal(SIGALRM, trata_SIGALRM); /* Instalação do tratador de sinal */
09:      pause();  /* Pai espera ser acordado pelo filho */
10:   }
11:   else {
12:     faz_alguma_coisa();
13:     kill (getppid(), SIGALRM); /* Filho envia sinal para acordar o pai */
14:
15:   return 0;
16: }
```

Descreva dois problemas que impediriam este código de funcionar como esperado.