Sincronización de Procesos

Semestre II-2013

Objetivos

- Introducir el problema de la sección crítica, revisar que soluciones pueden ser usadas para asegurar consistencia en datos compartidos
- Presentar las soluciones de hardware y software al problema de la sección crítica
- Examinar los problemas clásicos de sincronización de procesos
- Explorar las diversas herramientas disponibles para resolver problemas relacionados con la sincronización de procesos

Antecedentes - Motivación

- Los procesos se ejecutan de manera concurrente
  - Una interrupción puede ocurrir en cualquier momento, lo cual podría ocasionar que la ejecución parcial
- El acceso concurrente a datos compartidos podría resultar en datos inconsistentes
- Mantener datos consistentes requiere mecanismos para asegurar una ejecución ordenada entre procesos cooperantes
- Imagine lo siguiente:
  - Suponga que nosotros deseamos proveer una solución al problema del productor-consumidor

Antecedentes - Motivación

- Imagine lo siguiente:
  - Suponga que nosotros deseamos proveer una solución al problema del productor-consumidor para llenar los elementos de un buffer. Nosotros podríamos tener un entero **counter** para seguir la pista de el número de elementos en el buffer. Inicialmente, **counter** se fija en el valor 0
  - **counter** es incrementado por el productor después de colocar un elemento en el buffer, y se decrementa por el consumidor después de que este consuma un elemento

Productor

```
while (true) {
      /* produce an item in next produced */

      while (counter == BUFFER SIZE) ;
            /* do nothing */
      buffer[in] = next produced;
      in = (in + 1) % BUFFER SIZE;
      counter++;
}
```

Consumidor

```
while (true) {
        while (counter == 0)
                ; /* do nothing */
        next consumed = buffer[out];
        out = (out + 1) % BUFFER SIZE;        counter--;
        /* consume the item in next consumed */
}
```

Condición de Carrera

- **counter++** podría implementarse como:
  - `register1 = counter`
    `register1 = register1 + 1`
    `counter = register1`
- **counter--** podría implementarse como:
  - **register2 = counter**
    register2 = register2 - 1
    counter = register2
- Considere esta ejecución intercalada con "count=5"
  - S0: producer execute `register1 = counter` {register1 = 5}
    S1: producer execute `register1 = register1 + 1` {register1 = 6}
    S2: consumer execute `register2 = counter` {register2 = 5}
    S3: consumer execute `register2 = register2 - 1` {register2 = 4}
    S4: producer execute `counter = register1` {counter = 6 }
    S5: consumer execute `counter = register2` {counter = 4}

Problema de la Sección Crítica

- Considere un sistema con n procesos $\{p_0, p_1, \ldots p_{n-1}\}$
- Cada proceso tiene un segmento de código llamado **sección crítica**
  - Los procesos pueden modificar los valores de variables comunes, actualizar tablas, escribir archivos, etc.
  - Cuando un proceso se encuentra en su sección crítica, ningún otro proceso en el sistema puede estar en su sección crítica
- Existe un protocolo diseñado para solventar el problema de la sección critica
- Cada proceso debe solicitar permiso antes de entrar en su sección critica (sección de entrada), seguido ejecuta su sección critica, y luego sale de la sección critica (sección de salida). Todo lo demas se considera sección remanente

Sección Crítica

- La estructura general de un proceso Pi es:

```
do {
        entry section
                critical section
        exit section
                remainder section
} while (true);
```

Solución al Problema de la Sección-Critica

- Exclusión Mutua – Si un proceso Pi se encuentra ejecutando su sección critica, entonces ningún otro proceso puede ejecutar su sección crítica
- Progreso – Si no existe un proceso ejecutando su sección critica y hay algunos procesos que desean entran en su sección critica, la selección del próximo proceso que entrará en su sección critica no debe posponerse indefinidamente
- Limite Finito – Debe existir un limite en el número de veces que a otros procesos se les permite entrar en su sección crítica después de que un proceso ha solicitado entrar en su sección crítica, y antes de conceder dicha solicitud
  - Se asume que los procesos se ejecutan a una velocidad diferente de cero
  - No se realizan presunciones sobre la velocidad relativa de los N procesos

