SISTEMAS OPERATIVOS I

Segundo curso Ingeniería Informática. Curso 2003-2004

Práctica 4: Regiones críticas condicionales. Solucionar el problema de los lectores escritores usando regiones críticas condicionales

Las soluciones para al problema de los lectores escritores se han visto en clase. Puede optarse por la solución con prioridad para los lectores o la que tiene prioridad para los escriores Realizaremos tres programas

- Programa inicializa.c. Crea la región e inicializa las variables que haya en ella; también crea e inicializa el objeto que comparten los lectores y escritores Después de que acaben todos los lectores y escritores elimina la memoria compartida y la región crítica y los recursos ipc que utiliza.
- Programa lector.c. Esta formado por un bucle en el que lee el objeto, imprime su valor y realiza otras cosas, que representaremos por una espera de un tiempo aleatorio

```
loop
   Leer objeto;
   Imprimir objeto;
   Hacer otras cosas (esperar tiempo aleatorio);
endloop;
```

- Programa escritor.c. Esta formado por un bucle en el que escribe el objeto, indica que ha escrito el objeto y realiza otras cosas, que representaremos por una espera de un tiempo aleatorio

```
loop
   Escribir el objeto;
   Avisar de que ha actualizado el objeto;
   Hacer otras cosas (esperar tiempo aleatorio);
end loop;
```

El objeto que se lee y se escribe será un array de 80 caracteres formado por la repetición, separada por el caracter *, de las tres ultimas cifras del pid del proceso que lo escribe . Por ejemplo, si el escritor que tiene el pid 51257 modifica el objeto lo dejará de la siguiente manera

257*257*257*257*257*257*257*257*257*257*257*257*257*257*257*257*257

Para poder ver el efecto de la sincronización deben incluirse retardos en el acceso al objeto por parte de los lectores y lo escritores. La salida por pantalla

será con la llamada al sistema write. (una sola llamada write para toda la salida en cada iteración del bucle). Por ejemplo para un lector la salida (en una iteración del bucle) sería

```
Lector pid 46219, objeto: 257*257*257*257*257*257*257*257*257*257*2..
```

y para un escritor

```
Lector pid 51257, actualizado objeto.
```

La práctica debe probarse con **varios** lectores y **varios** escritores y también sin usar las regiones críticas condicionales para comprobar que en este caso falla (los lectores se encuentran un valor del objeto inconsistente).

Comentarios En el problema de los lectores escritores el objeto al que acceden los lectores y los escritores no debe formar parte de la región crítica, pues de hacerlo el acceso a él sería dentro de una sentencia *region* y por tanto habría exclusión mutua tambien entre los lectores, lo que no es adecuado. Dicho objeto debe por tanto compartirse aparte con las funciones de compartición de memoria vistas en una práctica anterior. Para evitar conflictos con los identificadores de las zonas de memoria compartida (la implementación de las regiones críticas que se suministra las usa) está disponible el fichero *sharedmem-reg.h* en

```
http://www.dc.fi.udc.es/os/~afyanez/Practicas/sources/sharedmem-reg.h
```

que, como puede comprobarse es igual que el *sharedmem.h* usado en dicha práctica anterior, pero utiliza una clave distinta para generar la memoria compartida.

Para usar regiones críticas debe hacerse lo siguiente

I Utilizar el include que se suministra "region.h", disponible en

```
http://www.dc.fi.udc.es/os/~afyanez/Practicas/sources/region.h
```

e incluirlo en el código fuente

```
#include "region.h"
```

II Declarar las variables que se quieren compartir en una estructura (el nombre de la estructura no es significativo)

```
struct COMPARTIR {
     /* lista de variables a compartir */
      ...
};
```

III Declarar un puntero al tipo REGION (definido en "region.h") que es

el que nos va a permitir acceder a la región.

IV Usar las funciones que se suministran en el include region.h.

