High Performance Computing

Concurrency

Fernando R. Rannou
Departamento de Ingeniería Informática
Universidad de Santiago de Chile

March 25, 2024

Concurrencia y sincronización

- Cuando dos o más tareas (procesos o hebras) comparten algún recurso en forma concurrente o paralela, debe existir un mecanismo que sincronice sus actividades
- El riesgo: corrupción del recurso compartido
- Ejemplos:
  1. Acceso a impresora en un sistema distribuido
  2. Consulta y actualización de una base de datos
  3. Ejecución paralela de tareas para resolver un problema en conjunto
- Los problemas de sincronización no sólo ocurren en sistemas multiprocesadores, sino también (y fundamentalmente) en monoprocesadores

Beneficios de concurrencia

- Aumento en el rendimiento mediante múltiples procesadores
  - Paralelismo
- Aumento del throughput de las aplicaciones
  - Un llamado a una operación de I/O sólo bloquea una hebra
- Mejora en interactividad de las aplicaciones
  - Asignar mayor prioridad a hebras que atiendan al usuario
- Mejora en la estructuración de aplicaciones
- Sistemas distribuidos

Ejemplo de problema de concurrencia

Considere dos hebras que ejecutan concurrentemente la función `Insert(item)` que inserta un elemento en una lista compartida

```
typedef struct list {
   int data;
   struct list *next;
} List;

void insert(int item) {
  struct list *p;

  p = malloc(sizeof(struct list));
  p->data = item;
  p->next = List;
  List = l;
}
```

Ambas hebras procuran e insertan correctamente el item en un nodo, pero dependiendo del orden de ejecución, es probable que sólo un nodo se inserte en la Lista

Condición de carrera

Condición de carrera (race condition)

Una condición de carrera es cuando el resultado de una operación depende del orden de ejecución de dos o más hebras

- A veces, el orden de ejecución produce resultados correctos o consistentes. Por ejemplo, si una hebra inserta el valor 8 y la otra el valor 5, pueden haber dos resultados correctos:
  - 5, 8
  - 8, 5
- Pero sólo el 8 o sólo el 5 es incorrecto

Desde ahora en adelante, un race condition siempre representa un error (potencial) de concurrencia

Sección crítica

Sección crítica

Una sección crítica (SC) es un trozo de código, ejecutado por múltiples hebras, en el cual se accede a datos compartidos y, al menos una de las hebras escribe sobre los datos

- Si ninguna hebra modifica los datos (write) no hay SC
- Si no hay recursos compartidos, no hay SC
- Si es mono hebra no hay SC
- En el ejemplo anterior, ¿cuál es la SC?

```
void insert(int item) {
  struct list *p;

  p = malloc(sizeof(struct list));
  p->data = item;
  p->next = List;
  List = l;
}
```

Exclusión mutua

Exclusión mutua

Exclusión mutua (EM) es un requerimiento sobre una SC, que dice que sólo una hebra puede estar ejecutando dicha SC

- EM se define por los datos que se acceden en ella, no tanto por el código
- Por ejemplo, en un Stack concurrente, los métodos `pop()` y `push()` son mutuamente exclusivos

```
void pop() {
   if (! empty()}
     top--;
}
```

```
void push(int item) {
    if (! full()) {
       top++;
       data[top] = item
    } else error()
}
```

Modelo de concurrencia

- Asumimos una colección de procesos/tareas del tipo

```
Process(i) {
     while (true) {
         codigo no critico
         ...
         enterCS();
         CS();
         exitCS();
         ...
         codigo no critico
      }
}
```

- `enterCS()` representa las acciones que la hebra debe realizar para poder entrar a su SC
- `exitCS()` representa las acciones que la hebra debe realizar para salir la SC y dejarla habilitada
- El código no crítico es cualquier código donde no se accede a recursos compartidos

Requerimientos para una solución de EM

Requerimientos

Una solución al problema de la EM es correcta cuando satisface:

1. **Exclusión mútua**: Cuando T_i está en su SC, ninguna otra hebra T_j está en su correspondiente SC. Esto significa que la ejecución de operaciones de las secciones críticas no son concurrentes
2. **Sin deadlock**: Deadlock es cuando todas las hebras quedan ejecutando `enterCS()` indefinidamente (ya sea bloqueadas o en busy-waiting), y por lo tanto, ninguna entra.
3. **Sin inanición**: Una hebra entra en inanición si nunca logra entrar a su SC.
4. **Progreso**: Si una hebra ejecuta `enterCS()`, y la SC está libre, se debe permitir que la hebra entre a la SC.

