Pontificia Universidad JAVERIANA Colombia

Sistemas Operativos

Docente J. Corredor.

Autores:Santiago Hernandez Morales, Jose Jesus Cepeda Vargas, Karol Torres, Andres Meneses.

Informe Taller Posix Sincronización

Estructura Código

Fichero producer.c

Fichero consumer.c

Fichero posixSincro.c

Fichero Bonus concurrenciaPosix.c

Análisis Actividad 1

Se conoce que ‘producer’ y ‘consumer’ no están relacionados, es decir, no son procesos padre e hijo. Entonces, ¿Cómo se busca garantizar en el programa una comunicación inequívoca?

Esto es, con named semaphores. En el código de este taller, se hace uso de uno para contar los espacios vacíos y otro para indicar los espacios llenos, se corrige el problema de intentar consumir vacío y de producir en donde ya hay. ¿Cómo se hace esto en el codigo? Así:

```
sem_t *vacio = sem_open("/vacio", 0);
sem_t *lleno = sem_open("/lleno", 0);
```

Aquí, los semáforos con nombre que tenemos funcionarían como unos contadores especializados, ¿Cómo?, El semáforo llamado ‘vacio’ cuanta cuantos espacios están libres y el semáforo que se llama ‘lleno’ cuenta cuantos espacios están ocupados.

En el código, el productor hace sem_wait(vacio) antes de escribir (si el contador ‘vacio’==0 se espera) y sem_post(lleno) después de escribir(como si publicara:”Acabo de colocar algo en el buffer”). De esta forma, nunca produce en una posición que ya está ocupada. El consumidor hace esto pero al contrario: sem_wait(lleno) antes de leer (si no hay nada para consumir, es decir ‘lleno’==0, se espera) y sem_post(vacio) después de leer (ahora hay un espacio), evitando intentar consumir cuando el búfer está vacío.

¿Por que se hace sem_wait(vacio) antes de escribir en el buffer?

El orden de las instrucciones va así:

a. sem_wait(vacio); "¿Hay 'cajones' disponibles para escribir? Si no, espero."
b. Escribo en el buffer
c. sem_post(lleno); "Listo, ya puse un ítem más, hay uno lleno extra."

Así, se garantiza que el producer nunca escribirá cuando el búfer esté lleno, porque si vacio llega a 0, se queda esperando hasta que el consumer libere un hueco.

Si lo hiciera al revés (escribir primero y luego sem_wait(vacio)), podría escribir en un lugar donde no hay espacio reservado y se pisarían datos.

Análisis Actividad 2

En la Actividad 2 se trabaja la concurrencia dentro de un mismo proceso usando hilos POSIX (pthreads). A diferencia de la Actividad 1, aquí todos los hilos comparten directamente las variables globales, por lo que el problema central es proteger el acceso concurrente a esos datos. En el programa posixSincro.c varios hilos productores escriben mensajes en un búfer global de cadenas, mientras que un hilo se encarga de leer ese búfer e imprimir las líneas.

Para evitar Race Condition se utiliza un mutex (pthread_mutex_t) que protege las variables compartidas (el búfer, los índices y los contadores), y dos variables de condición (pthread_cond_t): una para esperar a que haya espacio libre en el búfer y otra para esperar a que haya líneas por imprimir. Los productores solo escriben cuando buffers_available indica que hay huecos; si no, se duermen con pthread_cond_wait. El hilo solo imprime cuando lines_to_print es mayor que cero; si no, también espera. Cada vez que algún hilo cambia esos contadores, despierta al otro lado con pthread_cond_signal.

En el ejemplo de concurrenciaPOsix.txt se muestra otro patrón: se reparten los datos de un vector entre varios hilos, cada uno calcula un máximo parcial sobre su segmento y luego el hilo principal combina esos resultados. Aquí los hilos también comparten memoria, pero cada uno trabaja sobre un rango distinto, por lo que se reduce la necesidad de sincronización fina. En conjunto, la actividad 2 permite entender cómo, dentro de un solo proceso, la combinación de mutex y variables de condición es la herramienta básica para coordinar hilos, evitar condiciones de carrera y estructurar patrones como productor–consumidor y procesamiento paralelo de datos.

Conclusiones

El desarrollo del taller permitió entender cómo funcionan los distintos mecanismos de comunicación y sincronización en POSIX, (productor y consumidor).

En la primera parte al implementar el problema en el problema de productor-consumidor usando semáforos con nombre y memoria compartida, se logra entender como dos procesos totalmente independientes pueden coordinarse sin estar necesariamente relacionados. La memoria compartida permite que ambos procesos pueden acceder al mismo espacio de los datos y los semáforos se encargaron de controlar cuando el productor podía escribir y cuando el consumidor podía leer.

Esto permite comprender la importancia de sincronizar correctamente el acceso al buffer para poder evitar errores, por ejemplo intentar consumir cuando no hay datos o producir cuando el buffer está lleno.

En la parte relacionada con hilos (POSIX), se trabajó con la búsqueda del máximo vector, se logró observar como al dividir una tarea en varios hilos puede mejorar el rendimiento. También aunque los hilos comparten la memoria facilita el trabajo para los hilos, pero si no se manejan los bloqueos se pueden generar problemas cuando varios hilos intentan acceder al mismo recurso al mismo tiempo.

En el ejercicio de productor usando las variables de mutex y condiciones, permite entender que aunque los hilos comparten memoria es necesario controlar los accesos para evitar errores, mutex asegura que un solo hilo usará el buffer a la vez y las variables de condición permiten que los demás hilos esperen hasta que exista el espacio disponible.

El taller permitió entender cuando es más conveniente usar procesos con semáforos, cuando es mejor usar memoria compartida y cuando es mejor usar hilos con mutex y variables de condición, además de comprender que si se va a usar concurrencia es necesario tener cuidado con las condiciones de carrera o bloqueos.