MC504—Sistemas Operacionais

Profa. Islene Calciolari Garcia

15 de abril 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) Suponha que um processo está inconformado com ter de morrer após a recepção de um sinal do tipo SIGTERM. Resolveu, então, criar um filho para garantir a sobrevivência da família nestes casos. Com base no seu conhecimento sobre a função `fork()` e tratamento de sinais, explique por que no código abaixo o filho morrerá logo após o pai. Codifique uma alteração para garantir que o filho ficará bravamente esperando um sinal e terá outro filho que fará o mesmo se receber um SIGTERM e assim sucessivamente.

```
01: void trata_SIGTERM(int signum) {
02:   if (fork()) {
03:     printf("Meu filho continuará a tradição da família!\n");
04:     signal(SIGTERM, SIG_DFL); /* Reinstala tratador padrão e */
05:     raise(SIGTERM);           /* levanta SIGTERM para encerrar execução*/
06:   }
07: }
08:
09: int main() {
10:   signal(SIGTERM, trata_SIGTERM); /* Instala o tratador de sinal */
11:   pause(); /* Interrompe a execução e aguarda um sinal */
12:   return 0;
13: }
```

2. (2.0) **O Jantar com vinho e água.** 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. Para evitar ressacas, eles também foram orientados a beber um copo de água a cada refeição.

   Como eles estão acostumados a compartilhar recursos, apenas **duas** taças para o vinho e **três** copos para água foram colocados no centro da mesa. Animados com a novidade, os filósofos F0, F1, F2, F3 resolveram primeiro disputar o copo e a taça e só depois os garfos. O filósofo F4 resolveu pegar os garfos primeiro e depois beber apenas água, não chegando a disputar uma taça. Suponha que os filósofos estão sentados em ordem FO, F1, F2, F3 e F4 e seguem o código abaixo. Este algoritmo baseado em semáforos está sujeito a *deadlock*? Justifique a sua resposta com uma argumentação que o *deadlock* é impossível ou com um cenário com a ordem de obtenção dos recurso que leva a um *deadlock*.

```
01: sem_t garfo[5] = {1,1,1,1,1};
02: sem_t taca = 2, copo = 3;
03:
04: F0, F1, F2, F3:                              F4:
05:   while (1) {                                while (1) {
06:     pensa();                                   pensa();
07:     sem_wait(&copo);                           sem_wait(&garfo[i]);
08:     sem_wait(&taca);                           sem_wait(&garfo[(i+1)%N]);
09:     sem_wait(&garfo[i]);                       sem_wait(&copo);
10:     sem_wait(&garfo[(i+1)%N]);
11:     come();                                    come();
12:     sem_post(&copo);
13:     sem_post(&taca);                           sem_post(&copo);
14:     sem_post(&garfo[i]);                       sem_post(&garfo[i]);
15:     sem_post(&garfo[(i+1)%N]);                 sem_post(&garfo[(i+1)%N]);
16:   }                                          }
```

3. (2.0) Implemente a função `proximo()` de maneira que a função `faz_alguma_coisa()` seja chamada exatamente **M** vezes para cada valor de `i` entre 0 e `NMAX` − 1 mesmo que existam múltiplas threads executando `f_thr()` simultaneamente. Quando o número máximo tiver sido atingido, a função `proximo()` deverá retornar -1. Você deve fazer as declarações de locks e variáveis auxiliares necessárias. Não é preciso escrever o trecho de código que inicializa o(s) lock(s) declarado(s).

```
#define NMAX 100
#define M    20

/*  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);
*/

/* Função auxiliar que deve ser invocada exatamente M vezes 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()) != -1) {
     faz_alguma_coisa(i);
  }
  return NULL;
}

/* Declarações:    */




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

4. (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 (suponha para a resolução desta questão que os futexes respeitam a política First In First Out perfeitamente). Tentando mudar esta situação, eles utilizaram um contador de threads esperando `n_espera` que é incrementado atomicamente. Também utilizaram uma variável inteira `fila` para controle de uma fila obrigatória para as threads que devem esperar. Analise o código abaixo:

```
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 val);  /* Bloqueia se *addr == val */
04:   int futex_wake(void *addr, int n);     /* Acorda até n threads */
05:
06:  typedef struct {
07:    volatile int fila;       /* Controle para fila do futex    */
08:    volatile int n_espera;   /* Contador de threads esperando */
09:  } mutex_t;
10:
11:  void mutex_init(mutex_t *m) {
12:     m->n_espera = 0;
13:     m->fila = 1;
14:  }
15:
16:  void mutex_lock(mutex_t *m) {
17:    if (atomic_inc(n_espera) > 0) /* Se não é a única thread, */
18:      futex_wait(&m->fila, 1);     /* entra na fila e espera sua vez. */
19:  }
20:
21: void mutex_unlock(mutex_t *m) {
22:   if (atomic_dec(n_espera) > 1) /* Se há mais threads na espera, */
23:     futex_wake(&m->fila, 1);    /* acorda uma thread.            */
23: }
```

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

5. (2.0) 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 (i) por que esta versão não funciona sempre e (ii) qual é o único cenário que permite que todas as threads passem pela barreira.

```
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:  }
```