MC504/MC514—Sistemas Operacionais

Profa. Islene Calciolari Garcia

Prova 2

28 de abril 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.

Funções atômicas:

```
int bool_cmpxchg (int *i, int old, int new);
    /* se *i == old, *i recebe new; caso contrário *i não é alterado;
       retorna 1 se executou a troca ou 0 caso contrário. */

int atomic_inc (int *i); /* Incrementa *i atomicamente;
                            retorna o valor de *i antes da operação. */

int atomic_dec (int *i); /* Decrementa *i atomicamente;
                            retorna o valor de *i antes da operação. */
```

1. (2.0) Variáveis de condição servem como um mecanismo de sincronização entre processos. Dentro deste contexto, explique o funcionamento da função `pthread_cond_wait()` e a necessidade da passagem de um mutex lock como parâmetro.

```
int pthread_cond_wait(pthread_cond_t *cond, pthread_mutex_t *mutex);
```

2. (2.0) **Incremento atômico se positivo.** Observe o código abaixo:

```
/* Executa atomicamente *i = (*i - 1) se *i > 0;
   retorna o valor de *i antes da operação. */
int atomic_dec_if_positive(int *i) {
  if (*i > 0)
    return atomic_dec(i);
  else
    return 0;
}
```

(a) Considere que *i é inicializada com um valor positivo e depois disso as threads da aplicação fazem apenas operações de decremento atômico. Descreva um caso em que o valor de *i pode atingir valores menores do que zero. (b) Reescreva a função `atomic_dec_if_positive` de maneira que funcione corretamente. Você só poderá utilizar operações atômicas como `bool_cmpxchg` e `atomic_dec`. Não é permitida a utilização de outras primitivas de sincronização

3. (2.0) **Mais uma tentativa de acabarmos com as 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 no futex (suponha que os futexes respeitam a política First In First Out perfeitamente). Tentando mudar esta situação, eles utilizaram um contador de threads esperando `n_waiters` que é incrementado atomicamente. Eles aproveitaram este contador para evitar algumas chamadas à operação `futex_wake()`. Vale ressaltar que o valor 0 no campo `val` indica lock livre e o valor 1 indica lock travado.

```
01:  typedef struct {
02:    volatile int val;
03:    volatile int n_waiters;
04:  } mutex_t;
05:
06:  void mutex_init(mutex_t *m) {
07:     m->val = m->n_waiters = 0;
08:  }
09:
10:  void mutex_lock(mutex_t *m) {
11:  int nwaiters_antes = atomic_inc(m->nwaiters);
12:
13:  if (nwaiters_antes == 0)
14:     m->val = 1;   /* Se não há threads na fila, basta pegar o lock! */
15:  else
16:    do {
17:       futex_wait(&m->val, 1);  /* Bloqueia se m->val == 1 (lock não está livre) */
18:    } while (!bool_cmpxchg(&m->val, 0, 1)); /* até pegar o lock  */
19: }
20:
21: void mutex_unlock(mutex_t *m) {
22:   m->val = 0;
23:   if (atomic_dec(m->n_waiters) > 1) /* Há mais threads na espera */
24:     futex_wake(&m->val, 1);  /* Acorda a prõxima thread */
25: }
```

Este algoritmo garante exclusão mútua? Esta abordagem eliminou o problema das threads espertinhas? Justifique as suas respostas.

4. (2.0) Barreiras são primitivas para sincronização de várias threads em algum ponto do programa. A versão abaixo é uma variação dos códigos apresentados no livro The Little Book on Semaphores, de Allen B. Downey. Indique se esta implementação funciona adequadamente para (a) barreiras simples, quando todas as threads da aplicação se sincronizam apenas uma vez, (b) barreiras reutilizáveis, quando todas as threads da aplicação se sincronizam várias vezes sem reinicializar a barreira e (c) barreiras restritas, quando o número de threads da aplicação é superior ao tamanho da 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:        for (int i = 0; i < N; i++)
17:          sem_post(&b->sem_barreira);
18:     sem_wait(&b->sem_barreira);
19:  }
```

5. (2.0) No código abaixo, um processo avô gera um processo pai que gera um processo filho. O filho envia um sinal para o pai que deve ser repassado para o avô. Supondo ue não há interferência de processos externos, descreva um cenário em que a transmissão dos sinais não ocorre como indicado acima.

```
01: void trata_SIGUSR1_avo(int signum) {
02:   printf("Recebi SIGUSR1! \n");
03: }
04:
05: void trata_SIGUSR1_pai(int signum) {
06:  printf("Vou repassar SIGUSR1 para o meu pai...\n");
07:  kill(getppid(), SIGUSR1);
08: }
09:
10: int main() {
11:
12:   if (fork()) { /* Processo avo */
13:      signal(SIGUSR1, trata_SIGUSR1_avo); /* Instala tratador de sinal */
14:      pause(); /* Aguarda um sinal */
15:   } else
16:      if (fork()) {   /* Processo pai */
17:        signal(SIGUSR1, trata_SIGUSR1_pai); /* Instala tratador de sinal */
18:        pause(); /* Aguarda um sinal */
19:      }
20:      else   /* Processo filho */
21:         kill(getppid(), SIGUSR1); /* Envia SIGUSR1 para o seu pai  */
22:
23:   return 0;
24: }
```