UNIVERSIDAD NEBRIJA

Grado en Ingeniería Informática

Sistemas Operativos

Práctica 4

Índice

Práctica 4: Variables de condición

Introducción

Las variables de condición proporcionan otro mecanismo para la sincronización de hilos/threads.

Mientras que el uso de mutex permitía sincronizar el acceso a un recurso compartido (variable), las variables de condición permiten la misma sincronización pero en esta ocasión, basada en una condición lógica que suele hacer referencia al estado del recurso (variable) que se esté compartiendo (por ejemplo, condición de buffer lleno o vacío para el problema de productor-consumidor).

Sin variables condicionales los programas tendrían que permanecer en espera-ocupada (hacer "polling" sobre datos) continuamente produciendo un deterioro lógico del rendimiento.

Las variables de condición siempre van asociadas a un mutex y esta solución combinada equivale al mecanismo de semáforos que también evitan una espera activa.

Variables de condición

Las variables de condición proporcionan un mecanismo general que combina la suspensión de hilos con los mutex.

Son un tipo abstracto de datos con tres operaciones básicas: ESPERA, AVISO_SIMPLE y AVISO_MÚLTIPLE.

El flujo de trabajo de este mecanismo hace que en un momento determinado, un hilo pruebe una condición y llame a ESPERA si la condición es falsa. Cuando otro hilo modifique la variable que pudiera hacer que se cumpla la condición, activa al hilo bloqueado invocando a AVISO.

Declaración e inicialización variables de condición:

Las variables de condición son de tipo: **pthread_cond_t**

Existen dos maneras de inicializar las variables de condición:

1. Estática, cuando se define:
   pthread_cond_t myconvar = PTHREAD_COND_INITIALIZER;

2. Dinámica, con la función **pthread_cond_init (condition,attr)**
   a. condition: variable de condición declarada previamente.
   b. attr = NULL

pthread_cond_destroy() elimina una variable de condición que no va a ser usada.

Espera y señalización de variables de condición

pthread_cond_wait (condition,mutex) - bloquea al thread que lo llama hasta que se cumple la condición y es señalizada.

- Debe llamarse mientras mutex está bloqueado y se liberará automáticamente el mutex para ponerse en estado de espera (no polling).
- Cuando sea despertado porque sea posible cumplir con la condición, el mutex se bloqueará automáticamente para ser usado por el hilo.
- Por último es responsabilidad del programador desbloquear el mutex cuando el hilo termine su ejecución.

pthread_cond_signal (condition) – Envía una señal (despierta) al thread que espera por la condición.

- Debe llamarse después de bloquear el mutex .
- Debe desbloquear el mutex para que pthread_cond_wait() pueda completarse.

pthread_cond_broadcast (condition) – Igual signal pero cuando hay más de un thread esperando por la condición.

```
hilo1:

   cierre(mutex);
   ...
   si (condicion) entonces
      espera(cond, mutex);
   ...
   /* Acceso a variables comunes */
   apertura(mutex);
   ...
```

```
hilo2:

   cierre(mutex);
   ...
   /* modificacion variables
   involucradas en condicion */
   ...
   aviso_simple(cond);
   apertura(mutex);
   ...
```

Ejemplo variables de condición

El programa principal crea tres threads:

