CC41B Sistemas Operativos
Control 1 – Semestre Otoño 2010
Prof.: Luis Mateu

Pregunta 1

Un thread ejecuta la función `T1` y otro thread ejecuta paralelamente `T2`.

```
int T1() {
  int v1= prologo1();
  int v2= reunion1(v1)
  epilogo1(v1, v2);
}
```

```
int T2() {
  int v2= prologo2();
  int v1= reunion2(v2)
  epilogo2(v1, v2);
}
```

En un punto de la ejecución (y una sola vez) deben intercambiar los valores que calcularon en sus prólogos. Este intercambio se realiza invocando las funciones `reunion1` y `reunion2`. Se tiene la siguiente implementación de estas 2 funciones:

```
volatile int flag1= FALSE, flag2= FALSE;
int val1, val2;
```

```
int reunion1(int v) {
  flag1= TRUE;
  val1= v;
  while (!flag2)
    ;
  return val2;
}
```

```
int reunion2(int v) {
  flag2= TRUE;
  val2= v;
  while (!flag1)
    ;
  return val1;
}
```

(a) Discuta si esta solución es correcta (sin importar la eficiencia). Si es incorrecta haga un diagrama de threads que muestre una situación en donde el resultado es incorrecto. Además discuta si es posible hacer un pequeño cambio para que la solución sí sea correcta.

(b) Escriba una solución correcta y *eficiente* de `reunion1` y `reunion2` usando 2 semáforos de nSystem.

Pregunta 2

Un grupo de programadores alternan su jornada diaria programando y comiendo pizza. Los hambrientos van a la pizzería, piden su pizza y esperan en la tienda hasta recibir su pizza. Los hackers son más eficientes porque ordenan por teléfono la pizza, luego programan y más tarde van a retirar su pizza. Estas actividades se ven reflejadas en estos procedimientos:

```
int hambriento() {
  for(;;) {
    Pizza* pizza=
              comprarPizza();
    comer(pizza);
    programar();
  }
}
```

```
int hacker() {
  for(;;) {
    Pizza* pizza;
    Orden* orden= ordenarPizza();
    programar1();
    pizza= esperarPizza(orden);
    comer(pizza);
    programar2();
  }
}
```

La implementación actual de la pizzería es ineficiente porque hornea una sola pizza a la vez (a pesar de que el horno permite hornear 4 pizzas simultáneamente) y las pizzas de los hackers solo comienzan a prepararse una vez que el cliente llega a la tienda:

```
typedef void Orden;
Horno* horno;
nSem sem;
Pizza *obtenerPizzaCruda();
void hornear(Horno* horno, Pizza* pizza_vec, int n_pizzas);
```

```
void iniciarPizzeria() {
  sem= nMakeSem(1);
}
Orden* ordenarPizza() {
  return NULL;
}
Pizza* esperarPizza(
           Orden* orden) {
  return comprarPizza();
}
```

```
Pizza* comprarPizza() {
  Pizza *pizza=
      obtenerPizzaCruda();
  Pizza *pizza_vec[1];
  pizza_vec[0]= pizza;
  nWaitSem(sem);
  hornear(horno, pizza_vec, 1);
  nSignalSem(sem);
  return pizza;
}
```

Usando los *monitores* de nSystem, haga un implementación *eficiente* de las funciones `iniciarPizzeria`, `ordenarPizza`, `esperarPizza` y `comprarPizza`. Especifique también la estructura de datos `Orden`. Los demás procedimientos son dados. Ud. necesitará crear un thread adicional para operar el horno. Considere que `hornear` es la única operación que toma mucho tiempo en ejecutarse.

Restricciones (ordenadas de mayor a menor importancia):

i. La invocación de `hornear` debe ocurrir en exclusión mutua.

ii. Invoque `hornear` para cocinar 4 pizzas suministradas en el arreglo `pizza_vec`. Si no han llegado suficientes clientes, puede hornear menos de 4 pizzas, pero hornee al menos una pizza. El horno no se abre hasta que `hornear` termina.

iii. Su solución no debe producir hambruna.

iv. Si el horno está disponible, comience a hornear las pizzas de los hackers antes de la invocación de `esperarPizza`, pero después del retorno de `ordenarPizza`.