Solución al Problema de la Sección-Critica

- Dos aproximaciones dependiendo si el kernel es apropiativo o no apropiativo
  - Apropiativo – Se permite apropiarse de los procesos cuando estos se ejecutan en modo kernel
  - No Apropiativo – La ejecución continua hasta salir del modo kernel, se bloquea, o voluntariamente cede el control del CPU
    - Esencialmente no existe condición de carrera en modo kernel

Solución de Peterson

- Buena descripción algorítmica a la solución del problema
- Solución para dos procesos
- Se asume que las instrucciones **load** y **store** son atómicas, esto es, que su ejecución no puede ser interrumpida
- Los dos proceso comparten dos variables:
  - int **turn**
  - Boolean **flag**[2]
- La variable **turn** indica a quien le corresponde entrar a la sección crítica
- El arreglo **flag** es usado para indicar si un proceso esta listo para en su sección crítica. **flag[i] = true** indica que el proceso Pi esta listo

Algoritmo para el Proceso Pi

```
do {
     flag[i] = true;
     turn = j;
     while (flag[j] && turn == j);
           critical section
     flag[i] = false;
           remainder section
} while (true);
```

- Es posible demostrar que:
  - La exclusión mutua es preservada
  - El requerimiento de progreso es satisfecho
  - El requerimiento de limite finito se cumple

Sincronización por Hardware

- Muchos sistemas proveen soporte vía hardware para el código de secciones críticas
- Todas las soluciones abajo se encuentra basadas en la idea de bloqueo
  - Proteger la sección crítica vía un bloqueo o cerradura
- Uniprocesador – Podrían deshabilitarse las interrupciones
  - La ejecución de código concurrente podría realizarse sin apropiación
  - Generalmente es muy ineficiente en sistemas multiprocesador
    - Un SO que implemente esta solución no es escalable
- Las máquinas modernas proveen instrucciones de hardware especiales y atómicas
  - Como leer un palabra de memoria y establecerle un valor
  - Intercambiar el contenido entre dos palabras de memoria

Solución al Problema de Sección Crítica Usando Bloqueo

```
do {
     acquire lock
           critical section
     release lock
           remainder section
} while (TRUE);
```

Instrucción test_and_set

- Definición:

```
boolean test_and_set (boolean *target)
          {
               boolean rv = *target;
               *target = TRUE;
               return rv:
          }
```

Solución usando test_and_set()

- Se comparte una variable booleana **lock** inicializada en **FALSE**
- Solución:

```
do {
   while (test_and_set(&lock))
      ; /* do nothing */
   /* critical section */
   lock = false;
   /* remainder section */
} while (true);
```

Instrucción compare_and_swap

- **Definición:**

```
int compare and swap(int *value, int expected, int
new value) {
    int temp = *value;
    if (*value == expected)
        *value = new value;
    return temp;
}
```

Solución usando compare_and_swap

- Se comparte una variable booleana compartida **lock** inicializada en **FALSE**, adicionalmente cada proceso tiene una variable booleana local **key**

```
do {
   while (compare and swap(&lock, 0, 1) != 0)
      ; /* do nothing */
      /* critical section */
   lock = 0;
      /* remainder section */
} while (true);
```

Limite de Espera y Exclusión Mutua con test_and_set

```
do {
   waiting[i] = true;
   key = true;
   while (waiting[i] && key)
      key = test_and_set(&lock);
   waiting[i] = false;
   /* critical section */
   j = (i + 1) % n;
   while ((j != i) && !waiting[j])
      j = (j + 1) % n;
   if (j == i)
      lock = false;
   else
      waiting[j] = false;
   /* remainder section */
} while (true);
```