Notas

- Una hebra puede tener más de una SC
- Aunque deadlock implica inanición, inanición no implica deadlock
- No se debe realizar suposiciones sobre la velocidad relativa y orden de ejecución de las hebras

Tipos de soluciones al problema de la SC

Las funciones `enterCS()` y `exitCS()` pueden ser implementadas de varias formas:

1. **Por hardware**: usar instrucciones nativas del procesador
2. **Por software**: algoritmos en lenguaje de alto nivel
3. **Por SO**: usar servicios del SO
4. **Por compilador**: usar estructuras sintácticas del lenguaje que implementan exclusión mutua

A veces, se usa el término **lock** en forma genérica. Es decir

1. `lock.acquire()` es equivalente a `enterCS()`
2. `lock.release()` es equivalente a `exitCS()`

Atomicidad

Atomicidad

- Una operación es atómica si no puede dividirse en partes
- Una operación atómica se ejecuta completamente o no se ejecuta
- Una operación atómica se ejecuta sin ser interrumpida

- Note que atomicidad necesariamente implica exclusión mutua, pero EM no implica atomicidad
- Es decir, si una o varias operaciones se ejecutan atómicamente, entonces también satisfacen el requerimiento de EM
- Pero un conjunto de operaciones que se ejecutan con EM, no necesariamente se ejecutan atómicamente

La ilusión de atomicidad

Considere el siguiente código ejecutado por dos hebras (asuma i=5):

```
inc(int &i) {                LOAD i, ACC   // carga el acumulador
   i = i + 1;                INC  ACC      // incrementa el acumulador
}                            STORE ACC, i  // almacena el resultado en memoria
```

Los posibles resultados son:

- 7, resultado válido
- 6, resultado inválido

Note que toda instrucción assembler es atómica, pero hasta la instrucción más simple de lenguaje de alto nivel puede no ser atómica.

Solución por hardware 1

- **Deshabilitación de interrupciones** : Recordar que una hebra se ejecuta hasta que se bloquea por algún evento (por ejemplo I/O) o es interrumpida, por ejemplo por expiración del quantum. Entonces para garantizar EM, la hebra deshabilita las interrupciones justo antes de entrar en su SC

```
while (true)
  {
       deshabilitar_interrupciones();
       SC();
       habilitar_interrupciones();
  }
```

- Sencillo de entender
- Habilidad del procesador para hacer context-switch queda limitada
- No sirve para multiprocesadores
- Funciona para cualquier número de hebras
- No está libre de inanición

Solución por hardware 2, `testset()`

- **Uso de instrucciones especiales de máquina** que realizan en forma **atómica** más de una operación.
- Por ejemplo `testset()`

```
boolean testset(int &i) {
   if (i == 0) {
      i = 1;
      return true;
   } else return false;
}
```

```
T(i) {
    while (true) {
       ...
       while (!testset(bolt));
       SC();
       bolt = 0;
       ...
     }
}
```

```
int bolt; // shared
void main() {
     bolt = 0;
     parbegin(T(1), T(2),...,T(n));
}
```

EM por instrucciones de máquina

- Ventajas
  - Aplicable a cualquier número de hebras y procesadores que comparten memoria física
  - Simple y fácil de verificar
  - Puede ser usada con varias SC, cada una controlada por su propia variable
- Desventajas
  - *Busy-waiting* (o *spin-lock*) consume tiempo de procesador, innecesariamente
  - Es posible inanición

Soluciones por software 1

```
int turno;
T0 {                                T1 {
    while (true) {                      while (true) {

       while (turno != 0);                 while (turno != 1);
       SC();                               SC();
       turno = 1;                          turno = 0;

     }                                   }
}                                     }

void main() {
    turno = 0;
    parbegin(T0, T1);
}
```

- Garantiza exclusión mútua
- Sólo aplicable a dos procesos
- Alternación estricta de ejecución
- No satisface progreso

Solución de Peterson

```
boolean flag[2];
int turno;

T0 {
   while (true) {
        ...
        flag[0] = true;
        turno = 1;
        while (flag[1] && turno == 1);
        SC();
        flag[0] = false;
        ...
    }
}
void main(){
    flag[0] = flag[1] = false;
    parbegin(T0, T1);
}
```

```
T1 {
      while (true) {
         ...
         flag[1] = true;
         turno = 0;
         while (flag[0] && turno == 0);
         SC();
         flag[1] = false;
         ...
      }
   }
```

