Portafolios Prácticas SCD

Problema del productor consumidor

Solución tipo LIFO

Para resolver el problema del productor-consumido podemos emplear una estructura de tipo pila (LIFO).

Utilizaremos una variable compartida que será el puntero de pila, en este caso será la variable `sp,` que se moverá sobre el `vector_pila.`

Utilizaremos tres semáforos, dos para controlar el productor y el consumido, otro para controlar el acceso a la variable `sp` y otro para la exclusión mutua del terminal, a continuación se describen cada uno de los semáforos.

```
// constantes
const unsigned
  num_items  = 40 ,
  tam_vector = 10 ;
// variables
unsigned int sp = 0;
// vector tipo pila
    int vector_pila[tam_vector];
// ---------------------------------------------------------------------
// semanforos
sem_t   puede_consumir;
sem_t   puede_producir;
sem_t   mutex_pila;
sem_t   mutex_terminal;
//----------------------------------------------------------------------
```

`puede_producir`: es el semáforo encargado de controlar el productor, por tanto, ya que el productor podría producir tantos datos como tamaño tenga el vector, antes de ser interrumpido, inicializaremos este semáforo con `tam_vector`. Cada vez que escribamos un dato en el vector debemos decrementar en uno el semáforo para controlar que no se escriba fuera del vector, haremos un `sem_wait(&puede_producir),` y cada vez que se lea un dato del vector incrementaremos en uno `puede_producir` para liberarlo, haremos `sem_post(&puede_producir)`, ya que si se ha leído un dato, al menos, se podrá producir otro dato.

`puede_consumir`: es el semáforo encargado de controlar al productor, este semáforo se inicializa a 0, ya que al principio de la ejecución no hay ningún dato disponible para ser consumido, cada vez que se produzca un dato, haremos `sem_post(&puede_consumir)`, para liberar al consumidor y poder leer el dato producido, de la misma manera, cada vez que se lea un dato debemos hacer

`sem_wait(&puede_consumir)` para asegurar que no accedemos a posiciones fuera del vector o que no leemos dos veces el mismo dato.

`mutex_pila`: debemos asegurarnos de que la variable que actúa como puntero de pila no sea manipulada en ningún momento por dos hebras a la vez, para ello utilizaremos el semáforo `mutex_pila`, con valor inicial 1.Cada vez que se ejecute una sección de código en la que se modifique el vector o el puntero de pila `sp` haremos `sem_wait(&mutex_pila)` y una vez terminada esta sección de código haremos un `sem_post(&mutex_pila).`

`mutex_terminal:` este semáforo se encarga de controlar la salidas por pantalla, lo inicializaremos a uno para permitir una primera salida y cada vez que se ejecute código que implique salida por pantalla haremos `sem_wait(&mutex_terminal)` para bloquearlo y `sem_signal(&mutex_terminal)` al final de la sección de código para liberarlo.

En la solución tipo LIFO lo principal es controlar el acceso a la variable compartida `sp`

Código de productor

```
void * productor( void * )
{
  for( unsigned i = 0 ; i < num_items ; i++ )
  {
     sem_wait(&mutex_terminal); //bloqueamos el terminal
    int dato = producir_dato();
     sem_post(&mutex_terminal); //liberamos el terminal
     sem_wait(&puede_producir);
     sem_wait(&mutex_pila);
    vector_pila[sp] = dato;
     sp++;
     sem_post(&mutex_pila);
     sem_post(&puede_consumir);
  }
  return NULL ;
}
```

Código de consumidor

```
void * consumidor( void * )
{
  for( unsigned i = 0 ; i < num_items ; i++ )
  {
    int dato;
     sem_wait(&puede_consumir);
     sem_wait(&mutex_pila);
     sp--;
     dato = vector_pila[sp];
     sem_post(&mutex_pila);
    sem_wait(&mutex_terminal); //bloqueamos el terminal
    consumir_dato( dato ) ;
     sem_post(&mutex_terminal); //liberamos el terminal
     sem_post(&puede_producir);
  }
  return NULL ;
}
```

Código del main

```
int main()
{
    sem_init (&puede_producir,0,tam_vector);
    sem_init (&puede_consumir,0,0);
    sem_init (&mutex_pila,0,1);
    sem_init (&mutex_terminal,0,1);

    pthread_t hebra_productora, hebra_consumidora;

    pthread_create(&hebra_productora,NULL,productor,NULL);
    pthread_create(&hebra_consumidora,NULL,consumidor,NULL);

    pthread_join(hebra_productora,NULL);
    pthread_join(hebra_consumidora,NULL);

    sem_destroy(&puede_consumir);
    sem_destroy(&puede_producir);
    sem_destroy(&mutex_terminal);
    sem_destroy(&mutex_pila);

   return 0 ;
}
```

Solución tipo FIFO

Para resolver el problema del productor-consumido podemos emplear una estructura de tipo cola (FIFO).