Bloqueo con Mutex

- Las soluciones previas pueden llegar a ser complejas y generalmente se encuentra inaccesibles a los programas de aplicación
- Los diseñadores de SO deben construir herramientas de software para lidiar con los problemas de sección crítica
- Una solución sencilla, funcional y elegante
  - Bloqueo con mutex
- Antes de entrar a una sección crítica primero debe adquirirse (**acquire()**) el bloqueo o cerrojo y al finalizar debe liberarse (**release()**)
  - Una variable booleana indica si el bloqueo o cerrojo se encuentra disponible
- Las llamadas a **acquire()** y **release()** deben ser atómicas
  - Usualmente se implementan vía instrucciones atómicas en hardware
- Sin embargo, esta solución requiere espera activa
  - Este tipo de solución recibe el nombre de **spinlock**

acquire() y release()

```
acquire() {
   while (!available)
      ; /* busy wait */
   available = false;;
}
release() {
   available = true;
}

do {
   acquire lock
      critical section
   release lock
      remainder section
} while (true);
```

Semáforo

- Herramienta de sincronización para evitar la espera activa
- Semáforo S – Variable Entera
- Dos operaciones estándares pueden modificar a S
  - wait() y signal()
  - Originalmente llamadas P() y V()
- Solución menos compleja
- Solo es posible acceder a la sección crítica vía dos operaciones indivisibles (atómicas)

Semáforo

```
wait (S) {
    while (S <= 0)
        ; // busy wait
    S--;
}


signal (S) {
    S++;
}
```

Uso de Semáforos

- Semáforo Contador – Valor entero que puede oscilar en un intervalo restringido
- Semáforo Binario – Valor entero cuto rango oscila entre 0 y 1
  - Entonces es un Bloqueo estilo Mutex
- Es posible implementar un semáforo contador S como un semáforo binario
- Es posible resolver varios problemas de sincronización
- Considere P1 y P2 los cuales requieren S1 antes que S2

```
P1:
    S1;
    signal(synch);
P2:
    wait(synch);
    S2;
```

Implementación de Semáforos

- Se debe garantizar que dos procesos no puedan ejecutar las instrucciones wait() o signal() sobre el mismo semáforo al mismo tiempo
- De esta manera, la implementación de wait y signal se convierte en un problema de sección crítica
  - Podríamos tener una implementación de sección crítica con **espera activa**
    - La implementación a nivel de código es corta
    - La sección crítica es ocupada ocasionalmente
- Note que las aplicaciones podrían gastar una cantidad de tiempo importante en sus secciones críticas, lo cual no es una buena solución

Implementación de Semáforo sin Espera Activa

- Con cada semáforo se asocia una cola de espera
- Cada entrada en la cola de espera posee los siguientes datos:
  - Valor (del tipo entero)
  - Un apuntador al próximo registro en la lista
- Dos operaciones:
  - **block** – Coloca al proceso que realizó la invocación en la cola de espera apropiada
  - **wakeup** – Remueve uno de los procesos de la cola de espera y lo coloca en la cola de listo

Implementación de Semáforo sin Espera Activa

```
typedef struct{
   int value;
   struct process *list;
} semaphore;
wait(semaphore *S) {
   S->value--;
   if (S->value < 0) {
      add this process to S->list;
      block();
   }
}
signal(semaphore *S) {
   S->value++;
   if (S->value <= 0) {
      remove a process P from S->list;
      wakeup(P);
   }
}
```

Deadlock e Inanición

- **Deadlock** – Dos o más proceso se encuentran esperando de manera indefinida por un evento que solo puede generar uno de los procesos que se encuentran esperando
- Sean S y Q dos semáforos inicializados en 1

P_0

```
wait(S);
wait(Q);
  .
signal(S);
signal(Q);
```

P_1

```
wait(Q);
wait(S);
  .
signal(Q);
signal(S);
```

Deadlock e Inanición

- Inanición (starvation) – Bloqueo indefinido
  - Un proceso podría nunca ser removido de la cola de espera del semáforo por la cual se encuentra esperando
- Inversión de prioridades – Problema de planificación que se genera cuando procesos de baja prioridad mantienen un bloqueo que requiere un proceso de mayor prioridad
  - Se solventa utilizando un protocolo de prioridades inherentes