- Satisface todos los requerimientos
- Es compleja de entender
- Funciona sólo para dos hebras
- Produce busy-waiting

Algoritmo de la panadería (simplificado)

La solución de software para N hebras utiliza la lógica de una panadería (o farmacia), en la que los clientes son atendidos de acuerdo a un ticket que sacan al momento de llegar

1. Cada hebra T_i tiene un variable entera `num[i]`, inicialmente inicializada en 0. Este es el ticket
2. Todas las hebras pueden leer cualquier `num[j]`, pero solo T_i puede escribir `num[i]`
3. Cuando T_i desea entrar a la SC, setea `num[i]` a un valor más grande que todos los otros `num[j]`. Esto corresponde a sacar el ticket
4. Para cada hebra j, T_i espera hasta que `num[j]` es cero o `num[j]` es mayor que `num[i]`, para todo $j \neq i$. Esto corresponde a la espera del turno
5. Luego que lo anterior se hizo ciero para todas la T_j, T_i entra a la SC
6. Al salir de la SC, T_i setea `num[i]` en cero

Algoritmo de la panadería (simplificado)

```
int num[N] = 0;

1 T(int i) {
2    num[i] = max(num[0],num[1],...,num[N-1]) + 1;
3    for (p=0, p < N; p++) {
4        while (num[p] != 0 && num[p] < num[i]) ;
5    }
6    SC();
7    num[i] = 0;
8 }
```

Para que esta solución sea correcta, se requiere que

1. Las operaciones de lectura y escritura de variables sean atómicas
2. La operación `max()` sea atómica

Algoritmo de la panadería (original)

- Si `max()` no es atómica, dos o más hebras pueden tomar el mismo número. ¿Cómo?
- Para romper la igualdad, se usa el par `(num[i], i)`. Es decir, de todas las hebras que hayan elegido el mismo número, la hebra con ID menor tiene la prioridad
- Aún se requiere atomicidad de read-write de variables enteras y booleanas

```
int num[N] = 0;
boolean choosing[N] = False;

1  T(int i) {
2     choosing[i] = true;
3     num[i] = max(num[0],num[1],...,num[N-1]) + 1;
4     choosing[i] = false;
5     for (p=0, p < N; p++) {
6         while (choosing[p]);
7         while (num[p] != 0 && (num[p],p) < (num[i],i)) ;
8     }
9     SC();
10    num[i] = 0;
11 }
```

Servicios del SO: Semáforos y Locks

- Una semáforo es una variable especial usada para que dos o más tareas se señalicen mútuamente
- Un semáforo es una variable entera, pero
- Accesada sólo por dos operaciones
  1. `wait(s)` decrementa el semáforo; si el valor resultante es negativo, la tarea se bloquea; sino, continúa su ejecución
  2. `signal(s)` incrementa el semáforo; si el valor resultante es menor o igual que cero, entonces se despierta una tarea que fue bloqueado por `wait()`

Primitivas sobre semáforos

```
struct semaphore {
        int count;
        queueType queue;
}
```

```
void wait(semaphore s)
{
   s.count--;
   if (s.count < 0) {
     place this process
          in s.queue;
     block this process
   }
}
```

```
void signal(semaphore s)
{
     s.count++;
     if (s.count <= 0) {
       remove a process P
            from s.queue;
       place process P on ready state
     }
}
```

El SO garantiza que estas operaciones se ejecutan cómo si fueran atómicas

Semáforo binario o mutex o lock

- Un semáforo binario sólo puede tomar los valores 0 o 1
- Un lock se toma y se libera (acquire() y release())

```
wait(s)
{
   if (s.value == 1)
       s.value = 0;
   else {
       place this process
          in s.queue
       block this process
   }
}
```

```
signal(s)
{
    if (s.queue is empty())
       s.value = 1
    else {
       remove a process P
              from s.queue
       place P on the ready list;
    }
}
```

EM usando semáforos

```
semaphore s;
Process(i) {
     while (true) {
         codigo no critico
         ...
         wait(s)
         SC();
         signal(s);
         ..
         codigo no critico
      }
}

void main() {
     s = 1;
     parbegin(P(1), P(2),...,P(n));
}
```

