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

Pregunta 1

Un computador posee dos discos de igual capacidad organizados en espejo. Esto significa que se escribe en paralelo la misma información en ambos discos: *un disco es un espejo del otro*. Al leer, basta leer solo uno de los 2 discos y cualquiera de ellos. La principal ventaja de esta organización es que si uno de los discos falla, no se pierde información. Además, se pueden satisfacer simultáneamente 2 lecturas diferentes, provenientes de 2 procesos, leyendo en paralelo ambos discos.

La siguiente es un implementación *incorrecta* de la lectura y escritura de un bloque en el disco.

```
int busy[2]= {FALSE, FALSE};
Disk *disks[2];
/* inicializados al partir el SO */

void mirrorWrite(
     char *buf, int block) {
  while(busy[0] || busy[1])
    /* wait */ ;
  busy[0]= busy[1]= TRUE;
  diskStartWrite(disks[0],
           buf, block);
  diskStartWrite(disks[1],
           buf, block);
  diskWait(disk[0]);
  diskWait(disk[1]);
  busy[0]= busy[1]= FALSE;
}
```

```
void mirrorRead(
     char *buf, int block) {
  int i= -1;
  while (i<0) {
    if (!busy[0]) i= 0;
    else if (!busy[1]) i= 1;
  }

  busy[i]= TRUE;
  diskStartRead(disks[i],
           buf, block);
  diskWait(disk[i]);
  busy[i]= FALSE;
}
```

Programe una solución correcta de ambos procedimientos usando los *monitores de nSystem*. Su solución debe ser capaz de satisfacer 2 requerimientos de lectura en paralelo y además debe evitar la *hambruna*.

Pregunta 2

En la columna de al lado se muestra la implementación secuencial de un conjunto en base a una lista simplemente enlazada.

(i) Haga un diagrama de threads que muestre cómo una invocación de `belongs` puede fallar con un *segmentation fault* si se realiza concurrentemente con una invocación de `insert`. Considere solo estos 2 threads. No olvide considerar las optimizaciones que puede realizar el compilador. Explique.

(ii) Haga un diagrama de threads que muestre cómo una invocación de `belongs` con una palabra que nunca fue insertada en el conjunto puede retornar verdadero si se invoca concurrentemente con una invocación de `remove`. Considere solo estos 2 threads.

```
typedef struct Node {
  char *elem;
  struct Node *next;
} Node;

Node head= {"", NULL};

int belongs(char *elem) {
  Node *pnode= head.next;
  while (pnode!=NULL) {
    if ( strcmp(pnode->elem,
                elem)==0 )
        return TRUE;
    pnode= pnode->next;
  }
  return FALSE;
}
```

```
void insert(char *elem) {
  Node *pnode=
    (Node*)malloc(sizeof(Node));
  pnode->elem= elem;
  pnode->next= head.next;
  head.next= pnode;
}

void remove(char *elem) {
  Node *pnode= &head;
  while (pnode->next!=NULL) {
    if ( strcmp(
           pnode->next->elem,
           elem) == 0 ) {
      Node *prem= pnode->next;
      pnode->next= prem->next;
      free(prem);
      return;
    }
    pnode= pnode->next;
  }
}
```

(iii) Modifique la implementación anterior de modo que pueda ser usada desde múltiples threads concurrentes. Restricciones:

- Ud. debe preservar la estructura de datos, es decir lista simplemente enlazada.
- Dado que `belongs` y `remove` toman tiempo lineal con respecto al tamaño del conjunto su implementación debe permitir ejecuciones paralelas de estas 2 operaciones. Es decir *no debe haber exclusión mutua* total entre una invocación de `remove` con otras invocaciones de `remove` o `belongs`. Sin embargo, para garantizar la correctitud de la solución sí será necesario garantizar la exclusión mutua parcial de la lista enlazada.
- Como primitiva de sincronización Ud. debe usar los semáforos de nSystem. Considere usar un semáforo por nodo de la lista pero piense cuidadosamente para qué datos se garantiza la exclusión mutua con este semáforo.