Universidade Federal da Fronteira Sul (UFFS)
Campus Chapecó - Curso de Ciência da Computação
Prof. Marco Aurélio Spohn
CCR: *Sistemas Operacionais – Exercícios (primeira parte da CCR)*

1. ***SO Kid*** se autodenomina um dos maiores especialistas em sistemas operacionais. Ele afirma que no escalonamento de processos por loteria (*lottery scheduling*) com um total de ***N*** bilhetes distribuídos entre os processos prontos (*ready*) para execução, a probabilidade de um processo pronto com ***x*** bilhetes ser sorteado pelo escalonador é equivalente a $\frac{x}{N}$. **SO Kid está correto? Explique**.

2. Considerando a solução de exclusão mútua com **espera ociosa baseada em chaveamento obrigatório** (abaixo exemplo de código para 2 processos: **(a)** processo **0**; **(b)** processo **1**), ***SO Kid*** pede que você implemente uma versão para tratar quatro (**4)** processos também assumindo espera ociosa baseada em chaveamento obrigatório. **Apresente a solução em um fragmento de código/pseudo-código (semelhante ao apresentado abaixo), explicando-o**.

```
while (TRUE) {
    while (turn !=0)          /* laço */ ;
    critical_region( );
    turn = 1;
    noncritical_region( );
}
```

(a)

```
while (TRUE) {
    while (turn !=1)          /* laço */ ;
    critical_region( );
    turn = 0;
    noncritical_region( );
}
```

(b)

3. Em um aplicativo *multi-thread* desenvolvido por ***SO Kid***, as *threads* concorrem pelo acesso aos recursos **X, Y, Z, T e W**. A utilização de cada recurso requer acesso exclusivo (*i.e.*, caso o recurso esteja disponível, a *thread* que conseguir acesso ao recurso adquire o *lock* sobre o mesmo, impedindo o acesso às demais *threads*). ***SO Kid*** definiu algumas regras a serem seguidas pelas *threads* a fim de se prevenir *deadlocks*. As regras são:

I) Caso a *thread* consiga acesso a **Z**, poderá tentar acessar o recurso **W** e nada mais;
II) Caso a *thread* tenha conseguido acesso a **X**, poderá somente tentar acesso aos recursos **T** e **W** mas, caso decida primeiro acessar **W,** não poderá mais tentar acessar **T**;
III) Caso a *thread* tenha conseguido acesso a **Y** e **X** (possível, desde que solicitado/obtido nessa ordem), poderá tentar acessar apenas **T**.

A solução de *SO Kid* previne impasses em sua aplicação? Explique.

OBS.: a) cada *thread* pode manter múltiplos recursos simultaneamente (**desde que possível segundo as regras estabelecidas**); **b)** assume-se que as *threads* acessam os recursos com frequência mas sempre por um tempo finito (caso consigam acesso, naturalmente); **c) considere apenas operações permitidas segundo as regras estabelecidas.**

4. **Para cada um** dos seguintes endereços binários virtuais, calcule o número da página virtual e o deslocamento (*offset*) considerando páginas de **256 bytes**. **Apresente o desenvolvimento do cálculo em decimal**.

(a) 0011 1000 0110 1111

(b) 1001 0000 0001 1011

5. Marque V (verdadeiro) ou F (falso) para cada uma das assertivas abaixo:

( ) No escalonamento não preemptivo de processos/*threads* o escalonador é ativado periodicamente e pode decidir pela alocação da CPU para outro processo/*thread*.
( ) A arquitetura *microkernel* exige que todos os serviços suportados pelo sistema operacional sejam carregados no momento da inicialização do SO.
( ) As instruções de configuração do intervalo de interrupção do TIMER pertencem ao conjunto de instruções privilegiadas.
( ) Quando se emprega gerenciamento de memória baseada em paginação, tem-se fragmentação externa de memória.

6. SO Kid implementou o problema do jantar dos filósofos em linguagem C na plataforma *Linux*. No entanto, observou que algo está errado pois, de fato, o sistema não evolui (aparentemente, alguns filósofos permanecem comendo para sempre e outros permanecem famintos para sempre). **Identifique e comente o problema, apresentando uma solução no próprio código abaixo.**

```
#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/time.h>
#include <errno.h>
#include <semaphore.h>
#include <fcntl.h>

#define N 5
int left(int id);
int right(int id);
void *philosopher(void *data);
void take_forks(int id);
void put_forks(int id);
void test(int id);
#define THINKING 0
#define HUNGRY 1
#define EATING 2
int state[N];
sem_t mutex;
sem_t s[N];

int main(void) {
  int i;
  pthread_t tids[N];

  sem_init(&mutex, 0, 1);
  for(i=0;i<N;i++)
  {
    sem_init(&s[i], 0, 0);
    state[i]=THINKING;
   }

   for(i=0; i<N; i++) {
    int *j = malloc(sizeof(int));
    *j=i;
    printf("\n creating philosopher %d \n",*j);
    pthread_create(&tids[i], NULL, philosopher, (void *)j);
   }


   for(i=0; i<N;i++) {
    pthread_join(tids[i], NULL);
    printf("Thread id %ld returned\n", tids[i]);
  }
  return(1);

}
```

```
void *philosopher(void *data){
     int id = *((int *) data);
     while(1){
        printf("\n Philosopher %d is thinking\n",id);
        sleep(2); // apenas para  passar tempo “pensando”
        take_forks(id);
        sleep(2); // apenas para passar tempo “comendo”
        printf("\n Philosopher %d is eating\n",id);
        put_forks(id);
     }
     pthread_exit(NULL);
}

int left(int id){
   return((id+N-1)%N);
}

int right(int id){
   return((id+1)%N);
}

void take_forks(int id){
    sem_wait(&mutex);
    test(id);
    sem_post(&mutex);
    sem_wait(&s[id]);
}


void put_forks(int id){
    sem_wait(&mutex);
    state[id]=THINKING;
    test(left(id));
    test(right(id));
    sem_post(&mutex);
}


void test(int id){
  if(state[id]==HUNGRY && state[left(id)]!=EATING &&
state[right(id)]!=EATING)
   {
         state[id]=EATING;
    }
}
```

7. Considere um grupo de três *threads* disputando acesso a uma variável global. **Exige-se que se garanta a exclusão mútua ao se atualizar a variável global**. Após inicializar a execução do programa de ***SO Kid***, observa-se que o mesmo não produz o resultado esperado e sequer finaliza. **Utilizando-se do fragmento principal do código de SO Kid (abaixo), identifique o(s) problema(s) e apresente a(s) respectiva(s) correção(ões) e explicação(ões) NO PRÓPRIO CÓDIGO!**

```
void *mythread(void *data);
pthread_mutex_t count_mutex = PTHREAD_MUTEX_INITIALIZER;

#define N 3 // number of threads
#define MAX 10
int global = 0;

int main(void) {
   pthread_t tids[N];
   int i;

     for(i=0; i<N; i++) {
         pthread_create(&tids[i], NULL, mythread, NULL);
     }

     for(i=0; i<N; i++) {
         pthread_join(tids[i], NULL);
         printf("Thread id %ld returned\n", tids[i]);
     }
     return(1);
}

void *mythread(void *data) {

   while(global < MAX) {
      pthread_mutex_lock(&count_mutex);
      global++;
      printf("Thread ID%ld: global is now %d.\n", pthread_self(), global);
      sleep(2);
   }

   pthread_exit(NULL);
}
```