Sincronización de Procesos

UNIVERSIDAD DEL BÍO-BÍO

Agenda

1. Objetivos
2. Fundamentos
3. El problema de la sección crítica

UNIVERSIDAD DEL BÍO-BÍO

Objetivos

- Presentar el problema de las secciones críticas, cuyas soluciones pueden utilizarse para asegurar la coherencia de los datos compartidos.
- Presentar soluciones tanto software como hardware para el problema de las secciones críticas.

Fundamentos

- Acceso concurrente a datos compartidos puede resultar con datos inconsistentes.
- Mantener datos consistentes requiere de mecanismos para asegurar la ejecución ordenada de procesos cooperativos.

Problema Consumidor-Productor

- Suponga una solución para el problema del consumidor-productor con buffer limitados.
- Utilizando un entero **counter** para manejar los buffers. Inicializado en 0, se incrementa cada vez que se añade nuevo elemento y se decrementa cada vez que se elimina un elemento del buffer.

Variables compartidas por procesos C - P

```
#define BUFFER_SIZE 10

typedef struct {
    ....
} item;

item buffer[BUFFER_SIZE];
int in = 0;
int out = 0;
int counter = 0;
```

Variables compartidas por procesos C - P

```
// Productor.
item nextProduced;

while (true) {
  /* produce an item and
  put in nextProduced */

  // do nothing (full)
  while (counter == BUFFER_SIZE);

  buffer[in] = nextProduced;
  in = (in + 1) % BUFFER_SIZE;
  counter++;
}
```

```
// Consumidor.
item nextConsumed;

while (true) {
  // do nothing (empty)
  while (counter == 0);

  nextConsumed = buffer[out];
  out = (out + 1) % BUFFER_SIZE;
  counter--;

  /* consume the item
  in nextConsumed */
}
```

UNIVERSIDAD DEL BÍO-BÍO

Condición de Carrera (Race Condition)

- Ambas rutinas son correctas por separado, no pueden funcionar correctamente cuando se ejecutan de forma concurrente.
- Se puede demostrar que el valor calculado en algún momento puede ser incorrecto.

Condición de Carrera (Race Condition)

```
counter++ puede ser implementado como:
  register1 = counter
  register1 = register1 + 1
  counter = register1

counter-- puede ser implementado como:
  register2 = counter
  register2 = register2 - 1
  counter = register2
```

Condición de Carrera (Race Condition)

Considere esta ejecución con counter = 5 inicialmente:

```
T0: prod. register1 = counter       {register1 = 5}
T1: prod. register1 = register1 + 1 {register1 = 6}
T2: cons. register2 = counter       {register2 = 5}
T3: cons. register2 = register2 - 1 {register2 = 4}
T4: prod. counter = register1       {counter = 6}
T5: cons. counter = register2       {counter = 4}
```

Hemos llegado al estado incorrecto **counter=4**, cuando el valor correcto es 5.

UNIVERSIDAD DEL BÍO-BÍO

Condición de Carrera (Race Condition)

- Ejecución de counter++ y counter– es equivalente a una ejecución secuencial donde:
  - instrucciones de menor nivel (registros) se intercalan en cierto orden aleatorio, pero el orden de cada instrucción de alto nivel se conserva.
- Se llega a este estado incorrecto porque se ha permitido que ambos procesos manipulen la variable **counter** de forma concurrente.
- Una situación como ésta, donde varios procesos manipulan y acceden a los mismos datos concurrentemente y el resultado de la ejecución depende del orden concreto en que se produzcan los accesos, se conoce como **Condición de Carrera**.

