

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. Um programador estava utilizando futex com uma variável inteira como base que tinha valor sempre constante. No entanto, o programador não pretendia que thread as dormissem todas as vezes. Você acha que o código deste programador pode estar correto?
4. A execução de uma thread cause um erro de execução em uma outra thread? Explique.
5. Explique o funcionamento da função `pthread_cond_wait()`. Por que esta função tem um `mutex lock` como parâmetro?
6. Explique o que são locks recursivos e quando eles podem ser utilizados com vantagem.
7. O algoritmo abaixo garante exclusão mútua?

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

8. 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 `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.
9. 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*?

10. 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`.

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) {
    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 (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?

12. 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:
<code>mutex_lock(&lock_v);</code>	<code>mutex_lock(&lock_o);</code>
<code>if (nv == 0)</code>	<code>while (no == 0)</code>
<code>cond_wait(&pos_vazia,</code>	<code>cond_wait(&pos_ocupada,</code>
<code>&lock_v);</code>	<code>&lock_o);</code>
<code>nv--;</code>	<code>no--;</code>
<code>mutex_unlock(&lock_v);</code>	<code>mutex_unlock(&lock_o);</code>
<code>mutex_lock(&lock_f);</code>	<code>mutex_lock(&lock_i);</code>
<code>f = (f+1)%N;</code>	<code>i = (i+1)%N;</code>
<code>buffer[f] = produz();</code>	<code>consume(buffer[i]);</code>
<code>mutex_unlock(&lock_f);</code>	<code>mutex_unlock(&lock_i);</code>
<code>mutex_lock(&lock_o);</code>	<code>mutex_lock(&lock_v);</code>
<code>no++;</code>	<code>nv++;</code>
<code>cond_signal(&pos_ocupada);</code>	<code>cond_signal(&pos_vazia);</code>
<code>mutex_unlock(&lock_o);</code>	<code>mutex_unlock(&lock_v);</code>

- (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`?
13. Explique o funcionamento da função `fork`. Quais são as alternativas para um processo pai esperar pelo morte de um processo filho?
14. Escreva um trecho de código que utiliza a função `fork()` e gera a seguinte hierarquia de processos:

```

      A
    /  \
   B C  D
    |
    E
```

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

16. 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.