Universidad Nacional

Sistemas Operativos

Título: Mutex con variables condicionales.

Las funciones para la implementación de *Mutex* con POSIX, tienen la posibilidad de ser usadas con variables condicionales. Estas variables permiten condicionar el funcionamiento de los *mutex*, lo que permite que sean muy adecuadas para ser usadas con problemas como el del productor-consumidor. En este problema, se tienen dos procesos, uno que produce datos y otro que los consume, y se debe establecer un mecanismo para determinar si el espacio de memoria que usan para comunicarse está $vacio$ o si por el contrario se encuentra $lleno$. Así, el proceso consumidor se bloqueará hasta que hayan datos, o el proceso productor se bloqueará hasta que se consuma un dato más.

Los **prototipos** de las funciones para crear y manipular variables condicionales con *mutex* son mostradas a continuación:

```
int pthread_mutex_init(pthread_mutex_t *restrict mutex,
                              const pthread_mutexattr_t *restrict attr);
int pthread_mutex_destroy(pthread_mutex_t *mutex);

int pthread_cond_destroy(pthread_cond_t *cond);
int pthread_cond_init(pthread_cond_t *restrict cond,
                             const pthread_condattr_t *restrict attr);

int pthread_cond_wait(pthread_cond_t *restrict cond,
                             pthread_mutex_t *restrict mutex);
int pthread_cond_signal(pthread_cond_t *cond);
```

Las primeras dos funciones servirán para crear y destruir los *mutex*. Las siguientes dos permiten crear y destruir las variables condicionales, que para el caso del problema del productor-consumidor serán *vacío* y *lleno*. Finalmente, las últimas dos funciones permiten bloquear un proceso o liberarlo, con base a la variable condicional creada y el *mutex* creado.

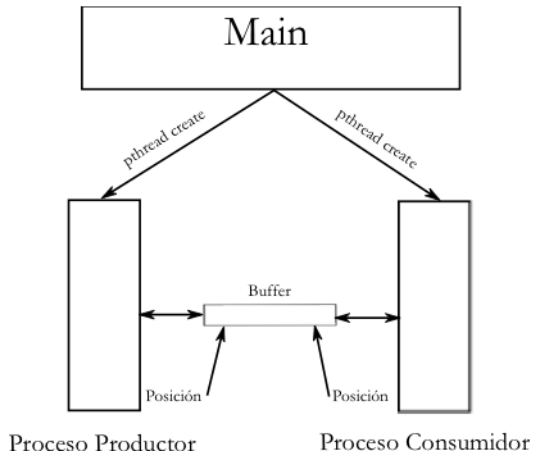

Figura 1: Diagrama de bloques del programa.

Si suponemos el problema del productor consumidor, podemos deducir que habrán dos circunstancias en las que alguno de los dos procesos se bloquea:

- El primer caso corresponde a cuando el espacio de comunicación entre los dos procesos está **vacío**, por lo que el proceso consumidor se deberá bloquear (*pthread_cond_wait*, por que no hay nada que consumir), hasta que el proceso productor ingrese al menos un dato en memoria, en donde realizará un *signal* avisando que ya no está vacía la memoria.

- El segundo caso se observa cuando el espacio de memoria está **lleno**, por lo que el proceso productor se deberá bloquear (*pthread_cond_wait*), hasta que el proceso consumidor lea o consuma al menos un dato, en cuyo caso notificará mediante un *pthread_cond_signal* al otro proceso para desbloquearlo.

Ten en cuenta que para la cada una de las funciones, se pide un puntero de tipo variable condicional (pthread_cond_t *cond) y/o un puntero de tipo *mutex* (pthread_mutex_t *mutex). Así, si **no** declaras las variables como punteros, deberás pasar la dirección de la variable (para que la función crea que le pasaste un puntero) anteponiendo un &.

La figura 1 muestra el diagrama de bloques del programa con procesos productor y consumidor. Se observa que el programa está compuesto por tres funciones (main, productor, consumidor), siendo estas dos últimas ejecutadas mediante dos hilos.

Para comunicar los dos procesos se crea una variable llamada *buffer* con longitud fija. Este espacio de memoria funcionará como una lista circular, lo que implica que se usen punteros o índices (*posición*) en cada hilo para conocer la próxima posición a extraer o almacenar de *buffer*.