Utilizaremos un vector buffer y dos índices que se mueven sobre este vector `índice_c` e `índice_p`, para consumir y producir respectivamente.

Utilizaremos tres semáforos, dos para controlar el productor y el consumidor y otro para la exclusión mutua del terminal, a continuación se describen cada uno de los semáforos.

```
// constantes
const unsigned
  num_items  = 40 ,
  tam_vector = 10 ;
//----------------------------------------------------------------------
//buffer
unsigned int buffer[tam_vector];
unsigned int indice_c = 0;
unsigned int indice_p = 0;
//----------------------------------------------------------------------
// semaforos
 sem_t puede_consumir;
 sem_t puede_producir;
 sem_t mutex_terminal;
// ---------------------------------------------------------------------
```

puede_producir: es el semáforo encargado de controlar el productor, este semáforo se inicializa al tamaño del vector, puesto que podremos producir un numero de datos igual al tamaño del vector antes de empezar a “pisar” datos no leidos, para controlar esto, cada vez que se produzca un dato haremos un `sem_wait(&puede_producir)` para decrementar el contador del semáforo y tras haber producido el dato incrementamos el valor de `puede_consumir` con un `sem_post(puede_consumir)`, para dejar que se consuman valores producidos.

puede_consumir: es el semáforo encargado de controlar el consumidor, inicializaremos este semáforo a cero puesto que inicialmente no habrá valores disponibles para consumir, es el productor quien incrementa este semáforo. Cada vez que consumamos un dato deberemos decrementar `puede_consumir` con un `sem_wait(puede_conumir)`, asi como incrementar `puede_producir` con un `sem_post(puede_producir)`.

mutex_terminal: este semáforo es el encargado de controlar las salidas por pantalla, inicializado a uno para permitir una primera salida. Cada vez que ejecutemos código que implique una salida por pantalla haremos `sem_wait(mutex_terminal)` y tras la ejecución de este código `sem_post(mutex_terminal)`

En la solución tipo FIFO lo primordial es controlar el avance de la lectura de datos para mantenerla siempre por detrás del escritor, de igual forma que hay que controlar al escritor para que no escriba sobre datos aun no leidos.

Código de productor

```
void * productor( void * )
{
  for( unsigned i = 0 ; i < num_items ; i++ )
  {
    sem_wait (&mutex_terminal);
    int dato = producir_dato();
    sem_post (&mutex_terminal);
    sem_wait (&puede_producir);
    buffer[indice_p%tam_vector] = dato;
    indice_p++;
    sem_post (&puede_consumir);

  }
  return NULL ;
}
```

Código de consumidor

```
void * consumidor( void * )
{
  for( unsigned i = 0 ; i < num_items ; i++ )
  {
    int dato ;
    sem_wait (&puede_consumir);
    dato = buffer[indice_c%tam_vector];
    indice_c++;
    sem_post (&puede_producir);

    sem_wait (&mutex_terminal);
    consumir_dato( dato );
    sem_post (&mutex_terminal);
  }
  return NULL ;
}
```

Código del main

```
int main()
{

    sem_init (&puede_producir,0,tam_vector);
    sem_init (&puede_consumir,0,0);
    sem_init (&mutex_terminal,0,1);

    pthread_t hebra_productora, hebra_consumidora;

    pthread_create(&hebra_productora,NULL,productor,NULL);
    pthread_create(&hebra_consumidora,NULL,consumidor,NULL);

    pthread_join(hebra_productora,NULL);
    pthread_join(hebra_consumidora,NULL);

    sem_destroy(&puede_consumir);
    sem_destroy(&puede_producir);
    sem_destroy(&mutex_terminal);

   return 0 ;
}
```

Problema de los fumadores

Para resolver el problema de los fumadores implementaremos dos funciones, la función Estanco, que produce los ingredientes, y la función Fumadores, que consume los ingredientes.

Para controlar estas funciones utilizaremos cinco semáforos, tres para los fumadores, uno para el estanquero y otro para la exclusión mutua del terminal. Utilizaremos un vector de hebras y otro de semáforos para los fumadores.

