Sincronización de productor-consumidor y semáforos.

- Producer-consumer synchronization se refiere a la coordinación entre dos procesos: uno que produce datos y los almacena en una estructura compartida, y otro que los consume. Este término describe la situación en la que un hilo no puede avanzar hasta que otro hilo tome cierta acción.

- La coordinación es esencial para evitar que el productor intente depositar datos en la estructura compartida cuando esté llena, así como para prevenir que el consumidor intente retirar datos cuando esté vacía. En algunas circunstancias, necesitamos gestionar la secuencia en la que los hilos acceden a los recursos.

Semaforos

En un entorno de concurrencia, se emplea para regular el acceso a recursos compartidos. Un semáforo posee un valor entero y permite dos operaciones principales.

- Wait: disminuye el valor del semáforo; si este valor resulta negativo, el proceso se detiene hasta que el semáforo sea no negativo.
- Signal: incrementa el valor del semáforo; si hay procesos bloqueados en espera, uno de ellos se libera.

En la sincronización entre productor y consumidor, se emplean dos semáforos para regular el acceso al buffer compartido y controlar el número de elementos en él. Esto asegura que el productor no pueda añadir datos si el buffer está lleno y que el consumidor no pueda retirar datos si está vacío.

Semaphores are not part of Pthreads; you need to add this.

```
#include <semaphore.h>

int sem_init(
      sem_t*     semaphore_p   /* out */,
      int        shared        /* in  */,
      unsigned   initial_val   /* in  */);

int sem_destroy(sem_t*   semaphore_p   /* in/out */);
int sem_post(sem_t*      semaphore_p   /* in/out */);
int sem_wait(sem_t*      semaphore_p   /* in/out */);
```

Barreras y variables de condición

Barreras

Se emplea para garantizar que todos los hilos estén sincronizados en un punto específico del programa. Ningún hilo puede avanzar más allá de esta barrera hasta que todos los demás hilos la hayan alcanzado.

```
point in program we want to reach;
barrier;
if (my_rank == 0) {
   printf("All threads reached this point\n");
   fflush(stdout);
}
```

Variables de condición.

Facilita que un hilo detenga su ejecución hasta que se cumpla una condición específica. Otro hilo puede luego notificar al hilo suspendido para que reanude su ejecución. Por lo general, se emplea en conjunto con un mutex.

```
lock mutex;
if condition has occurred
   signal thread(s);
else {
   unlock the mutex and block;
   /* when thread is unblocked, mutex is relocked */
}
unlock mutex;
```

Implementación: Barrera y semáforo

```
/* Shared variables */
int counter;          /* Initialize to 0 */
sem_t count_sem;      /* Initialize to 1 */
sem_t barrier_sem;    /* Initialize to 0 */
. . .
void* Thread_work(...) {
   . . .
   /* Barrier */
   sem_wait(&count_sem);
   if (counter == thread_count-1) {
      counter = 0;
      sem_post(&count_sem);
      for (j = 0; j < thread_count-1; j++)
         sem_post(&barrier_sem);
   } else {
      counter++;
      sem_post(&count_sem);
      sem_wait(&barrier_sem);
   }
   . . .
```

Implementación: Barrera y variables de condición

```
/* Shared */
int counter = 0;
pthread_mutex_t mutex;
pthread_cond_t cond_var;
. . .
void* Thread_work(. . .) {
    . . .
    /* Barrier */
    pthread_mutex_lock(&mutex);
    counter++;
    if (counter == thread_count) {
       counter = 0;
       pthread_cond_broadcast(&cond_var);
    } else {
       while (pthread_cond_wait(&cond_var, &mutex) != 0);
    }
    pthread_mutex_unlock(&mutex);
    . . .
 }
```