CC4302 Sistemas Operativos – Control 1 – Semestre Otoño 2015 – Prof.: Luis Mateu

Pregunta 1

Se ha especificado un sistema de mensajes alternativo al de nSystem. El sistema se diferencia en que los mensajes se envían indirectamente a través de un canal y no a un proceso. El receptor indica por qué canal espera recibir el mensaje. Por simplicidad un mensaje es siempre un número entero. Los mensajes no necesitan ser respondidos pero *el emisor debe esperar hasta que su mensaje sea recibido*. En el siguiente ejemplo, la tarea receptora debe desplegar 0 1 2 3 4:

```
Canal *c= nuevoCanal(); /* compartido por ambas tareas */
```

Emisor	Receptor

`int i;`
`for (i= 0; i<5; i++)`
`  enviar(c, i);`
	`int j;`
	`for (j= 0; j<5; j++)`
	`  nPrintf("%d ", recibir(c));`

La siguiente es una implementación incorrecta de este sistema de mensajes:

```
typedef struct {
  nSem envSem, recSem;
  int msg;
} Canal;

Canal *nuevoCanal() {
  Canal *c= nMalloc(
          sizeof(Canal));
  c->envSem= nMakeSem(0);
  c->recSem= nMakeSem(0);
  return c;
}
```

```
void enviar(Canal *c, int msg){
  c->msg= msg;
  nSignalSem(c->envSem);
  nWaitSem(c->recSem);
}

int recibir(Canal *c) {
  nSignalSem(c->recSem);
  nWaitSem(c->envSem);
  return c->msg;
}
```

(a) Haga un *diagrama de threads* que muestre que esta implementación es incorrecta cuando varias tareas envían mensajes a un mismo canal (1 punto).

(b) Haga un diagrama de threads que muestre que esta implementación podría no desplegar 0 1 2 3 4 con el código de ejemplo de más arriba (1,5 puntos).

(c) Corrija esta implementación de modo que funcione correctamente para el caso de varias tareas que envían y varias que reciben por el mismo canal. La solución debe basarse en semáforos únicamente. No olvide que el emisor debe esperar hasta que su mensaje sea recibido (2 puntos).

(d) Confeccione la siguiente tabla: en las columnas coloque las estrategias de *scheduling first come first served*, *shortest job first* (*preemptive*) y *round robin*. En las filas coloque los atributos tiempo de despacho, tiempo de respuesta y número de cambios de contexto innecesarios. En la celdas de la tabla indique si esa estrategia es peor, mejor o promedio para ese atributo al compararla con las otras estrategias (1,5 puntos).

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.