El problema del productor/consumidor

- Uno o más procesos productores generan datos y los almacenan en un buffer compartido
- Un proceso consumidor toma (consume) los datos del buffer uno a la vez
- Claramente, sólo un agente (productor o consumidor) pueden accesar el buffer a la vez

```
producer() {
      while (true) {
          v = produce();
          while ((in + 1) % n == out);

          b[in] = v;
          in = (in + 1) % n;
      }
}
```

```
consumer() {
      while (true) {
          while (in == out);

          w = b[out];
          out = (out + 1) % n;
          consume(w);
 }
```

- El buffer es de largo n y tratado circularmente
- `in` indica la siguiente posición libre y `out` el siguiente elemento a consumir

Solución productor/consumidor con semáforos

```
const int sizeofbuffer;
semaphore s = 1;
semaphore full = 0;
semaphore empty = sizeofbuffer;

producer()
{
     while(true) {
          v = produce();
          wait(empty);
          wait(s);
          b[in] = v;
          in = (in + 1) % n;
          signal(s);
          signal(full);
      }
}
```

```
consumer()
{
     while(true) {
          wait(full);
          wait(s);
          w = b[out];
          out = (out + 1) % n;
          signal(s);
          signal(empty);
          consume(w);
      }
 }
```

Sincronización con hebras POSIX

- Hebras POSIX definen funciones para la sincronización de hebras
  1. *mutexes*: parecidos a los semáforos binarios y sirven para proveer exclusión mútua a cualquier número de hebras de un proceso
  2. *variables de condición*: un mecanismo asociado a los mutexes. Permiten que una hebra se bloquee y libere la SC en la que se encuentra
- Estos mecanismos no sirven para sincronizar hebras de distintos procesos

Mutex

- Un mutex es la forma más básica de sincronización y provee el mismo servicio que un semáforo binario
- Generalmente se utilizan para implementar acceso exclusivo a SC de las hebras

```
#include <pthread.h>

int pthread_mutex_lock(pthread_mutex_t *mutex);
int pthread_mutex_trylock(pthread_mutex_t *mutex);
int pthread_mutex_unlock(pthread_mutex_t *mutex);

int pthread_mutex_init(pthread_mutex_t *mutex, const pthread_mutexattr_t *attr);
                    pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;
int pthread_mutex_destroy(pthread_mutex_t *mutex);
```

- Si una hebra invoca `pthread_mutex_lock(m)` y el mutex `m` ya está tomado, entonces la hebra se bloquea (en ese mutex), sino toma (cierra) el mutex y continúa su ejecución
- Si una hebra invoca `pthread_try_lock(m)` y el mutex `m` ya está tomado, entonces `pthread_try_lock(m)` retorna un código de error `EBUSY`; la hebra no se bloquea y puede continuar su ejecución (fuera de la SC)

Variables de condición

- ¿ Qué sucedería si una hebra se bloquea dentro de una SC?
- Una variable de condición permite que una hebra se bloquee dentro de una SC y al mismo tiempo libere la sección crítica

```
int pthread_cond_wait(pthread_cond_t *cond, pthread_mutex_t * mutex);
int pthread_cond_signal(pthread_cond_t *cond);
int pthread_cond_broadcast(pthread_cond_t *cond);

int pthread_cond_destroy(pthread_cond_t *cond);
int pthread_cond_init(pthread_cond_t *cond, const pthread_condattr_t * atti
                    pthread_cond_t cond = PTHREAD_COND_INITIALIZER;
```

- La hebra que invoca `pthread_cond_wait(c, m)` se bloquea hasta que la condición `c` se *señalice*; además libera el mutex `m`
- `pthread_cond_signal(c)` señaliza o despierta alguna otra hebra que está bloqueada en la variable de condición `c`
- `pthread_cond_broadcast(c)` desbloquea todas las hebras esperando en `c`

Productor/consumidor usando pthreads

```
typedef struct {
     int buf[QUEUESIZE];
     int head, tail;
     int full, empty;
     pthread_mutex_t mutex;
     pthread_cond_t notFull, notEmpty;
} buffer_t;

void main() {
     buffer_t *buffer;
     pthread_t pro, con;

     buffer = bufferInit();
     pthread_mutex_init(&buffer->mutex, NULL);
     pthread_cond_init(&buffer->notFull, NULL);
     pthread_cond_init(&buffer->notEmpty, NULL);

     pthread_create(&pro, NULL, producer, (void *) buffer);
     pthread_create(&con, NULL, consumer, (void *) buffer);

     pthread_join (pro, NULL);
     pthread_join (con, NULL);
     exit 0;
}
```

