CC4302 Sistemas Operativos
Control 1 – Semestre Primavera 2012
Prof.: Luis Mateu

Pregunta 1

```
typedef struct node {
  int x;
  struct node *next;
} Node;
```

```
typedef struct {
  Node *head;
} Set;
```

```
int buscar(Set *set, int y) {
  Node *pnode= set->head;
  while (pnode!=NULL
         && pnode->x!=y)
    pnode= pnode->next;
  return pnode!=NULL;
}

void insertar(Set *set, int y){
  Node *ins= (Node*)nMalloc(
                 sizeof(Node));
  ins->x= y;
  ins->next= set->head;
  set->head= ins;
}
```

```
int borrar(Set *set, int y) {
  Node **ppnode= &set->head;
  while (*ppnode!=NULL &&
         (*ppnode)->x!=y)
    ppnode= &(*ppnode)->next;
  if (*ppnode==NULL)
    return FALSE;
  else {
    Node *del= *ppnode;
    *ppnode= del->next;
    nFree(del);
    return TRUE;
  }
}
```

El programa anterior es la implementación secuencial de un conjunto en base a una lista simplemente enlazada. Suponga que múltiples threads realizan operaciones sobre un mismo conjunto.

(a) Haga un diagrama de threads que muestre que cuando 2 threads insertan elementos concurrentemente, uno de los elementos podría perderse. *(1.5 puntos)*

(b) Haga un diagrama de threads que muestre que si *buscar* se invoca concurrentemente con *borrar*, se podría encontrar exitosamente un número que jamás fue insertado en el conjunto. Piense en qué pasa si justo después de eliminar un nodo, su memoria se reutiliza inmediatamente en otro nMalloc. *(1.5 puntos)*

(c) Reprograme las estructuras *Node* y *Set* y el procedimiento *borrar* de tal forma que se admitan eliminaciones en paralelo. Es decir, se debe permitir que 2 threads borren distintos nodos de la lista en paralelo. Ud. debe conservar la lista simplemente enlazada como estructura de datos para implementar el conjunto. *No se le pide* reprogramar *buscar* o *insertar*. *(3 puntos)*

Importante: Implementar la exclusión mutua a nivel de la lista enlazada completa tiene 0 puntos porque esto no borraría en paralelo.

Pregunta 2

La siguiente es una implementación secuencial del juego de la vida.

```
int **mat, **newMat; /* se inicializan en el nMain */
int step;

void lifeGame(int n, int k) {
  int i, j;
  for (step= 0; step<k; step++) { /* ciclo externo */
    for (i= 0; i<n; i++)
      for (j= 0; j<n; j++)
        newMat[i][j]= compute(mat, i, j); /* dado */
    swap(&mat, &newMat); /* dado */
  }
}
```

El programa parte con una Matriz M (mat en el programa) y calcula las matrices M_0, M_1, M_2, ..., M_{k-1}. Cada matriz es de n x n. El valor de cada elemento de la matriz M_{step} depende únicamente de M_{step-1}. Este hecho simplifica la paralelización del procedimiento. El procedimiento `compute` es dado y su implementación no es relevante en esta pregunta.

Paralelice `lifeGame` haciendo uso de p threads adicionales. Para ello haga que cada thread construya n/p filas (o columnas) de la nueva matriz. Suponga que n es múltiplo de p. Observe que en cada instante todos los threads deben trabajar en la construcción de la misma matriz M_{step}, es decir el mismo valor para `step`, porque de otro modo su programa podría no entregar el mismo resultado que la versión secuencial. Esto significa que al final de cada iteración de `step`, se debe esperar a que todos los threads completen la construcción de su respectiva submatriz, y luego comenzar una nueva iteración.

Restricciones:

- Ud. debe crear solo p threads adicionales (el cliente no acepta el sobrecosto adicional de crear más de p threads).
- Resuelva el problema utilizando nSystem y los *monitores* de nSystem.
- Todos los threads creados deben tener su respectivo nWaitTask.

Observación: no pueden crearse p threads para cada iteración de step. Esto violaría la primera restricción. El número total de invocaciones de nEmitTask no puede exceder p. Esto significa que se usa el mismo thread para calcular las k versiones de una parte de la matriz. Un thread calcula su parte de la matriz, luego se sincroniza esperando que todos los demás threads terminen su parte, y recién entonces recomienza el cálculo de una nueva iteración.