MC514—Sistemas Operacionais: Teoria e Prática

Lista de exercícios I

1. Suponha que você gostaria de criar várias threads, mas deixá-las aguardando uma condição antes de iniciarem seu processamento. Como você faria para implementar esta espera (i) utilizando espera ocupada, (ii) utilizando futexes, (iii) utilizando semáforos e (iv) utilizando mutex_locks e variáveis de condição?

2. Explique a finalidade da função `pthread_join()`. Se esta função não existisse, como você faria para simular o seu comportamento (i) utilizando espera ocupada, (ii) utilizando futexes, (iii) utilizando semáforos e (iv) utilizando mutex_locks e variáveis de condição?

3. A execução de uma thread pode causar um erro de execução em uma outra thread? Explique.

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

5. Explique o que são locks recursivos e quando eles podem ser utilizados com vantagem.

6. Considere o algoritmo do desempate proposto por Peterson:

```
1: int ultimo = 0, interesse[2] = {false, false};
2: Thread_i:
3: int adv = i^1; /* Id da thread adversária */
4: while (true)
5:   interesse[i] = true;
6:   ultimo = i;
7:   while (ultimo == i && interesse[adv]) ;
8:   regiao_critica();
9:   interesse[i] = false;
```

   (a) Caso trocássemos a ordem das linhas 5 e 6 o algoritmo continuaria correto?

   (b) Um programador achou que seria mais interessante codificar este código utilizando o conceito de `primeiro` e não de `ultimo`. Ele fez as mesmas inicializações, mas modificou o teste do while para `while (primeiro != i && interesse[adv]) ;` Mostre um cenário no qual esta versão não garante exclusão mútua.

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

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

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

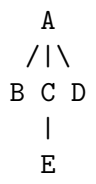
8. Quando viu que a solução anterior não funcionava, esse programador resolveu implementar a seguinte versão:

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

Elimine os comandos de espera ocupada utilizando futexes.

9. Explique o funcionamento da função `fork`. Quais são as alternativas para um processo pai esperar pelo morte de um processo filho?

10. Escreva um trecho de código que utiliza a função `fork()` e gera a seguinte hierarquia de processos:



11. Explique a razão pela qual um tratador de sinais pode encontrar estruturas de dados inconsistentes.

12. 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 (i) *mutex locks* e variáveis de condição e (ii) utilizando futexes.

(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?