CC4302 – Sistemas Operativos
Auxiliar N°2

Profesor: Luis Mateu
Auxiliar: Juan Manuel Barrios

3 de Septiembre de 2010

1. Control 1, 2007, pregunta 1

```
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. Control 1, Otoño 2010, 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):

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. Control 1, 2002, pregunta 2

La municipalidad de Geek Island posee una barcaza para el transbordo de vehículos desde y hacia el continente. La barcaza tiene capacidad para llevar un solo vehículo. Los siguientes procedimientos describen la rutina de los isleños que viven en la isla y los turistas que la visitan:

```
int isleno() {
    for(;;) {
        cruzarAContinente();
        trabajar();
        cruzarAIsla();
        dormir();
    }
}
```

```
int turista() {
    for (;;) {
        cruzarAIsla();
        turistear();
        cruzarAContinente();
        dormir();
    }
}
```

En donde `trabajar`, `turistear` y `dormir` son procedimientos dados. Para coordinar el uso adecuado de la barcaza, actualmente se usa la siguiente implementación de `cruzaraAContinente` y `cruzarAIsla`:

```
nSem semBarcaza; /* =nMakeSem(1) en nMain */
int enContinente = TRUE;

void cruzarAIsla() {
    nWaitSem(semBarcaza);
    if (!enContinente)
        navegarAContinente();
    navegarAIsla();
    enContinente=FALSE;
    nSignalSem(semBarcaza);
}
```

```
void cruzarAContinente() {
    nWaitSem(semBarcaza);
    if (enContinente)
        navegarAIsla();
    navegarAContinente();
    enContinente=TRUE;
    nSignalSem(semBarcaza);
}
```

En donde `navegarAContinete` y `navegarAIsla` son procedimientos dados y se demoran un tiempo considerable. Esta implementación no es eficiente porque puede ocurrir que habiendo un vehículo en el continente, la barcaza navegue vacía a la isla (y viceversa). No es de extrañar entonces que se formen filas de vehículos en espera de la barcaza en ambas orillas. Utilizando *monitores* de nSystem, escriba una nueva implementación de `cruzarAIsla` y `cruzarAContinente` que permita aprovechar eficientemente la barcaza. Observe que:

- La barcaza no puede navegar dos veces seguidas en el mismo sentido.
- Un usuario que llama a `cruzarA...` no puede continuar mientras que su vehículo no haya sido transportado y la barcaza no haya llegado la otra orilla.
- La barcaza no puede transportar un vehículo si éste no llegó a la orilla de partida (momento en que se invoca a `cruzarA...`).
- En cada viaje, si existen vehículos en espera en la orilla de la partida, la barcaza debe transportar un y sólo un vehículo, y ese vehículo tiene que ser el que llegó primero a la orilla.
- Ud. no debe detener la barcaza si hay vehículos esperando en alguna de las orillas.

- Ud. sí debe detener la barcaza si no hay ningún vehículo que transportar en ninguna de las dos orillas.

4. Solución

4.1. P1

1. Más de un thread puede cumplir con la condición `*ppnodo==NULL` y cada uno invocará el método `crearNodoHoja` pero finalmente sólo uno va quedar asignado a `*ppnodo`.

2. Usar el semáforo `sem` para que la pregunta `*ppnodo==NULL` y luego la asignación sobre `*ppnodo` no pueda ser interrumpida por otro thread.

```
void insertar(nSem sem, Nodo **ppnodo, char *llave; char *def) {
    Nodo *pnodo;
    nWaitSem(sem);
    if (*ppnodo==NULL) {
        *ppnodo= crearNodoHoja(llave, def);
        nSignalSem(sem);
    } else {
        nSignalSem(sem);
        pnodo= *ppnodo;
        if (strcmp(llave, pnodo->llave) < 0) {
            insertar(pnodo->semIzq, &pnodo->izq, llave, def);
        } else {
            insertar(pnodo->semDer, &pnodo->der, llave, def);
        }
    }
}
```

4.2. P2

```
typedef struct Orden {
    Pizza *pizza;
    int cocida;
} Orden;

nMonitor ctrl;
nTask pizzeriaTask;
FifoQueue *pendientes;

Orden* ordenarPizza() {
    Orden *ord =
         (Orden*) nMalloc(sizeof(Orden));
    ord->pizza = obtenerPizzaCruda();
    ord->cocida = FALSE;
    nEnter(ctrl);
    PutObj(pendientes, ord);
    nNotifyAll(ctrl);
    nExit(ctrl);
    return ord;
}
```

```
void iniciarPizzeria() {
    ctrl = nMakeMonitor();
    pizzeriaTask = nEmitTask(pizzeriaProc);
    hambrientos = MakeFifoQueue();
    hackers = MakeFifoQueue();
}

Pizza* esperarPizza(Orden *po) {
    nEnter(ctrl);
    while (!po->cocida)
        nWait(ctrl);
    nExit(ctrl);
    Pizza* pizza = po->pizza;
    nFree(po);
    return pizza;
}

Pizza* comprarPizza() {
    return esperarPizza(ordenarPizza());
}
```

```
int pizzeriaProc() {
    int cont, i, P = 4;
    Pizza *pizza_vec[P];
    Orden *orden_vec[P];
    for (;;) {
        cont = 0;
        nEnter(ctrl);
        while (EmptyFifoQueue(pendientes))
            nWait(ctrl);
        while (!EmptyFifoQueue(pendientes) && cont < P) {
            orden_vec[cont] = (Orden*) GetObj(pendientes);
            pizza_vec[cont] = orden_vec[cont]->pizza;
            cont++;
        }
        nExit(ctrl);
        hornear(horno, pizza_vec, cont);
        nEnter(ctrl);
        for (i = 0; i < cont; i++)
            orden_vec[i]->cocida = TRUE;
        nNotifyAll(ctrl);
        nExit(ctrl);
    }
    return 0;
}
```

4.3. P3

```
int ticket_isla=0, ticket_cont=0;
int visor_isla=0, visor_cont=0;
nMonitor mon; /* = nMakeMonitor() en nMain */

void cruzarAIsla() {
    nEnter(mon);
    int mi_ticket = ++ticket_cont;
    nNotifyAll(mon);
    while(mi_ticket > visor_cont)
        nWait(mon);
    nExit(mon);
}

void cruzarAContinente() {
    nEnter(mon);
    int mi_ticket = ++ticket_isla;
    nNotifyAll(mon);
    while(mi_ticket > visor_isla)
        nWait(mon);
    nExit(mon);
}

int barcaza(){
    int enContinente = TRUE;
    int atiendo_cont, atiendo_isla;
    for(;;){
        nEnter(mon);
        while(ticket_isla <= visor_isla
           && ticket_cont <= visor_cont)
            nWait(mon);
        atiendo_cont = atiendo_isla = FALSE;
        if(enContinente && ticket_cont > visor_cont)
            atiendo_cont = TRUE;
        else if(!enContinente && ticket_isla > visor_isla)
            atiendo_isla = TRUE;
        nExit(mon);
        if(enContinente)
            navegarAIsla();
        else
            navegarAContinente();
        enContinente = !enContinente;
        if(atiendo_cont){
            nEnter(mon);
            visor_cont++;
            nNotifyAll(mon);
            nExit(mon);
        } else if(atiendo_isla){
            nEnter(mon);
            visor_isla++;
            nNotifyAll(mon);
            nExit(mon);
        }
    }
}
```