- Dos de los threads realizan el trabajo y actualización de la variable "count".
- El tercer thread espera a que la variable de condición alcance un determinado valor.

```
#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>

#define NUM_THREADS  3
#define TCOUNT 10
#define COUNT_LIMIT 12

int     count = 0;
pthread_mutex_t count_mutex;
pthread_cond_t count_threshold_cv;

void *inc_count(void *t)
{
  int i;
  long my_id = (long)t;

  for (i=0; i < TCOUNT; i++) {
    pthread_mutex_lock(&count_mutex);
    count++;

    /*
    Check the value of count and signal waiting thread when condition is
    reached.  Note that this occurs while mutex is locked.
    */
    if (count == COUNT_LIMIT) {
      printf("inc_count(): thread %ld, count = %d  Threshold reached. ",
             my_id, count);
      pthread_cond_signal(&count_threshold_cv);
      printf("Just sent signal.\n");
      }
    printf("inc_count(): thread %ld, count = %d, unlocking mutex\n",
               my_id, count);
    pthread_mutex_unlock(&count_mutex);

    /* Do some work so threads can alternate on mutex lock */
    sleep(1);
    }
  pthread_exit(NULL);
}

void *watch_count(void *t)
{
  long my_id = (long)t;

  printf("Starting watch_count(): thread %ld\n", my_id);

  /*
  Lock mutex and wait for signal.  Note that the pthread_cond_wait routine
  will automatically and atomically unlock mutex while it waits.
  Also, note that if COUNT_LIMIT is reached before this routine is run by
  the waiting thread, the loop will be skipped to prevent pthread_cond_wait
  from never returning.
  */
  pthread_mutex_lock(&count_mutex);
  while (count < COUNT_LIMIT) {
    printf("watch_count(): thread %ld Count= %d. Going into wait...\n", my_id,count);
    pthread_cond_wait(&count_threshold_cv, &count_mutex);
    printf("watch_count(): thread %ld Condition signal received. Count= %d\n", my_id,count);
    }
  printf("watch_count(): thread %ld Updating the value of count...\n", my_id);
  count += 125;
  printf("watch_count(): thread %ld count now = %d.\n", my_id, count);
  printf("watch_count(): thread %ld Unlocking mutex.\n", my_id);
  pthread_mutex_unlock(&count_mutex);
  pthread_exit(NULL);
}

int main(int argc, char *argv[])
{
  int i, rc;
  long t1=1, t2=2, t3=3;
  pthread_t threads[3];
  pthread_attr_t attr;

  /* Initialize mutex and condition variable objects */
  pthread_mutex_init(&count_mutex, NULL);
  pthread_cond_init (&count_threshold_cv, NULL);

  /* For portability, explicitly create threads in a joinable state */
  pthread_attr_init(&attr);
  pthread_attr_setdetachstate(&attr, PTHREAD_CREATE_JOINABLE);
  pthread_create(&threads[0], &attr, watch_count, (void *)t1);
  pthread_create(&threads[1], &attr, inc_count, (void *)t2);
  pthread_create(&threads[2], &attr, inc_count, (void *)t3);

  /* Wait for all threads to complete */
  for (i = 0; i < NUM_THREADS; i++) {
    pthread_join(threads[i], NULL);
  }
  printf ("Main(): Waited and joined with %d threads. Final value of count = %d. Done.\n",
          NUM_THREADS, count);

  /* Clean up and exit */
  pthread_attr_destroy(&attr);
  pthread_mutex_destroy(&count_mutex);
  pthread_cond_destroy(&count_threshold_cv);
  pthread_exit (NULL);

}
```

Salida:

```
Starting watch_count(): thread 1
inc_count(): thread 2, count = 1, unlocking mutex
inc_count(): thread 3, count = 2, unlocking mutex
watch_count(): thread 1 going into wait...
inc_count(): thread 3, count = 3, unlocking mutex
inc_count(): thread 2, count = 4, unlocking mutex
inc_count(): thread 3, count = 5, unlocking mutex
inc_count(): thread 2, count = 6, unlocking mutex
inc_count(): thread 3, count = 7, unlocking mutex
inc_count(): thread 2, count = 8, unlocking mutex
inc_count(): thread 3, count = 9, unlocking mutex
inc_count(): thread 2, count = 10, unlocking mutex
inc_count(): thread 3, count = 11, unlocking mutex
inc_count(): thread 2, count = 12  Threshold reached. Just sent signal.
inc_count(): thread 2, count = 12, unlocking mutex
watch_count(): thread 1 Condition signal received.
watch_count(): thread 1 count now = 137.
inc_count(): thread 3, count = 138, unlocking mutex
inc_count(): thread 2, count = 139, unlocking mutex
inc_count(): thread 3, count = 140, unlocking mutex
inc_count(): thread 2, count = 141, unlocking mutex
inc_count(): thread 3, count = 142, unlocking mutex
inc_count(): thread 2, count = 143, unlocking mutex
inc_count(): thread 3, count = 144, unlocking mutex
inc_count(): thread 2, count = 145, unlocking mutex
Main(): Waited on 3 threads. Final value of count = 145. Done.
```

Entregable:

1. Implementar el problema de productor consumidor utilizando variables condicionales.