CC4302 – Sistemas Operativos
Auxiliar 2

Profesor: Luis Mateu
Auxiliar: Diego Madariaga

25 de marzo de 2020

1 P1 Control 1, 2007

```
typedef struct Nodo {
    char *llave, *def;
    struct Node *izq, *der;
    nSem semIzq, semDer;
} Nodo;

typedef struct {
    Nodo *raiz;
    nSem semRaiz;
} ConcDict;

void addDef(ConcDict *dict,
            char *llave, char *def) {
    insertar(dict->semRaiz,
             &dict->raiz,
             llave, def);
}
```

```
void insertar(nSem sem, Nodo **ppnodo,
              char *llave; char *def) {
    Nodo *pnodo;
    if (*ppnodo == NULL) {
        *ppnodo =
            crearNodoHoja(llave,
                          def); /* dado */
    } else {
        /*** Supuesto: la llave ***
        *** no está en el árbol ***/
        pnodo = *ppnodo;
        if (strcmp(llave,
                   pnodo->llave) < 0) {
            insertar(pnodo->semIzq,
                     &pnodo->izq,
                     llave, def);
        } else {
            insertar(pnodo->semDer,
                     &pnodo->der,
                     llave, def);
} } }
```

El programa anterior es la implementación incompleta de un diccionario concurrente en base a un árbol de búsqueda binaria. El procedimiento `crearNodoHoja` es dado e inicializa correctamente todos los campos. Esto incluye la creación de los semáforos que contiene el nodo, cada uno con un ticket. Cuando se crea el diccionario, se crea automáticamente el semáforo para la raíz en la estructura `ConcDict`, aún cuando el diccionario esté vacío. Suponga que no existe una operación para eliminar definiciones en el diccionario.

1. Haga un diagrama de threads que muestre que el diccionario puede quedar en un estado inconsistente cuando 2 threads llaman a `addDef` concurrentemente.

2. Complete el programa de más arriba agregando la líneas que faltan. Ud. necesita garantizar la exclusión mutua cuando 2 threads están trabajando en el mismo nodo, pero Ud. *debe* permitir que continúen en paralelo una vez que trabajan en ramas distintas del árbol. El código dado es correcto, sólo agregue las líneas que faltan para la sincronización. La declaración de los semáforos y su paso como parámetros es una ayuda para que resuelva el problema.

   Importante: Implementar la exclusión mutua a nivel del árbol completo tiene 0 puntos.

2 P2 Control 1, 2010/1

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();}
```

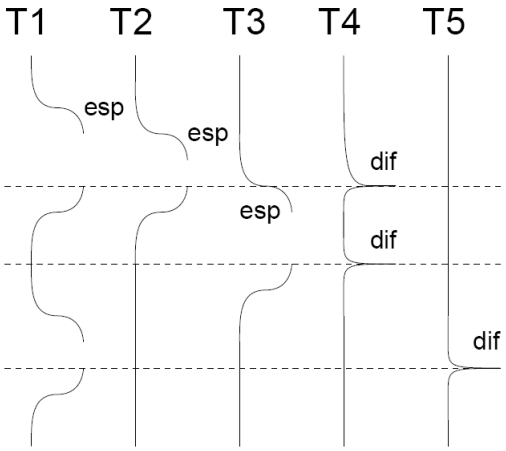
```
Pizza* comprarPizza() {
  Pizza *pizza= obtenerPizzaCruda();
  Pizza *pizza_vec[1];
  pizza_vec[0]= pizza;
  nWaitSem(sem);
  hornear(horno, pizza_vec, 1);
  nSignalSem(sem);
  return pizza;
}
```

Usando los monitores de nSystem, implemente eficiente las siguientes 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):

1. La invocación de hornear debe ocurrir en exclusión mutua.
2. 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.
3. Su solución no debe producir hambruna.
4. 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`.

3 P1 Control 1, 2008

Para los procedimientos `esperar` y `difundir` se especifica que cuando cualquier thread invoca `esperar`, ese thread debe quedar bloqueado hasta que algún thread invoque `difundir`, retornando la información suministrada por `difundir`. Si la invocación de `esperar` ocurre simultáneamente con la invocación de `difundir` (es decir, si hay algún traslape en las dos invocaciones), el thread puede continuar de inmediato o bien puede bloquearse hasta la siguiente invocación de `difundir`. Si ocurren 2 invocaciones simultáneas de `difundir`, `esperar` puede retornar la información suministrada en cualquiera de esas 2 invocaciones de `difundir`.



Se propone la siguiente implementación:

```
int cont = 0;
Info *info;

void difundir(Info *infoP) {
    cont++;
    info = infoP;
}
```

```
Info *esperar() {
    int micont = cont;
    while (micont == cont)
        ;
    return info;
}
```

1. Haga un diagrama de threads que muestre que la solución propuesta es inconsistente. Por ejemplo un thread podría retornar información equivocada. (Considere que nunca se produce el desborde de la variable `cont`.)

2. Indique si es posible corregir esta solución haciendo una pequeña modificación de forma tal que ya no se produzcan inconsistencias.

3. Critique esta solución desde el punto de vista de la eficiencia.

4. Escriba una solución consistente y *eficiente* de `esperar` y `difundir` usando como herramienta de sincronización los semáforos de nSystem. Considere que estos semáforos garantizan que los `nWaitSem` serán atendidos en orden FIFO.