Problemas Clásicos de Sincronización

- Estos problemas se usan para probar las nuevas propuestas de sincronización de procesos:
  - Problema del buffer limitado (productor/consumidor)
  - Problema de Lectores/Escritores
  - Problema de la Cena de los Filósofos

Problema del Buffer Limitado

- N buffers, cada uno mantiene un elemento
- Semáforo mutex inicializado en el valor 1
- Semáforo full inicializado en el valor 0
- Semáforo empty inicializado en el valor N

Problema del Buffer Limitado

- La estructura del proceso productor

```
do {
        ...
    /* produce an item in next_produced */
        ...
    wait(empty);
    wait(mutex);
        ...
    /* add next produced to the buffer */
        ...
    signal(mutex);
    signal(full);
} while (true);
```

Problema del Buffer Limitado

- La estructura del proceso consumidor

```
do {
    wait(full);
    wait(mutex);
        ...
    /* remove an item from buffer to next_consumed */
        ...
    signal(mutex);
    signal(empty);
        ...
    /* consume the item in next consumed */
        ...
} while (true);
```

Problema de los Lectores/Escritores

- Un conjunto de datos es compartido por un conjunto de procesos concurrentes
  - Lectores – Solo realizan operaciones de lectura, ellos no realizan ninguna actualización
  - Escritores – Pueden leer y escribir en el conjunto de datos
- Problemas
  - ¿Permitir el acceso a múltiples lectores al mismo tiempo?
  - ¿Permitir el acceso a múltiples escritores al mismo tiempo?
    - Solo un escritor debe tener acceso al conjunto de datos compartidos en el mismo instante de tiempo
- Datos compartidos
  - Conjunto de datos
  - Semáforo **rw_mutex** inicializado en 1
  - Semáforo **mutex** inicializado en 0
  - Entero **read_count** inicializado en 0

Problema de los Lectores/Escritores

- Estructura del proceso escritor

```
do {
    wait(rw mutex);
        ...
    /* writing is performed */
        ...
    signal(rw mutex);
} while (true);
```

Problema de los Lectores/Escritores

- Estructura del proceso lector

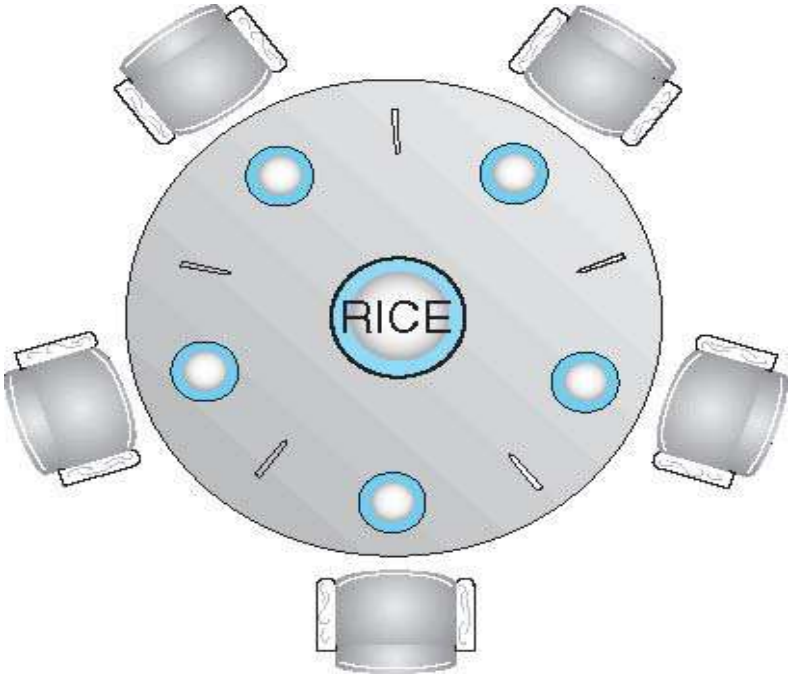
```
do {
wait(mutex);
read count++;
if (read count == 1)
wait(rw mutex); signal(mutex);
...
/* reading is performed */
... wait(mutex);
read count--;
if (read count == 0)
signal(rw mutex); signal(mutex);
} while (true);
```

