Universidade Federal do Ceará
Sistemas Operacionais
Prof. Ms. Rafael Ivo

Lista de Exercícios 3 – Comunicação de Processos

1) Explique o que é condição de corrida.

2) O que é região crítica? E exclusão mútua?

3) Qual o problema com a solução que desabilita as interrupções para implementar a exclusão mútua?

4) O que é espera ocupada e qual o seu problema?

5) Explique porque uma simples variável compartilhada indicando que um processo está na região crítica, não é suficiente para garantir a exclusão mútua.

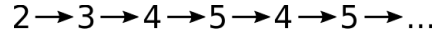
6) A técnica de alternância obrigatória garante a exclusão mútua? Qual o problema dessa técnica?

7) Por que fazer um processo dormir e depois acordar é mais eficiente do que as soluções de espera ocupada?

8) Suponha dois processos P1 e P2 com regiões críticas correspondentes.
a) Implemente a solução com variável de impedimento e espera ocupada.
b) Implemente a solução com variável de impedimento e funções **`sleep`** e **`wakeup`**.
c) Mostre que se a leitura da variável de impedimento e as instruções **`sleep`** e **`wakeup`** não forem indivisíveis, essas soluções não funcionam.

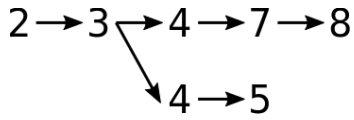
9) Um grafo de precedência é um grafo direcionado em que a relação `(a) → (b)` indica que o acontecimento `(a)` certamente acontece antes de `(b)`. Neste exercício, os nós do grafo devem representar as instruções numeradas no código abaixo. Desta forma, o grafo de precedência representará a ordem em que as instruções serão executadas. Observe que se o programa não tivesse **`fork()`**, o grafo seria linear, pois a execução desse programa seria uma sequência de instruções (com desvios ou não). Ex:

```
1. int main(){
2.     int a;
3.     a = 5;
4.     while(a > 0){
5.         a = a - 1;
6.     }
7. }
```



No entanto, sempre que há um **`fork()`**, passamos a ter execuções em paralelo. Ex:

```
 1. int main(){
 2.     int pid;
 3.     pid = fork();
 4.     if(pid == 0){
 5.         printf("Sou o processo filho!");
 6.     }
 7.     else if(pid > 0){
 8.         printf("Sou o processo pai!");
 9.     }
10.}
```



Desenhe o grafo de precedência referente ao código a seguir:

```
 1. int pid;
 2.
 3. void helloWorld(int i){
 4.     printf("Ola mundo! ”);
 5.     printf(“Sou o processo %d.\n",i);
 6.     exit();
 7. }
 8.
 9. int main(){
10.     pid = fork();
11.     if(pid == 0){
12.         pid = getpid();
13.         helloWorld(pid);
14.     }
15.     else if(pid > 0){
16.         wait(pid);
17.         pid = getpid();
18.         helloWorld(pid);
19.     }
20. }
```

10) Construa um grafo de precedência como da questão anterior, mas considerando apenas as instruções de escrita do código abaixo.

```
void f1(){
    printf("2");
    exit();
}

void f2(){
    printf("3");
    printf("4");
    printf("5");
    exit();
}

void f3(){
    printf("6");
    printf("7");
    exit();
}

int main(){
    printf("1");
    int pid = fork();
    if(pid == 0) f2();
    else if(pid > 0){
        pid = fork();
        if(pid == 0) f3();
        else if(pid > 0) f1();
    }
}
```

11) Adicione semáforos ao programa da questão anterior, e as respectivas chamadas às suas operações (**down** e **up**), de modo que estas novas restrições de precedência sejam satisfeitas:

