MC514—Sistemas Operacionais: Teoria e Prática

Lista de exercícios I

Data de entrega: 5 de maio

A nota desta lista valerá como um projeto de laboratório

1. Após uma thread executar `pthread_create()`, a thread criada começa a executar imediatamente? Imagine que você gostaria de separar as operações de criar uma thread e de começar a executá-la. Como você faria para simular este comportamento?

2. Explique a finalidade da função `pthread_join()`. Se esta função não existisse, como você faria para simular o seu comportamento? Seria eficiente utilizar espera ocupada?

3. É possível que a execução de uma thread cause um erro de execução em uma outra thread? Explique.

4. Explique a razão pela qual o código abaixo pode gerar um erro de execução.

```
typedef struct S {
  int i; int j;
} S;

S f_thread(S s) {
  printf("Nova thread %d %d.\n", s.i, s.j);
  return s;
}

int main() {
  pthread_t thr;
  pthread_create(&thr, NULL, (void * (*)(void *)) f_thread, NULL));
  pthread_join(thr, NULL);
  return 0;
}
```

5. (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
```

6. 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;
    acessa_regiao_critica();
    vez = -1;
```

7. 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.

   (c) Você tem alguma sugestão para deixar o código correto e com uma interpretação simples?

8. 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*?

9. 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?

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

11. 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;
```

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) {
  wait(&mutex);
  state[i] = HUNGRY;
  test(i);
  signal(&mutex);
  wait(&s[i]);
}

void put_forks(int i) {
  wait(&mutex);
  state[i] = THINKING;
  test(LEFT);
  test(RIGHT);
  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?

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

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

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

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

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

```
  A
 /|\
B C D
  |
  E
```

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

18. 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.

```
void trata_SIGALRM(int signum) {
  printf("Ai que sono! Queria dormir mais...\n");
}

int main() {

  if ((pid = fork()) != 0) {
    signal(SIGALRM, trata_SIGALRM); /* Instalação do tratador de sinal */
    pause();  /* Pai espera ser acordado pelo filho */
  }
  else
    kill (getppid(), SIGALRM); /* Filho envia sinal para acordar o pai */

  return 0;
}
```

Descreva problemas que impediriam este código de funcionar corretamente.