Universidade Federal da Fronteira Sul – Campus Chapecó/SC
Curso de Ciência da Computação - Sistemas Operacionais – 2021.2
Prof. Marco Aurélio Spohn - Avaliação de Reposição 1 (RP1) – 14/03/2022

9,0

Nome: Alex Sandro Zampieri
Obs.:

- **MANTER DESLIGADOS E GUARDADOS CELULARES E COMPUTADORES.**
- **A COMPREENSÃO DAS QUESTÕES FAZ PARTE DA AVALIAÇÃO!!!**
- **RESPONDA À CANETA NOS ESPAÇOS RESERVADOS**

1. [1,5 ponto] *SO Kid*, autodenomina-se o maior especialista em sistemas operacionais. ***SO Kid*** afirma que, caso a CPU suporte isolamento de memória via MMU (unidade de gerenciamento de memória), não se torna necessário no mínimo dois modos de operação da CPU (*i.e.*, modo *kernel* e modo usuário) para suportar um sistema protegido/seguro. **SO Kid está correto? <u>Explique</u>.**

1,0

Correto! Se a memória está devidamente isolada, não existe risco de usuário(pessoa) em modo avançado (modo kernel) comprometer o sistema.

2. [1,5 ponto] Assuma que existem somente **4** processos executando em uma determinada máquina e que cada um desses processos pode necessitar **até 3** unidades de fita simultaneamente. ***SO Kid*** afirma que **9** unidades de fita são suficientes para garantir que o sistema fique totalmente livre de impasses (*deadlocks*), considerando o cenário apresentado (deve-se garantir que todos os processos finalizem, independentemente dos intervalos de tempo de espera pelos recursos solicitados). ***SO Kid* está correto? Explique.**

1,0

Incorreto! De fato, se em um caso específico os processos que total requerem 12 fitas no total, está causada aí o deadlock.

3. [2,0 pontos] *SO Kid* implementou o programa *multithread* abaixo no Linux (apenas o fragmento do código necessário à resolução da questão é apresentado) e disse que nesse programa as *threads* estão livres de *deadlock*? *SO Kid* está correto? **Justifique a sua resposta.**

1,0

```
pthread_mutex_t
mutex1=PTHREAD_MUTEX_INITIALIZER;

pthread_mutex_t
mutex2=PTHREAD_MUTEX_INITIALIZER;

// thread #1
void thread1( ){

againT1:
      if (!pthread_mutex_lock(&mutex1)) {
          if(!pthread_mutex_lock(&mutex2)) {
              ...
              pthread_mutex_unlock(&mutex2);
              pthread_mutex_unlock(&mutex1);
          }
          else {
              pthread_mutex_unlock(&mutex1);
              goto againT1;
          }
      } else goto againT1;
}
```

```
//(continuação...)
    ...

// thread #2
void thread2( ){

againT2:
      if (!pthread_mutex_lock(&mutex2)) {
          if(!pthread_mutex_lock(&mutex1)){
              ...
              pthread_mutex_unlock(&mutex1);
              pthread_mutex_unlock(&mutex2);
          }
          else {
              pthread_mutex_unlock(&mutex2);
              goto againT2;
          }
      } else goto againT2;
}
```

??

Incorreto! Apesar das condições de entrada do if estarem invertidas (pega o mutex se está indisponível), o efeito será o mesma, com ambos threads disputando um mutex sem liberar o anterior e causando deadlock.

4. *[2,0 pontos]* 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 a correção 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);
    take_forks(id);
    sleep(2);
    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);
  state[id]= HUNGRY;
  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;
  }
}
```

1,5

5. [1,5 ponto] Para cada um dos seguintes endereços binários lógicos (*i.e.*, virtuais), calcule o número da página virtual e o deslocamento assumindo páginas de **1 KB (i.e., 1024 bytes)**. **Apresente todo o cálculo em decimal.**

(a) **0101 1110 0010 1011**

page = 16 + 4 + 2 + 1 = 23

offset = 512 + 32 + 8 + 2 + 1 = 555

(b) **1001 0000 1000 1110**

page = 32 + 4 = 36

offset = 128 + 8 + 4 + 2 = 142

6. [1,5 ponto] Um sistema tem quatro processos e cinco recursos alocáveis. A alocação atual e as necessidades máximas são as seguintes:

0

Processo	Alocado	Máximo
A	1, 0, 2, 1, 1	1, 1, 2, 1, 2
B	2, 0, 1, 1, 0	2, 1, 2, 2, 0
C	1, 1, 0, 1, 0	2, 1, 5, 3, 0
D	1, 1, 1, 1, 0	1, 1, 2, 2, 1

Recursos disponíveis: 0, 0, X, 1, 1

Qual o **menor valor de X** para que esse estado seja considerado seguro? Para o valor informado, apresente a sequência de execução até que todos os processos finalizem sua execução (apresentando os recursos disponíveis a cada iteração).

X = 1 ??