I. O 2 deve ser escrito antes que o 4
II. O 2 deve ser escrito antes que o 7
III. O 7 deve ser escrito antes que o 5

12) Considere o problema dos leitores e escritores. Há diversos processos que eventualmente fazem acessos de leitura a uma base de dados e diversos processos que eventualmente fazem acessos de escrita à mesma base de dados. Vários acessos de leitura podem ocorrer simultaneamente, mas um acesso de escrita não pode ocorrer simultaneamente com nenhum outro acesso. Considere o código a seguir para os processos de leitura e escrita. Suponha que todos os semáforos (**ler**, **escrever** e **mutex**) são iniciados com valor 1:

```
//LEITORES
 1. while(TRUE){
 2.     down(ler);
 3.     down(mutex);
 4.     nLeitores++;
 5.     if(nLeitores == 1) down(escrever);
 6.     up(mutex);
 7.     up(ler);
 8.
 9.     ler_banco_de_dados();
10.
11.     down(mutex);
12.     nLeitores--;
13.     if(nLeitores == 0) up(escrever);
14.     up(mutex);
15.
16.     consumir_dados();
17. }
```

```
//ESCRITORES
 1. while(TRUE){
 2.     produz_dados();
 3.
 4.     down(ler);
 5.     down(escrever);
 6.
 7.     altera_banco_de_dados();
 8.
 9.     up(escrever);
10.     up(ler);
11. }
```

a) Essa solução pode levar a ***starvation*** dos escritores? (Pode acontecer de um escritor nunca conseguir o acesso à base de dados?) Explique sua resposta, usando os números das linhas de código para se referir aos passos do programa.

b) Explique o papel do semáforo **mutex**. Dê um exemplo de problema que poderia ocorrer caso as operações sobre ele fossem retiradas.

13) Resolva o problema dos leitores e escritores usando monitor. O monitor deve possuir as funções **leitorQuerLer()**, **leitorSaiu()**, **escritorQuerEscrever()**, **escritorSaiu()**. Crie as variáveis que necessitar. As únicas primitivas de sincronização são **wait** e **signal**.

14) Suponha o seguinte problema: uma thread produtor gera triângulos ou quadrados de forma aleatoria. Após gerar o polígono, ele armazena o novo poligono em uma variável compartilhada **P**. Há duas outras threads consumidores: uma delas consome apenas triângulos (aplicando a cor amarela) e outra consome apenas quadrados (aplicando a cor verde). Adicione semáforos ao código abaixo de forma a sincronizar o funcionamento do produtor e dos consumidores apropriadamente. (***Obs***: A classe Poligono possui um método **`nLados()`** que retorna a quantidade de lados do polígono).

```
Poligono P;

void *produtor(){
    while(1){
        Poligono novo = geraPoligono();
        P = novo;
    }
}

void *consumidorT(){
    while(1){
        Poligono novo = P;
        novo.aplicaCor(AMARELO);
    }
}

void *consumidorQ(){
    while(1){
        Poligono novo = P;
        novo.aplicaCor(VERDE);
    }
}
```

15) Seja o seguinte problema: Uma barbearia é composta por uma sala de espera com n cadeiras e um barbeiro com uma cadeira de barbear. Quando não há clientes a serem atendidos, o barbeiro dorme. Se um cliente entrar na barbearia e todas as cadeiras estiverem ocupadas, ele sairá da barbearia. Se o barbeiro estiver ocupado, mas cadeiras estiverem disponíveis, o cliente sentará em uma das cadeiras livres. Se o barbeiro estiver dormindo, o cliente o acordará. O programa abaixo, resolve a sincronização entre o processo barbeiro e os processos clientes. Os semáforos **`clientes`**, **`barbeiro`** e **`mutex`** e a variável **`vagas`** são compartilhadas entre os processos.

a) Verifique a correturde do programa, testando diversos pontos onde o SO poderia interromper dado processo.

b) Se não houvesse o semáforo **`mutex`**, que problema poderia acontecer? (Mostre um caso, com valores de variáveis e semáforos, onde ocorre um erro)

```
#define NUMERO_CADEIRAS 3

semaforo clientes = 0;
semaforo barbeiro = 0;
semaforo mutex = 1;
int vagas = NUMERO_CADEIRAS;

void processoBarbeiro(){
    while(1){
        down(clientes);
        down(mutex);
        vagas++;
        up(barbeiro);
        up(mutex);
        printf("Barbeiro cortando cabelo");
    }
}

void processoCliente(){
    down(mutex);
    if(vagas > 0){
        printf("Cliente chegou");
        vagas--;
        up(clientes);
        up(mutex);
        down(barbeiro);
        printf("Cliente tendo cabelo cortado");
    }
    else{
        up(mutex);
        printf("Cliente desistiu");
    }
}
```