uc3m | Universidad Carlos III de Madrid

Programación con Hilos II

Contenido

- Revisión de la clase anterior
- Condiciones de carrera (*race conditions*)
- Semáforos
- Seguridad en hilos
- Interbloqueo (*deadlock*)

Repaso: Recursos compartidos

- El acceso a recursos compartidos requiere ser controlado para asegurar operaciones deterministas
- Sincronización de datos mediante: mutexes, semáforos, lectura/escritura de candados, barreras
- Mutex: mecanismo de bloqueo / desbloqueo
  - **int** pthread_mutex_init(pthread_mutex_t *mutex, **const** pthread_mutexattr_t * attr);
  - **int** pthread_mutex_destroy(pthread_mutex_t *mutex);
  - **int** pthread_mutex_lock (pthread_mutex_t *mutex);
  - **int** pthread_mutex_trylock (pthread_mutex_t *mutex);
  - **int** pthread_mutex_unlock (pthread_mutex_t *mutex);

Repaso: Variables de Condición

- Bloqueo/desbloqueo (con mutex) basado en el valor en tiempo de ejecución de variables condición.
- Permite al hilo esperar hasta que una condición se convierta en verdadera.
- Otros hilos desbloquean (con signals) a hilos en estado de espera.
  - **int** pthread_cond_init(pthread_cond_t *cond,
    const pthread_condattr_t *attr);
  - **int** pthread_cond_destroy(pthread_cond_t *cond);
  - **int** pthread_cond_wait(pthread_cond_t *cond,
    pthread_mutex_t *mutex);
  - **int** pthread_cond_broadcast(pthread_cond_t *cond);
  - **int** pthread_cond_signal(pthread_cond_t *cond);

Programación multihilos

- OS determina a través del planificador qué hilo se ejecuta y cuándo
- El planificador del SO puede ejecutar los hilos en cualquier orden
- Sin una sincronización adecuada, el código puede ejecutarse de forma no-deterministica
- Supongamos tener dos hilos:
  - 1 lee una variable,
  - 2 modifica esa variable
- El planificador puede ejecutar:
  - 1, luego 2,
  - o 2 y luego 1
- El no-determinismo origina la condición de carrera (*race condition*) – el comportamiento (resultado) depende del orden de ejecución.

Condiciones de carrera

- Condiciones de carrera ocurren cuando varios hilos comparten una variable sin una sincronización adecuada
- La sincronización usa variables especiales como “mutex” para asegurarse el orden de ejecución correcto
- Ejemplo: el hilo T1 requiere hacer algo antes que el hilo T2
  - La variable de condición obliga al hilo T2 a esperar por el hilo T1
  - Modelo de programa productor-consumidor
- Ejemplo: dos hilos necesitan tener acceso a una misma variable y
  - Su modificación se basa en el acceso a través de un “mutex”.
  - Un “mutex” agrupa operaciones para convertirlas en atómicas y tratarlas como una unidad.

Ejemplo

Considere el programa race.c:

```
unsigned int cnt = 0;
void *count ( void *arg ) { /* thread body */
  int i ;
  for ( i = 0; i < 100000000; i ++)
      cnt ++;
  return NULL ;
}
int main ( void ) {
  pthread_t t i d s [ 4 ] ;
  int i ;
  for ( i = 0; i < 4; i ++)
    pthread_create (& t i d s [ i ] , NULL, count , NULL ) ;
  for ( i = 0; i < 4; i ++)
     pthread _join ( t i d s [ i ] , NULL ) ;
  p r i n t f ( " cnt=%u \ n " , cnt ) ;
  return 0;
}
```

¿Cuál es el valor de cnt?

Resultados del ejemplo

Idealmente cnt debería ser incrementada 4 × 100000000 veces, por lo tanto cnt = 400000000.

Sin embargo, una ejecución del programa da:

```
athena% ./race.o
cnt=137131900
athena% ./race.o
cnt=163688698
athena% ./race.o
cnt=163409296
athena% ./race.o
cnt=170865738
athena% ./race.o
cnt=169695163
```

¿Qué ocurrió?

Condiciones de carrera

- C no fue diseñado para permitir concurrencia
- C no tiene operaciones atómicas
- La operación en C: cnt++; se traduce en tres operaciones en lenguaje ensamblador:
  - cargar cnt en un registro
  - incrementar el valor del registro
  - Guardar el nuevo valor en el registro como el nuevo cnt
- ¿Qué sucede si un hilo interrumpe en medio de esas operaciones?

  ¡Condición de carrera!