UNIVERSIDAD DEL BÍO-BÍO

El problema de la sección crítica

- Cada proceso tiene un segmento de código llamada **Sección Crítica**
  - cada proceso puede modificar variables comunes, actualizar tablas, escribir archivos, etc.
- Idea principal
  - Cuando un proceso está ejecutando su SC, ningún otro proceso puede ejecutar su correspondiente SC. **Dos procesos no pueden ejecutar su SC al mismo tiempo.**
- El problema de la SC consiste en:
  - Diseñar un protocolo que los procesos puedan usar para cooperar en tal escenario.
- Estructura general de un proceso típico es:
  - **sección de entrada**, solicitud para entrar a SC.
  - **sección crítica**
  - **sección de salida**
  - **sección restante**

UNIVERSIDAD DEL BÍO-BÍO

Estructura general de un proceso

```
do {
     ------------------
    |sección de entrada|
     ------------------
     sección crítica
     -----------------
    |sección de salida|
     -----------------
    sección restante
} while {TRUE};
```

El problema de la sección crítica

- Cualquier solución al problema de la SC debe satisfacer los tres siguientes requisitos:
  - **Exclusión mutua**
    - Si proceso P está ejecutando su SC, los demás procesos no pueden estar ejecutando sus SC.
  - **Progreso**
    - Si ningún proceso está ejecutando su SC y algunos procesos desean entrar a sus SC, sólo aquellos procesos que no estén ejecutando sus secciones restantes participan de la decisión de cual entrará a su SC.
    - Tal decisión no puede posponerse indefinidamente.
  - **Espera limitada**
    - Existe límite en el número de veces que se permite que otros procesos entren en sus SC después de que otro proceso haya solicitado entrar a su SC.

UNIVERSIDAD DEL BÍO-BÍO

El problema de la sección crítica

- Procesos en Modo Kernel pueden estar sujeto a posibles condiciones de carrera.
  - Ej. estructura de datos del kernel que mantenga lista de todos los archivos abiertos en el sistema. Tal lista se modifica cuando se abre (añade) o cierra (elimina) un archivo. ¿Qué sucede si dos procesos abren simultáneamente archivos?

Métodos para gestionar las SC en los SSOO

- **Kernel Apropiativos**
  - proceso puede ser desalojado mientras se ejecuta en modo kernel.
  - especialmente difíciles de implementar en arquitecturas SMP.
  - Versiones comerciales de UNIX (Solaris, IRIX)
  - Linux V2.6 (Procesor type and features – Preemption Model – ...)
  - Windows 7, 10.
- **Kernel No Apropiativos**
  - proceso no puede ser desalojado, se ejecuta hasta que salga de dicho modo, se bloquee o hasta que ceda voluntariamente el control de la CPU.
  - Libre de condiciones de carrera, sólo un proceso activo en el kernel en cada momento.
  - Anteriores a Windows 95, kernel tradicional de UNIX. Linux hasta antes de V2.6

Soluciones a SC

- En general, cualquier solución requiere una herramienta muy simple, un **Cerrojo**. Condiciones de carrera se evitan requiriendo que las regiones críticas se protejan mediante cerrojos.

```
do {
     ------------------
    | adquirir cerrojo |
     ------------------
       sección crítica
     -----------------
    | liberar cerrojo |
     -----------------
       sección restante
} while (TRUE);
```

Soluciones a SC

- Hardware de sincronización.
  - Soporte por HW para la SC.
  - Monoprocesadores deshabilitan las interrupciones.
  - Ejecución de código debe ser sin Apropiación.
  - Instrucción TestAndSet() para leer y modificar atómicamente una variable. Es una instrucción de máquina que no puede ser interrumpida por ningún proceso.
  - Implementación en multiprocesador es más difícil y costoso.

UNIVERSIDAD DEL BÍO-BÍO

Sincronización en Linux con Pthreads

- Esta API proporciona los cerrojos **mutex**, variables de condición y cerrojos de lectura-escritura para la sincronización de hebras.
- Disponible para programadores y no forma parte de ningún kernel.
- #include <pthread.h>
- pthread_mutex_t mutex;
- pthread_mutex_init (&mutex, NULL);
- pthread_mutex_lock (&mutex);
- pthread_mutex_unlock (&mutex);

UNIVERSIDAD DEL BÍO-BÍO

Sincronización en Linux con clase thread (C++11)

- #include <mutex>
- #include <thread>
- std::mutex mutex;
- mutex.lock();
- mutex.unlock();