## Sistemas Operativos, Pauta Certamen #2, 02/2019 Santiago, 10.01.2020

- **1.** [24% ~ 6% c/u] Conteste brevemente las siguientes preguntas:
- a) ¿Qué es la afinidad del procesador en multiprocesadores? ¿Qué se logra?

  Respuesta: Significa que, si una hebra es itinerada en un procesador, vuelve al mismo procesador una que es se vuelve a itinerar. Se logra aprovechar la utilización de la memoria caché permitiendo una mejora de desempeño.
- **b)** ¿Existen ventajas al disponer diferentes quantums de tiempos en diferentes niveles en una implementación de MFQ?
  - **Respuesta**: Tareas que requieran más uso de CPU que E/S gastaran sus ciclos en colas de tiempos grandes o que usen FCFS, produciendo menos cambios de contexto y mejor uso de recursos. Tareas que requieran mayor uso de E/S se verán favorecidas en colas con tiempos pequeños.
- c) Considere un proceso de varias hebras que se ejecuta en un procesador de un núcleo. ¿Pueden ocurrir condiciones de carrera? Fundamente.
  - Respuesta: Si. Las condiciones de carrera se producen en los cambios de contexto.
- **d)** Describa brevemente las técnicas utilizadas por el SO para intentar mantener todas las CPU's ocupadas.

Respuesta: Los métodos son:

- Push: Frecuentemente buscar procesadores sobrecargados y migrar tareas a otros que puedan estar libres o menos cargados.
- Pull: Procesadores disponibles pueden solicitar tareas a los ocupados.
- **2.** [25%] **Threads**: Usando la API simple thread y pseudocódigo construya un programa que utilizando 4 hebras sume en paralelo los elementos de una matriz entera de 2000 x 4. La idea es que cada hebra sume una columna. Algunas de las funciones útiles:
  - void sthread create(thread, func, arg)
  - int sthread join(thread)
  - void sthread\_exit(ret)

Respuesta: Una posible implementación:

```
#include <stdio.h>
#include "sthread.h"

int conts fila=2000, columna=4;
static int matriz[fila][columna];
int conts hebras= columna;
static int workers[hebras];
static int count=0;
static void suma (int i);
```

```
int main (int argc, char**argv) {
     int a;
     for (a=0; a<hebras; a++){
              sthread_create(&(workers[a]), &suma, a);
     for (a=0; a<hebras; a++){
              count = count + sthread_join(workers[a]);
     }
     printf("%d", count);
     return 0;
}
void suma (int i) {
     int x, j = 0;
     for (x=0; x<fila;x++) {
             j = j + matriz[x][i];
     sthread_exit(j);
}
```

3. [26%] Synchronization: Un conjunto de hebras necesita utilizar dinámicamente bloques de memoria. Para lograrlo disponen de X bloques, todos del mismo tamaño y pueden estar libres u ocupados. Para solucionar en forma segura la gestión de estos bloques se construye la clase MEM que contiene un stack que permite registrar los bloques libres almacenando sus direcciones y las hebras obtienen y devuelven bloques usando las funciones públicas take() y takeBack() respectivamente. Utilizando el mínimo número de variables e instrucciones se solicita implementar la clase y los métodos correspondientes. Guíese por el siguiente código:

## Respuesta:

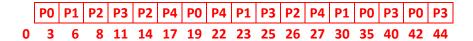
```
const int X=20;
                                                      MEM::MEM() {
class MEM {
                                                      top = X; //stack inicializado con X bloques
private:
        int top;
                                                     int MEM::take(){
        int stack[X];
                                                              int a;
        Lock lock;
                                                             lock.acquire();
                                                             while (top == 0){
        CV cv;
                                                               cv.wait(&lock);}
public:
                                                              a= stack[--top];
        int take();
                                                              lock.release();
        int takeBack(int dir);
                                                              return a;
        MEM();
                                                     void takeBack(int d){
                                                             lock.acquire();
                                                              stack[top++]=d;
                                                              cv.signal();
                                                              lock.release();
```

**4.** [25%] **Planificación de uso de CPU**: Una MFQ de 3 colas (Q1, Q2 y Q3, priorizadas en este orden) utiliza los algoritmos de itineración RR=3, RR=5 y FCFS respectivamente. Considere la siguiente carga de trabajo:

Process	CPU, I/O, CPU	Arrival Time
P0	5,6,7	0
P1	4,2,3	3
P2	2,3,4	4
P3	5,2,7	7
P4	3,2,4	14

Si inicialmente los procesos llegan a Q1 se pide construir la carta Gantt de planificación y calcular los tiempos de espera, respuesta y ejecución promedio. Nota: Si un proceso hace I/O al volver se mantiene en la misma cola donde estaba.

Respuesta: Una posible solución:



	T_ej	T_res	T_esp	E/S	CPU
P0	42	0	24	6	12
P1	27	0	18	2	7
P2	22	2	11	3	6
Р3	37	1	22	2	12
P4	13	0	4	2	7
	28,2	0,6	15,8		