Grado en Ingeniería Informática
Sistemas Operativos II

Práctica 2: Sincronización de procesos con semáforos

Objetivo

Conocer el funcionamiento de los mecanismos de solución al problema de las carreras críticas, en particular, los semáforos. Desarrollar habilidades en su manejo.

Desarrollo

1. Estudia el problema del **productor-consumidor** visto en clase usando **procesos y espera activa** para dormir y despertar a los procesos**.**

- Comprueba que si el productor solamente escribe en el `buffer` y el consumidor lee en él utilizando la variable "`cuenta`" pueden presentarse *carreras críticas*. El tamaño del `buffer` debe ser `N=8`, definido como una constante del código.
    - Para comprobarlo puedes aumentar la posibilidad de que ocurran carreras críticas haciendo **que la región crítica dure más tiempo**, por ejemplo, con llamadas a `sleep` o `usleep` de forma que el productor vaya más rápido que el consumidor o viceversa.
- Constata la ocurrencia de carreras críticas.
    - Añade al código **salidas que muestren detalladamente** como van evolucionando tanto el productor como el consumidor.
- Dado que el consumidor y el productor serán dos **procesos diferentes** (*no uses threads*), es necesario un mecanismo para definir zonas de memoria donde almacenar **variables compartidas**:
    - Para definir en cada uno de esos procesos un espacio de memoria compartida con otros procesos, se puede mapear una zona de memoria en todos los procesos con la función **`mmap`**.
    - **Implementa** el productor y el consumidor creando **dos procesos**, uno para el productor y otro para el consumidor. Pueden ser dos programas diferentes o uno solo.

2. Usando como base el código anterior, programa la solución vista en clase usando semáforos teniendo en cuenta lo siguiente:

- El `buffer` debe ser **tipo `int`** y funcionar como una cola L**`IFO`** (*Last In First Out*).
- El **tamaño** del `buffer` debe ser `N=8`.

- La función **`produce_item()`** debe generar un entero aleatorio entre 0 y 10.
- La función **`insert_item()`** debe colocar el entero en el buffer.
- La función **`remove_item()`** debe retirar un entero del buffer. Además debe sumar al entero extraido todos los valores que en ese momento haya en el buffer.
- La función **`consume_item()`** debe mostrar por pantalla el entero leído así como el resultado de la suma de los valores contenidos en el buffer.
- Soluciona las carreras críticas utilizando semáforos teniendo en cuenta los siguientes puntos.
- Los programas deben incluir las siguientes **librerias**:
  `<sys/mman.h> <pthread.h> <semaphore.h> <unistd.h> <fcntl.h>`
- Las **variables** semáforo se declaran como punteros a estructuras de tipo *sem_t*:
  `sem_t *nombre_del_semaforo;`
- Para **crear un semáforo** usa la función *sem_open*. Un ejemplo de uso es:
  `vacias = sem_open("NOMBRE",O_CREAT,0700,N);`
- Uno de los procesos debe crear el semáforo e **inicializarlo** con la función anterior. Los procesos que usen ese semáforo posteriormente deben **abrirlo** pero no inicializarlo nuevamente. Para ello también usan la función *sem_open* del siguiente modo:
  `nombre_del_semaforo = sem_open("NOMBRE",0);`
- Para **cerrar los semáforos** y el espacio de memoria compartida, usa las funciones del siguiente ejemplo:
  ```
  munmap(buf, N*sizeof(char));
  sem_close(nombre_del_semaforo);
  ```
- Las funciones de **manejo de los semáforos** de POSIX son `sem_wait` y `sem_post`.
- Para no tener que abortar el programa en cada ejecución, no incluyas un lazo externo infinito, pon un **número de iteraciones finito** (por ejemplo 100).
- Añade llamadas a la función `sleep` en los códigos del productor y consumidor **fuera de la región crítica** para forzar situaciones en las que el productor y el consumidor vayan a **diferentes velocidades** de modo que eventualmente el buffer se llegue a llenar o a vaciar según el caso, para que los procesos se bloqueen alguna vez.
  - La versión final **debe incluir** llamadas a `sleep` de valores aleatorios entre `0 y 3` (para generar números aleatorios usa las funciones `rand` y `srand`).

- Al final deben eliminarse los semáforos usando la función **`sem_unlink`**. Si no se usa esta función, los semáforos quedarán activos en el kernel. Una buena práctica es eliminar los semáforos antes de crearlos al principio del programa.
- Añade salidas que muestren muy claramente como va evolucionando el consumo y producción de elementos a lo largo del tiempo.
- Ten en cuenta que las regiones críticas deben tener un tiempo de ejecución lo más pequeño posible.
- Para **compilar** se debe incluir la opción de compilación: **`-pthread`**.

3. EJERCICIO VOLUNTARIO 1: Resuelve el problema con **`threads`** en lugar de procesos.

4. EJERCICIO VOLUNTARIO 2: Generaliza el código con procesos para un número arbitrario seleccionado por el usuario.

5. EJERCICIO VOLUNTARIO 3: Realiza el ejercicio con procesos de manera que existe otra cola LIFO de 10 entradas para la que el productor juega el papel del consumidor y el consumidor juega el rol del productor. De tal manera que el productor, produce un elemento para la primera cola y posteriormente consume otro de la segunda. Por su parte el consumidor hace la operación simétrica.

Formato y fecha de entrega

Hacer un informe breve incluyendo comentarios y las conclusiones obtenidas para los apartados 1 y 2. Sube un único archivo comprimido incluyendo los códigos de los dos ejercicios con comentarios exhaustivos sobre su funcionamiento.

En su caso, para cada uno de los ejercicios voluntarios, debe incluirse la misma información en un archivo comprimido.

Para cada grupo, la fecha de entrega será la indicada en el Campus Virtual.