MC514—Sistemas Operacionais: Teoria e Prática

Lista de exercícios I

Data de entrega: 3 de maio

A nota desta lista valerá como um projeto de laboratório

1. Enumere quais recursos são gerenciados separadamente para processos e threads pelo sistema operacional.
2. É possível que uma thread possa causar um erro de execução em uma outra thread? Explique.
3. Explique a finalidade da função `pthread_join()`. Se esta função não existisse, como você faria para simular o seu comportamento?
4. Dê um exemplo (trecho de código) onde pode ocorrer uma condição de corrida (disputa) entre duas threads.
5. A abordagem da alternância para acesso a uma região crítica de código tem uma limitação clara quando uma thread é mais veloz do que a outra. O algoritmo abaixo poderia ser usado para resolver este problema?

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

6. Considere o algoritmo do desempate proposto por Peterson:

```
int vez = 0, interesse[2] = {false, false};
Thread_i:
   int adv = i^1; /* Id da thread adversária */
   while (true)
     interesse[i] = true;
     vez = i;
     while (vez == i && interesse[adv]) ;
     regiao_critica();
     interesse[i] = false;
```

   (a) Caso trocássemos a ordem da atribuição da vez e da indicação de interesse no início do algoritmo, ele continuaria correto?

(b) Um programador achou este código pouco intuitivo, pois neste algoritmo *quem tem a vez é que espera*. Ele implementou a seguinte variante, na qual *quem tem a vez não espera*, alterando o teste do while para `while (vez != i && interesse[adv]) ;`
Esta versão garante exclusão mútua? Caso contrário, mostre um cenário de erro.

7. Outro programador precisava de uma solução para o algoritmo do desempate que funcionasse corretamente para N threads. Ele implementou a seguinte idéia:

```
int vez = 0, interesse[N] = {false, ..., false};
Thread_i:
   while (true)
     interesse[i] = true;
     vez = i;
     while (vez == i && existe j!= i tal que interesse[j]) ;
     regiao_critica();
     interesse[i] = false;
```

Este algoritmo garante exclusão mútua? Existe algum cenário de *deadlock*?

8. Quando viu que a solução anterior não funcionava, esse programador resolveu implementar a seguinte versão:

```
int interesse[N] = {false, ... , false},
    fase[N] = {-1, ..., -1}, vez[N-1];
Thread i:
  interesse[i] = true;
  for (f = 0; f < N-1; f++)
     fase[i] = f;
     vez[f] = i;
     for (j = 0; j < N && vez[f] == i; j++ )
        if (j != i && interesse[j])
           while (f <= fase[j] && vez[f] == i);
  regiao_critica();
  interesse[i] = false;
  fase[i] = -1;
```

Você saberia explicar para ele a razão dos testes `vez[f] == i`? Sua existência é essencial para o funcionamento do algoritmo ou pode ser encarada apenas como uma otimização?

9. (Tanenbaum) Considere um computador sem a instrução TSL, mas que tenha uma instrução que troque o conteúdo de um registrador e de uma palavra de memória em uma ação única e indivisível. Essa instrução pode ser usada para a implementação de uma rotina para entrada na região crítica semelhante à seguinte?

```
entra_RC:
    TSL RX, lock
    CMP RX, #0
    JNE entra_RC
    RET
```

10. Explique o funcionamento da função `pthread_cond_wait()`. Por que esta função tem um `mutex lock` como parâmetro?

11. Esta solução para o problema dos filósofos famintos descrita no livro do Tanenbaum permite o máximo de paralelismo para o sistema.

```
#define N 5                     /* number of philosophers */
#define LEFT  (i+N-1) % N       /* number of i's left neighbor */
#define RIGHT (i+1) % N         /* number of i's right neighbor */
#define THINKING 0              /* philosopher is trying to get forks */
#define HUNGRY 1                /* philosopher is hungry */
#define EATING 2                /* philosopher is eating */

semaphore mutex = 1, semaphore s[N] = {0, ..., 0}

void philosopher(int i) {
  while (TRUE) {
    think();
    take_forks(i);
    eat();
    put_forks(i);
  }
}

void take_forks(int i) {          void put_forks(int i) {
  wait(&mutex);                     wait(&mutex);
  state[i] = HUNGRY;                state[i] = THINKING;
  test(i);                          test(LEFT);
  signal(&mutex);                   test(RIGHT);
  wait(&s[i]);                      signal(&mutex);
}

void test(i) {
  if (state[i] == HUNGRY && state[LEFT] != EATING && state[RIGHT] != EATING) {
    state[i] = EATING;
    signal(&s[i]);
  }
}
```

(a) Reescreva esta versão utilizando *mutex locks* e variáveis de condição.

(b) Em sala de aula, foi mostrado um cenário com 5 filósofos no qual um deles morria de fome. Mostre um cenário com 8 filósofos no qual dois morrem de fome enquanto os outros 6 fazem refeições regularmente.

(c) Suponha que um programador foi desatento ao implementar esta solução e inverteu a ordem das duas últimas linhas da função `take_forks()` (colocou o `wait` antes do `signal`). Qual problema poderá ocorrer?

12. Uma característica interessante do algoritmo da padaria é que os clientes são servidos na ordem de chegada. Escreva uma versão deste algoritmo que use *mutex locks* e variáveis de condição.

```
escolhendo[N] = { false, false, ..., false }
num[N] = { 0, 0, ..., 0 }

Thread i:
escolhendo[i] = true;
num[i] = max (num[0]...num[N-1]) + 1
escolhendo[i] = false;
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;
```

13. Considere o seguinte código para o caso de um *buffer* controlado por um único produtor e vários consumidores:

```
cond_t pos_vazia, pos_ocupada;
mutex_t lock_v, lock_o, lock_i, lock_f;
int nv = N, no = 0, i = 0, f = 0;

Produtor:                          Consumidores:
  mutex_lock(&lock_v);                 mutex_lock(&lock_o);
  if (nv == 0)                         while (no == 0)
     cond_wait(&pos_vazia,                cond_wait(&pos_ocupada,
               &lock_v);                            &lock_o);
  nv--;                                no--;
  mutex_unlock(&lock_v);               mutex_unlock(&lock_o);
  mutex_lock(&lock_f);                 mutex_lock(&lock_i);
  f = (f+1)%N;                         i = (i+1)%N;
  buffer[f] = produz();                consome(buffer[i]);
  mutex_unlock(&lock_f);               mutex_unlock(&lock_i);
  mutex_lock(&lock_o);                 mutex_lock(&lock_v);
  no++;                                nv++;
  cond_signal(&pos_ocupada);           cond_signal(&pos_vazia);
  mutex_unlock(&lock_o);               mutex_unlock(&lock_v);
```

(a) É necessário usar um comando `while` antes do chamada a `cond_wait()` no consumidor? Pode ser usado apenas um comando `if` antes da chamada a `cond_wait()` no produtor?

(b) A chamada a `cond_signal()` no consumidor poderia ser trocada por uma chamada a `cond_broadcast()`?

(c) Existe o risco de *starvation* de um consumidor no código descrito? E na versão com `cond_broadcast`?