Condiciones de Carrera

Arreglemos nuestro código:

```
pthread_mutex_t mutex;
unsigned int cnt = 0;

void *count ( void *arg) {/* thread body */
  int i;
  for ( i = 0; i < 100000000; i ++) {
    pthread_mutex_lock(&mutex );
     cnt++;
    pthread_mutex_unlock(&mutex );
  }
  return NULL ;
}
int main ( void ){
  pthread_t tids [4];
  int i;
  pthread_mutex_init(&mutex, NULL);
  for (i =0; i<4; i++)
    pthread_create(&tids[i], NULL, count, NULL);
  for (i =0; i<4; i++)
   pthread _join(tids[i], NULL);
  pthread_mutex_destroy(&mutex );
  printf ("cnt=%u\n " ,cnt );
  return 0;
}
```

Condiciones de Carrera

- El nuevo código funciona correctamente pero es más lento
- Las instrucciones de C no son atómicas, los hilos pueden ser interrumpidos a nivel de ensamblaje, en el medio de una instrucción de C.
- Operaciones atómicas tales como el bloqueo de un “mutex” deben ser especificadas como atómicas utilizando operaciones especiales a nivel de ensamblaje.
- Asegurarse de que todas las instrucciones que accedan/modifican variables compartidas están sincronizadas

Semáforos

- *Semáforo* – variable especial entera y no negativa s, inicialmente con valor 1 que implementa dos operaciones atómicas:
  - P(s) – espera hasta que s> 0, decrementa s y regresa
  - V(s) – incrementa s en 1, desbloquea a un hilo que está bloqueado por s
- Mutex –
  - Llamadas bloqueantes a P(s) y
  - Llamadas desbloqueantes a V(s)
- Implementado en `<semaphore.h>`, parte de la biblioteca `rt`, y no `pthread`

Uso de Semáforos

- Inicializa el semáforo a `value`:

  int sem_init(sem_t ∗sem, **int** pshared, **unsigned int** value);
- Destruye el semáforo:

  int sem_destroy(sem_t ∗sem);
- Espera a bloquear, (es llamada bloqueante):

  int sem_wait(sem_t ∗sem);
- Trata de bloquear, regresa inmediatamente (0 si bloqueado, −1 de otra manera):

  int sem_trywait(sem_t ∗sem);
- Incrementa el semáforo, desbloquea un hilo en espera:

  int sem_post(sem_t ∗sem);

Productor-Consumidor

- Usar semáforos para conocer qué lugares están disponibles en un bufer compartido
- Usar semáforos para seguirle la pista a ítems en un bufer compartido
- Usar un semáforo/mutex para sincronizar las operaciones sobre un bufer

Productor-Consumidor

```c
#include <stdio.h>
#include <pthread.h>
#include <semaphore.h>

sem_t mutex, slots, items;

#define SLOTS 2
#define ITEMS 10

void* produce(void* arg)
{
  int i;
  for (i = 0; i < ITEMS; i++)
  {
    sem_wait(&slots);
    sem_wait(&mutex);
    printf("produced(%ld):%d\n",
           pthread_self(), i+1);
    sem_post(&mutex);
    sem_post(&items);
  }
  return NULL;
}

void* consume(void* arg)
{
  int i;
  for (i = 0; i < ITEMS; i++) {
    sem_wait(&items);
    sem_wait(&mutex);
    printf("consumed(%ld):%d\n",
           pthread_self(), i+1);
    sem_post(&mutex);
    sem_post(&slots);
  }
  return NULL;
}

int main()
{
  pthread_t tcons, tpro;

  sem_init(&mutex, 0, 1);
  sem_init(&slots, 0, SLOTS);
  sem_init(&items, 0, 0);

  pthread_create(&tcons,NULL,consume,NULL);
  pthread_create(&tpro,NULL,produce,NULL);
  pthread_join(tcons,NULL);
  pthread_join(tpro,NULL);

  sem_destroy(&mutex);
  sem_destroy(&slots);
  sem_destroy(&items);
  return 0;
}
```

Otros retos

- Sincronización de objetos ayuda a resolver condiciones de carrera
- Uso impropio puede causar otros problemas
- Algunos asuntos:
    - Seguridad en hilos y funciones reentrantes
    - interbloqueo
    - inanición

Seguridad en hilos

- Una función es "*thread safe" si siempre funciona correctamente cuando es llamada por múltiples y concurrentes hilos.*
- Funciones no-seguras pertenecen a una de estas categorías:
  - accesa/modifica variables compartidas no sincronizadas.
  - Funciones que hacen uso de variables estáticas tales como `rand(), strtok()`
  - Funciones que regresan punteros a memoria estática tales como `gethostbyname()`
  - Funciones que invocan a funciones no seguras

Funciones reentrantes

- Función reentrante– no referencia datos compartidos cuando es utilizada por múltiples hilos
- Todas las funciones reentrantes son thread-safe
- Existen versiones reentrantes de mucha funciones estándar de C no-seguras