El Productor

```
void *producer(void *arg)
{
     int v;
     buffer_t *buffer;
     buffer = (buffer_t *) arg;

     while (true) {
        v = produce();
        pthread_mutex_lock (&buffer->mutex);
        while (buffer->full) {
              pthread_cond_wait (&buffer->notFull, &buffer->mutex);
        }
        put_in_buffer(buffer, v);
        pthread_cond_signal(&buffer->notEmpty);
        pthread_mutex_unlock(&buffer->mutex);
     }
}
```

El Consumidor

```
void *consumer (void *arg)
{
     int v;
     buffer_t *buffer;
     buffer = (buffer_t *) arg;

     while (true) {
        pthread_mutex_lock (&buffer->mutex);
        while (buffer->empty) {
            pthread_cond_wait (&buffer->notEmpty, &buffer->mutex);
        }
        v = take_from_buffer(buffer);
        pthread_cond_signal(&buffer->notFull);
        pthread_mutex_unlock(&buffer->mutex);
     }
}
```

Barreras para hebras

- Mecanismo de sincronización para detener (bloquear) la ejecución de un grupo de hebras en un mismo punto del programa
- Cuando la barrera está abajo las hebras se detienen en ella en el orden en que llegan
- La barrera no se levanta hasta que no lleguen todas las hebras
- Cuando la barrera se levanta, las hebras continúan su ejecución
- Las barreras son útiles para implementar *puntos de sincronización* globales en un algoritmo
- Usamos mutex y variables de condición para implementar barreras
- Implementamos las siguientes funciones
  1. `barrier_init()`
  2. `barrier_wait()`
  3. `barrier_destroy()`

Implementación barreras con pthreads

- Una posible forma de implementación es usando dos mutexes y dos variables de condición
- Cada par representa una barrera

```
struct _sb {
     pthread_cond_t  cond;
     pthread_mutex_t mutex;
     int runners; // numero de hebras que aun no llegan a la barrera
};

typedef struct {
     int maxcnt; // numero de hebras participando en la barrera
     struct _sb sb[2];
     struct _sb *sbp;
} barrier_t;
```

Inicialización de barrera

```
int barrier_init(barrier_t *bp, int count)
{
        int i;
        if (count < 1) {
                printf("Error: numero de hebras debe ser mayor que 1\n");
                exit(-1);
        }

        bp->maxcnt = count;
        bp->sbp = &bp->sb[0];

        for (i=0; i < 2; i++) {
             struct _sb *sbp = &(bp->sb[i]);
             sbp->runners = count;
             pthread_mutex_init(&sbp->mutex, NULL);
             pthread_cond_init(&sbp->cond, NULL);
        }
        return(0);
}
```

Espera con barreras

```
int barrier_wait(barrier_t *bp)
{
        struct _sb *sbp = bp->sbp;

        pthread_mutex_lock(&sbp->mutex);
        if (sbp->runners == 1) {
           if (bp->maxcnt != 1) {
                 sbp->runners = bp->maxcnt;
                 bp->sbp = (bp->sbp == &bp->sb[0])? &bp->sb[1] : &bp->sb[0];
                 pthread_cond_broadcast(&sbp->cond);
           }
        } else {
           sbp->runners--;
           while (sbp->runners != bp->maxcnt)
                pthread_cond_wait(&sbp->cond, &sbp->mutex);
        }
        pthread_mutex_unlock(&sbp->wait_mutex);
}
```

Uso de barreras

```
void *proceso(void *arg) {
     int *tidptr, tid;
     tidptr = (int *) arg; tid = *tidptr;
     printf("hola justo antes de la barrera%d\n", tid);
     barrier_wait(&barrera);
}

int main(int argc, char *argv[])
{
     int i, status;
     pthread_t hebra[10];

     barrier_init(&barrera, 10);
     for (i=0; i < 10; i++){
         taskids[i] = (int *) malloc(sizeof(int));
         *taskids[i] = i;
         pthread_create(&hebra[i], NULL, &proceso, taskids[i]);
     }

     for (i=0; i < 10; i++)
          pthread_join(hebra[i], (void **) &status);

     barrier_destroy(&barrera);
}
```