- **REGION * rcreate (size_t s)**
  Crea una región y devuelve un puntero a la región creada. Recibe como parmetro el tamaño de la estructura que se comparte.
- **int rdestroy (REGION *r)**
  Destruye una región creada con rcreate y elimina los recursos que utilizaba.
- **REGION * rmalloc (size_t s)**
  Devuelve un puntero a una región ya creada.
- **int rfree (REGION *r)**
  Libera la memoria ocupada por una región sin eliminar los recursos ipc.
- **int Region (REGION * r, void (*sentencia)())**
  Ejecuta region r do sentencia. Sentencia es una función que devuelve void y recibe un puntero a la estrutura compartida declarado como (void *).
- **int RegionP (REGION *r, void (*sentencia)(), void * parm)**
  Ejecuta region r do sentencia. Sentencia es una función que devuelve void y recibe un puntero a la estrutura compartida declarado como (void *) y un parámetro adicional tambien declarado como (void *).
- **int RegionCond (REGION * r,int (*cond)(), void (*sentencia)())**
  Ejecuta region r when cond do sentencia. Sentencia es una función que devuelve void recibe un puntero a la estrutura compartida declarado como (void *). cond es una función que devuelve int y recibe como parámetro un puntero a la estructura compartida, tambien declarado como (void *).
- **int RegionCondP (REGION * r,int (*cond)(), void * parmcond, void (*sentencia)(), void * sentparm)**
  Igual que la anterior, pero cond y sentencia reciben un parámetro adicional
- **int RegionCond2 (REGION * r, int (*cond)(), void (*sentencia1)(), void (*sentencia2)())** Forma segunda de la region crítica condicional. Ejecuta

```
region r do
  begin
```

```
    sentencia1;
    esperar (cond);
    sentencia2
  end;
```

- **int RegionCond2P (REGION *r,int(*cond)(),void *parmcond, void (*sentencia1)(), void *sent1parm, void (*sentencia2)(), void *sent2parm)**
  Igual que la anterior pero cond, sentencia1 y sentencia2 reciben parametros adicionales parmcond, sent1parm y sent2parm.

En caso de error las funciones que devuelven un puntero devuelven NULL y las que devuelven entero devuelven -1. El entero regerrno indica que error y reg_errlist[regerrno] nos da una descripcion del error. Con sys_errlist[errno] o con perror podemos obtener más información del error.
Por ejemplo si rcreate devuelve NULL y reg_errlist[regerrno] es "semget error" sabemos que rcreate falló porque no pudo obtener el semáforo. Si ahora vemos que perror indica "File exists", conocemos el motivo por el que semget falló.

Ejemplo de uso: Productor en el problema de los productores-consumidores

Declaración de la variable compartida

```
struct PC {
     TIPOITEM buff [N]; /*tipoitem estaria hecho con typedef*/
     in,out,cont: integer;
};
```

La función añadir al buffer

```
void AniadirBuffer (struct PC *p, TIPOTITEM * item)
{
   p->buff[p->in]=*item;
   p->in = (p->in +1) %N;
   (p->cont)++;
}
```

Esta construcción de AniadirBuffer es posible que produzca *warnings* dependiendo del compilador y de las opciones de compilacion ya que en "region.h" los punteros estan declarados como (void *). Para evitar los *warnings* podríamos hacer

```
void AniadirBuffer (void * p1, void *p2)
{
   struct PC *p;
   TIPOITEM *item;

   p = (struct PC *) p1;
   item = (TIPOITEM *) p2;
   p->buff[p->in]=*item;
   p->in = (p->in +1) %N;
   (p->cont)++;
}
```

Como se ha usado una función que recibe un parámetro adicional (el item), la función condición tambien debe recibir otro parámetro aunque no lo use.

```
int PuedoAniadir (void * p1, void * nouso)
{
   struct PC * p = (struct PC *) p1;
   return (p->cont == N);
}
```

El código del productor (sin el control de errores):

```
Productor ()
{
  REGION *r;
  TIPOITEM item;
  int veces = 30; /* en lugar de un repeat until false */
  r=rmalloc (sizeof (struct PC));
  while (-- veces) {
     producir (&item);
     RegionCondP (r,PuedoAniadir, NULL, AniadirBuffer, &item);
     }
  rfree (r);
}
```

FORMA DE ENTREGA Como en la práctica anterior.

FECHA DE ENTREGA VIERNES 30 ENERO 2004