Unsafe function	Reentrant version
`rand()`	`rand_r()`
`strtok()`	`strtok_r()`
`asctime()`	`asctime_r()`
`ctime()`	`ctime_r()`
`gethostbyaddr()`	`gethostbyaddr_r()`
`gethostbyname()`	`gethostbyname_r()`
`inet_ntoa()`	(none)
`localtime()`	`localtime_r()`

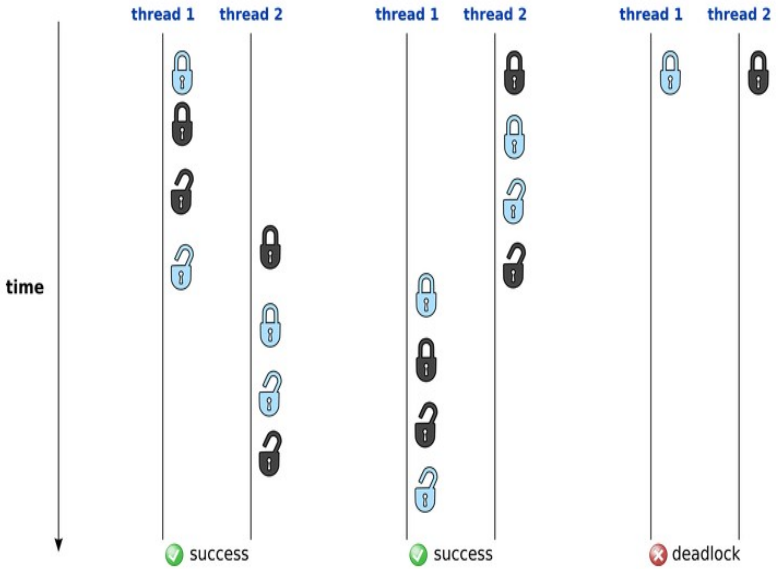
Seguridad en hilos

Para hacer tu código “*thread-safe*”:

- Utiliza objetos de sincronización alrededor de variables compartidas
- Usa funciones reentrantes
- Utiliza la sincronización alrededor de funciones que devuelvan punteros a memoria compartida (*lock-and-copy*):
    1. Bloquea cada función con un mutex
    2. Llamada a función no-segura
    3. Solicita memoria dinámica para resultado; copia el resultado en la memoria asignada dinámicamente
    4. Desbloquea el mutex

Interbloqueo

```
#include <assert.h>
#include <pthread.h>

static void * simple_thread(void *);

pthread_mutex_t mutex_1= PTHREAD_MUTEX_INITIALIZER;
pthread_mutex_t mutex_2= PTHREAD_MUTEX_INITIALIZER;

int main()
{
pthread_t tid = 0;

pthread_create(&tid, 0, &simple_thread, 0); // create a thread

pthread_mutex_lock(&mutex_1);          // acquire mutex_1
pthread_mutex_lock(&mutex_2);          // acquire mutex_2
pthread_mutex_unlock(&mutex_2);        // release mutex_2
pthread_mutex_unlock(&mutex_1);        // release mutex_1

pthread_join(tid, NULL);

return 0;
}

static void * simple_thread(void * dummy)
{
pthread_mutex_lock(&mutex_2);          // acquire mutex_2
pthread_mutex_lock(&mutex_1);          // acquire mutex_1
pthread_mutex_unlock(&mutex_1);        // release mutex_1
pthread_mutex_unlock(&mutex_2);        // release mutex_2

return NULL;
}
```



Interbloqueo

- Interbloqueo – ocurre cuando cada hilo espera por otro hilo para desbloquearse
- Usualmente ocurre por un orden inadecuado de la sincronización
- Es un error difícil de localizar y reproducir pues depende del planificador del SO
- Puede visualizarse utilizando grafos de progreso que trazan el progreso de los hilos en términos de sincronización de objetos

Interbloqueo

- Vencer el interbloqueo es una tarea difícil
- Cuando solo se “mutexes”, puede usarse la “*mutex lockordering rule*” para evitar escenarios interbloqueantes:

  Un programa está libre de interbloqueos si,

  Para cada par de “mutexes (s, t)” en el programa,

  cada hilo que usa s y t simultáneamente

  los bloquea en el mismo orden

Inanición e inversión de prioridades

- La inanición es un problema similar el interbloqueo
  - El planificador nunca asigna los recursos (ejemplo tiempo de CPU) para que un hilo termine su tarea
- Ocurre durante una inversión de prioridades
  - Ejemplo:
    - Hilo de más alta prioridad T1 espera por un hilo de baja prioridad T2 para terminar de usar un recurso.
    - Mientras el hilo T3, que tiene mayor prioridad que T2, se le permite que ejecute indefinidamente
  - Se considera que el hilo T1 está en inanición

uc3m | Universidad Carlos III de Madrid

Gracias por su atención