Variantes del Problema Lectores/Escritores

- Primera Variante – Ningún lector debe mantenerse en espera, al menos que un escritor tenga permiso de usar el conjunto de datos compartido
- Segunda Variante – Cuando un escritor se encuentra listo, este debe realizar la escritura tan pronto como sea posible
- Ambas variantes pueden conducir a inanición
  - ¿Por qué?
- El problema en solventado en algunos sistemas usando bloqueos de lectura-escritura a nivel de kernel

Problema de la Cena de los Filósofos

- Los filósofos invierten su vida en dos actividades: pensar y comer
- Estos no interactúan con sus vecinos, ocasionalmente un filósofo trata de tomar 2 palillos (uno a la vez) para comer de su plato, luego de servirse del bowl principal
  - Se necesitan ambos palillos para comer, luego de comer los palillos son liberados
- En el caso de 5 filósofos
  - Datos compartidos
    - Bowl de arroz (conjunto de datos)
    - Semáforo chopstick[5] inicializado en 1



Problema de la Cena de los Filósofos

- La estructura del filósofo i:

```
do  {
        wait ( chopstick[i] );
         wait ( chopStick[ (i + 1) % 5] );

              //  eat

          signal ( chopstick[i] );
          signal (chopstick[ (i + 1) % 5] );

              //  think

} while (TRUE);
```

- ¿Cuál es el problema con este algoritmo?

Problema con los Semáforos

- Uso incorrecto de las operaciones en los semáforos
  - signal (mutex) …. wait (mutex)
  - wait (mutex) … wait (mutex)
  - Omisión del wait (mutex) o signal (mutex) (o ambos)
- Deadlock and starvation

Ejemplos de Sincronización

- Solaris
- Windows XP
- Linux
- Pthreads

Sincronización en Solaris

- Implementa una variedad de bloqueos para dar soporte a multitasking, multithreading (incluyendo threads en tiempo real), y multiprocesamiento
- Usa mutexes adaptativos por eficacia cuando se pretende proteger datos de segmentos cortos de código
  - El estándar en la implementación se basa en spin-lock
  - Si se da un bloqueo, y existe un thread ejecutándose en otro CPU, spins
  - Si el bloqueo ocurre por un thread el cual no esta en ejecución, entonces se bloquea y duerme en espera de una señal que indique que el bloqueo ha sido liberado
- Usa variables de condición
- Usa bloqueo de lectura/escritura cuando secciones extensas de código requieren acceso a datos
- Usa torniquetes para ordenar la lista de threads que esperan la adquisición de bloqueos adaptativos de tipo mutex o lectura/escritura
- Una prioridad inherentes por torniquete le otorga la más alta prioridad al thread en ejecución

Sincronización en Windows XP

- Utiliza enmascaramiento de interrupciones para proteger el acceso a recursos globales en sistemas uniprocesador
- Usa spinlocks en sistemas multiprocesador
  - El spinlock a nivel de threads nunca será apropiativo
- También provee un despachador de eventos que puede usarse para simular mutexes, semáforos, eventos, y temporizadores
  - Eventos
    - Un evento actúa como una variable de condición
  - Los temporizadores solo notifican a uno o más threads cuando ha expirado una marca de tiempo

Sincronización en Linux

- Linux
  - Por debajo de la Versión de kernel 2.6, deshabilita las interrupciones para la implementación de secciones críticas cortas
  - Versiones 2.6 y posteriores son netamente apropiativas
- Linux provee
  - Semáforos
  - Spinlocks
  - Bloqueo estilo lectura/escritura
- En sistemas de un solo CPU, spinlocks reemplazan la habilitación o no de apropiatividad a nivel del kernel

Sincronización en Pthreads

- El API de Pthreads es independiente del SO
- Este provee:
  - Bloqueo estilo mutex
  - Variables de condición
- Incluye una extensión no portable:
  - Bloqueos estilo lectura/escritura
  - Spinlocks

Videos

- http://www.youtube.com/watch?v=3TJHdETtfhE
- http://williamstallings.com/OS/Animation/Animations.html