En el código a continuación se observa que cada vez que se almacene un dato en buffer, posicion es incrementado a fin de guardar en el siguiente espacio de memoria la próxima vez. De igual forma, cuando se lea un dato, se incrementará a fin de leer la próxima posición de memoria la siguiente vez.

Así mismo, se puede detallar, que hay una variable llamada *num_datos* para llevar la cuenta del número de datos presentes en el *buffer* sin leer. Esta variable será la que se utilice para bloquear un proceso u otro. Lo anterior implica que si *num_datos* es 0, es por que el *buffer* está vacío, y si es MAX_DATOS es por que está lleno. En el primer caso deberá bloquearse el proceso consumidor, y en el segundo deberá bloquearse el proceso productor.

Listing 1: Librerías

```
1  //Poner aqui las librerias requeridas
2
3  #define MAX_BUFFER 16
4  #define MAX_DATOS 1000
5
6  //Variables globales
7  pthread_mutex_t mutex;
8  pthread_cond_t vacio, lleno;
9  int buffer[MAX_BUFFER], num_datos;
```

Listing 2: Proceso consumidor

```
1  // proceso consumidor
2  * consumidor()
3  {
4          struct datos_tipo *datos_proceso;
5          int a, i, j, p, posicion=0, dato;
6
7          for(i=0; i < MAX_DATOS; i++)
8          {
9           pthread_mutex_lock(&mutex);
10          while(num_datos == 0)
11                  pthread_cond_wait(&vacio, &mutex);
12
13          dato = buffer[posicion];
14          if(posicion == 15)
15                  posicion = 0;
16          else
17                  posicion ++;
18
19          num_datos --;
20
21          if(num_datos == MAX_BUFFER -1)
22                  pthread_cond_signal(&lleno);
23
24          pthread_mutex_unlock(&mutex);
25          printf("\nse ha consumido el dato: %d", dato);
26          fflush(stdout);
27          sleep(1);
28         }
```

```
29          pthread_exit(0);
30  }
```

Listing 3: Proceso productor

```
1   * productor()
2   {
3           struct datos_tipo *datos_proceso;
4           int a, i, j, p, posicion = 0, dato;
5
6           for(i=0; i<MAX_DATOS; i++)
7           {
8                   pthread_mutex_lock(&mutex);
9                   while(num_datos == MAX_BUFFER)
10                          pthread_cond_wait(&lleno, &mutex);
11
12                  buffer[posicion] = i;
13                  dato = i;
14
15                  if(posicion == 15)
16                          posicion = 0;
17                  else
18                          posicion ++;
19                  num_datos ++;
20                  if(num_datos == 1)
21                          pthread_cond_signal(&vacio);
22
23                  pthread_mutex_unlock(&mutex);
24                  printf("\nse ha producido el dato: %d", dato);
25                  fflush(stdout);
26          }
27          pthread_exit(0);
28  }
```

Listing 4: Función main

```
1   main()
2   {
3   int error;
4   char *valor_devuelto;
5
6   /* Variables para hilos*/
7   pthread_t idhilo1, idhilo2, idhilo3, idhilo4;
8
9   pthread_mutex_init(&mutex, NULL);
10  pthread_cond_init(&lleno, NULL);
11  pthread_cond_init(&vacio, NULL);
12
13  error=pthread_create(&idhilo1, NULL, (void *)productor, NULL);
```

```
14  if (error != 0)
15  {
16          perror ("No puedo crear hilo");
17          exit (-1);
18  }
19
20  error=pthread_create(&idhilo2, NULL, (void *)consumidor, NULL);
21  if (error != 0)
22  {
23          perror ("No puedo crear thread");
24          exit (-1);
25  }
26
27  pthread_join(idhilo2, (void **)&valor_devuelto);
28  pthread_join(idhilo1, (void **)&valor_devuelto);
29  pthread_mutex_destroy(&mutex);
30  pthread_cond_destroy(&lleno);
31  pthread_cond_destroy(&vacio);
32  return 0;
33  }
```

Para esta práctica se pide:

1. Identifica los lugares del programa donde se bloquean los hilos para en los casos que se llena *buffer* y cuando está vacío.
2. Compila el programa y pruébalo. Comenta el resultado.
3. Prueba el programa sin el uso de las variables condicionales. ¿qué sucede?.
4. Implementa el programa haciendo uso de tuberías en vez de variables condicionales. Compila y prueba.