Declaración de los semáforos

```
using namespace std;

const int NUMF = 3;

//Declaramos los semaforos que vamos a usar
sem_t fumadores[NUMF];
sem_t estanquero;
sem_t mutex;
```

Código del main

```
int main() {

  pthread_t hebras_f[NUMF];
  pthread_t estanquero_h;

//Incializamos los semanfors
  for (int i=0; i<NUMF; i++)
    sem_init(&fumadores[i],0,0);

  sem_init(&estanquero,0,1);
  sem_init(&mutex,0,1);

//Inicializamos las hebras
  for (unsigned int i = 0; i<NUMF; i++)
    pthread_create(&(hebras_f[i]),NULL,Fumadores,(void*)i);

  pthread_create(&estanquero_h,NULL,Estanco,NULL);

//Esperamos que terminen todas las hebras
  for (int i=0; i<NUMF; i++)
    pthread_join(hebras_f[i],NULL);

  pthread_join(estanquero_h,NULL);

//Destruimos los semaforos
  for (int i=0; i<NUMF; i++)
    sem_destroy(&fumadores[i]);

  sem_destroy(&estanquero);
  sem_destroy(&mutex);


  return 0 ;
}
```

Al estar los semáforos y las hebras declaradas en un vector, las identificaremos mediante su índice, de forma que la hebra uno estará controlada por el semáforo uno, de igual manera trataremos las otras dos. A continuación se describe el uso de los semáforos.

`fumadores`: estos semáforos son los encargados de controlar a las hebras que fuman, los inicializaremos a 0, puesto que no podrán fumar hasta que se produzca el ingrediente que necesitan, el uso de estos semáforos se describe más adelante.

`estanquero`: este semáforo es el encargado de controlar la función Estanco, que produce los ingredientes que necesitan los fumadores, inicializaremos este semáforo a 1 para poder producir al menos un ingrediente antes de producir.

`mutex`: utilizaremos este semáforo para la exclusión mutua del terminal, lo inicializaremos a 1 y cada vez que se ejecute código que implique una salida por pantalla haremos `sem_wait(&mutex)` y liberaremos el terminal con `sem_post(&mutex_terminal)`.

Función Estanco

```
void* Estanco (void *) {
    unsigned int ingrediente;

    while (true) {

        sem_wait(&estanquero);
        ingrediente = rand() % 3;

        sem_wait(&mutex);
        cout << "El estanquero pone en el mostrador el ingrediente " << ingrediente << "\n";
        sem_post(&mutex);

        sem_post(&fumadores[ingrediente]);

    }

    return NULL;
}
```

Es la función que produce los ingredientes, para producir los ingredientes utilizaremos la función `rand()`, que genera un numero aleatorio, en este caso, entre 0 y 2.

Cada vez que se vaya a producir un ingrediente deberemos decrementar en 1 el semáforo con `sem_wait(&estanquero)`, para asegurar que no se produzcan ingredientes antes de que algún fumador recoja el que se ha producido. Una vez generado el ingrediente despertaremos a la hebra que corresponde al ingrediente generado, para ello utilizaremos el numero aleatorio generado por `rand()`, almacenado en la variable `ingrediente` como índice del vector de semáforos y con `sem_post(&fumadores[ingrediente])`, despertaremos a la hebra.

Función Fumadores

```
void* Fumadores(void* ingrediente_f) {

    while (true) {

        sem_wait(&fumadores[(int)ingrediente_f]);

        sem_wait(&mutex);
        cout << "El fumador consume el ingrediente: " << (int)ingrediente_f << "\n";
        sem_post(&mutex);

        sem_post(&estanquero);
        fumar();

    }
    return NULL;
}
```

Es la función que consume los ingredientes producidos por el estanquero y simula la acción de fumar como un retardo aleatorio llamando a la función `fumar()`.

Esta función se ejecuta tres veces en paralelo para representar a los tres fumadores que consumen papel, tabaco y cerillas. Cada vez que se vaya a consumir un dato decrementaremos en uno el semáforo correspondiente al ingrediente que se le pasa a la función con `sem_wait(&fumadores[(int)ingrediente_f])`, así una vez consumido el ingrediente la hebra esperara a que se vuelva a producir el ingrediente que necesita. En esta función debemos liberar al estanquero para que vuelva a producir un ingrediente una vez consumido el que produjo por el fumador correspondiente, haremos un `sem_post(&estanquero)`, y para asegurar que dos fumadores puedan estar fumando a la vez, llamaremos a la función fumar tras liberar al estanquero.