Universidade Federal da Fronteira Sul (UFFS)
Campus Chapecó - Curso de Ciência da Computação
Prof. Marco Aurélio Spohn
CCR: *Sistemas Operacionais - Primeira Prova (P1) – 2021.2 - 07/03/2022*

4,6

Nome: Alex Sandro Zanoleon

- **MANTER DESLIGADO E GUARDADO CELULARES, COMPUTADORES E CALCULADORAS.**
- **A COMPREENSÃO DAS QUESTÕES FAZ PARTE DA AVALIAÇÃO!!!**
- **RESPONDA À CANETA NOS ESPAÇOS RESERVADOS**

1. **[1,0 ponto]** ***SO Kid*** se autodenomina um dos maiores especialistas em sistemas operacionais. Ele afirma que no escalonamento de processos por loteria (*lottery scheduling*) os processos com apenas um (1) bilhete sempre entram em um estado de inanição (*starvation*). **SO Kid está correto? <u>Explique</u>.**

Está correto. Ter um ou mais bilhetes de loteria não tira a chance daquele processo ser sorteado, como qualquer um. ??

??

0,5

2. **[1,5 ponto]** 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 **4** processos. **<u>Apresente a solução em um fragmento de código/pseudo-código, explicando-o</u>**.

```
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)

1,0

```
#define N 4
int num;
while (TRUE){
    while (turn != num);
    critical_region();
    turn = (turn+1) % N;
    noncritical_region();
}
```

??

QUAIS OS VALORES P/ TURN ??

3. **[2,0 pontos]** 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 evitar *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.**

Previne impasses. A solução apresentada sempre faz com que as threads tentem o recurso, caso não consigam, desistam e partam para uma próxima, ou mesma tentam outra recurso que por ventura possa estar livre.

4. **[1,0 ponto] 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

page | offset

64 32 16 | 256 128 64 32 16 8 4 2 1

64 + 32 + 16 = 112

256 + 128 + 64 + 4 + 2 + 1 = 455

$256 = 2^8$

page = 112
offset = 455

(b) 1001 0000 0001 1011

256 128 64 32 | 32 16 8 4 2 1

256 + 32 = 288

32 + 16 + 4 + 1 = 53

page = 288
offset = 53

5. **[1,0 ponto]** Marque V (verdadeiro) ou F (falso) para cada uma das assertivas abaixo:

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

6. **[2,0 pontos]** Considere o seguinte problema envolvendo programação *multithread*, *mutexes* e semáforos:

"- A partir da thread principal criar N threads;
- Cada thread executa, basicamente, a mesma tarefa que consiste em incrementar uma variável global inicializada com valor zero (0); no entanto, a cada rodada envolvendo todas as threads, cada thread incrementa a variável global uma única vez. Além disso, a alternância entre as threads dá-se sempre em ordem crescente de identificadores. Assumir identificadores das threads incrementais iniciando-se a primeira thread com ID=0. O incremento se encerra quando for atingido um valor máximo (MAX). Exemplo: assumindo que existem 3 threads (ids 0, 1 e 2), ter-se-á a seguinte apresentação de incremento da variável global:

- ***thread 0: global = 1;***
- ***thread 1: global = 2;***
- ***thread 2: global = 3;***
- ***thread 0: global = 4;***
- ***thread 1: global = 5;***
- ..."

0,5

Pois bem, ***SO Kid*** codificou em linguagem C (plataforma Linux) mas, apesar de compilar corretamente, a execução do programa dele não apresenta o resultado esperado. **Utilizando-se do fragmento principal do código dele (abaixo), identifique o(s) problema(s) e apresente a(s) respectiva(s) correção(ões), <u>mas sem remover ou acrescentar novas estruturas de dados, mutexes e semáforos</u>. APRESENTE A(S) CORREÇAO(ÕES), INCLUINDO UMA BREVE EXPLANAÇÃO, NO PRÓPRIO CÓDIGO ABAIXO!**

```
...
void *mythread(void *data);

#define N 3 // number of threads
#define MAX 10
// vetor de semáforos (uma entrada por thread)
// utilizado para controlar o rodízio/rodada
sem_t turn[N];

int global = 0;

int main(void) {
  pthread_t tids[N];
  int i=0;

  // inicializa vetor de semáforos
  for(i=0; i<N; i++) {
     sem_init(&turn[i], 0, 0);
  }

  for(i=0; i<N; i++) {
    int *j = malloc(sizeof(int));
    *j = i;
   pthread_create(&tids[i], NULL, mythread, (void *)j);
  }

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

  return(1);

}
```

```
void *mythread(void *data) {

int id;

  id = *((int *) data);

  while(global < MAX) {

    sem_wait(&turn[id]);
    global++;
    printf("\n thread %d: global = %d", id, global);
    sem_post(&turn[(id+1)%N]);
    sleep(2);
  }

  pthread_exit(NULL);

}
```

7. **[1,5 ponto]** Semelhante ao exercício anterior, agora há um conjunto de *threads* tentando manipular uma variável global mas sem nenhuma ordem preestabelecida. No entanto, **apenas 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 explicações, 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);
}
```

1,5

→ pthread_mutex_lock(&count_mutex);

Logo após incrementada, a variável global perde ser liberada para os próximos threads que a concorrem.