POSIX Threads Part II

Multikern-Praktikum Wintersemester 06-07
Agenda

- Basics
- Thread Management
- Pthreads Mutex Variables
- Condition Variables
Condition Variables

• another way for threads to synchronize
• allow threads to synchronize based upon the actual value of data (compare to mutexes that implement synchronization by controlling thread access to data)

• Without condition variables
  • continually polling (spinnlock)
  • very resource consuming
Condition Variables and Semaphores

- Semaphores are conditional on the semaphore count (zero or non-zero)
- Condition variable is associated with an arbitrary conditional
  - Same operations: wait and signal
- Provides mutual exclusion
  - wait on the condition variable until signaled
  - Once signaled and woken up,
    - if the conditional expression evaluates correctly, a thread will proceed;
    - otherwise, the thread should be directed to return to waiting on the condition variable.
Condition Variable and Mutex

- Mutex is associated with condition variable
  - Protects evaluation of the conditional expression
  - Prevents race condition between signaling thread and threads waiting on condition variable
# Using condition variables

## Main Thread
- Declare and initialize global data/variables which require synchronization (such as "count")
- Declare and initialize a condition variable object
- Declare and initialize an associated mutex
- Create threads A and B to do work

## Thread A
- Do work until a certain condition must occur (such as "count" must reach a specified value)
- Lock associated mutex and check value of a global variable
- Call `pthread_cond_wait()` to perform a blocking wait for signal from Thread-B. Note that a call to `pthread_cond_wait()` automatically and atomically unlocks the associated mutex variable so that it can be used by Thread-B.
- When signalled, wake up. Mutex is automatically and atomically locked.
- Explicitly unlock mutex
- Continue

## Thread B
- Do work
- Lock associated mutex
- Change the value of the global variable that Thread-A is waiting upon.
- Check value of the global Thread-A wait variable. If it fulfills the desired condition, signal Thread-A.
- Unlock mutex.
- Continue

## Main Thread
- Join / Continue
Lost and Spurious Signals

• Signal to condition variable is not saved
  • If no thread waiting, signal is “lost”
  • Thread can be deadlocked waiting for signal that will not be sent

• Condition variable can (rarely) receive spurious signals
  • from predictable signals
  • Need to retest conditional expression
Condition Variable Algorithm

- Avoids problems with lost and spurious signals

acquire mutex;
while (conditional is true)
  wait on condition variable;
perform critical region computation;
update conditional;
signal sleeping thread(s);
release mutex;

Negation of condition needed to proceed

Mutex is automatically released when thread waits

Could be optional
Condition Variable Types

- Data types used
  - `pthread_cond_t`
    - the condition variable
  - `pthread_condattr_t`
    - condition variable attributes
    - only one attribute: process-shared,
      - allows the condition variable to be seen by threads in other processes.(not all Pthread implementations may provide the process-shared attribute)
    - may be specified as NULL to accept defaults

- Before use, condition variable (and related mutex) must be initialized
Condition Variables

- `pthread_cond_init, pthread_cond_destroy`
  - initialize/destroy condition variable

- `pthread_cond_wait`
  - thread goes to sleep until signal of condition variable

- `pthread_cond_signal`
  - signal release of condition variable

- `pthread_cond_broadcast`
  - broadcast release of condition variable
Dynamic Initialization

- `int pthread_cond_init( cond, attr );`

```c
pthread_cond_t *cond
  • condition variable to be initialized

const pthread_condattr_t *attr
  • attributes to be given to condition variable
    (only one attribute: process-shared)
```

- ENOMEM - insufficient memory for condition variable
- EAGAIN - insufficient resources (other than memory)
- EBUSY  - condition variable already initialized
- EINVAL - attr is invalid
Alternate Initialization

• Can also use the static initializer

```c
PTHREAD_COND_INITIALIZER
pthread_cond_t cond1 = PTHREAD_COND_INITIALIZER;
```

• Uses default attributes

• Programmer must always pay attention to condition (and mutex) scope
  • Must be visible to threads
**pthread_cond_wait**

- `int pthread_cond_wait( cond, mutex );`

  `pthread_cond_t *cond`
  - condition variable to wait on

  `pthread_mutex_t *mutex`
  - mutex to be unlocked

- blocks the calling thread until the specified condition is signalled.
- should be called while `mutex` is locked, and it will automatically release the mutex while it waits
- After signal is received and thread is awakened, `mutex` will be automatically locked for use by the thread.
- The programmer is then responsible for unlocking `mutex` when the thread is finished with it.
pthread_cond_wait

• Returned error Code:

- EINVAL  - cond or mutex is invalid
- EINVAL  - different mutex for concurrent waits
- EINVAL  - calling thread does not own mutex
**pthread_cond_signal**

- `int pthread_cond_signal( cond );`

  **pthread_cond_t** *cond
  - condition variable to be signaled

  - Signal condition variable, wake one waiting thread
  - It should be called after `mutex` is locked, and must unlock `mutex` in order for `pthread_cond_wait()` routine to complete.
  - If no threads waiting, no action taken
    - Signal is not saved for future threads

  **EINVAL** - cond is invalid
**pthread_cond_signal**

- Proper locking and unlocking of the associated mutex variable is essential
  - Failing to lock the mutex before calling `pthread_cond_wait()` may cause it **NOT** to block
  - Failing to unlock the mutex after calling `pthread_cond_signal()` may not allow a matching `pthread_cond_wait()` routine to complete (remain blocked)
`pthread_cond_broadcast`

- `int pthread_cond_broadcast( cond );`

`pthread_cond_t *cond`
- condition variable to signal
pthreads_cond_broadcast

- Wake all threads waiting on condition variable
- If no threads waiting, no action taken
  - Broadcast is not saved for future threads

EINVAL - cond is invalid
# Example: Condition Variables

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

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

int     count = 0;int     thread_ids[3] = {0, 1, 2};
pthread_mutex_t count_mutex;
pthread_cond_t count_threshold_cv;

void *inc_count(void *idp);
void *watch_count(void *idp);

int main (int argc, char *argv[]){
    int i, rc;
    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);
    ...
Example: Condition Variables

/* explicitly create threads in a joinable state (portable) */
pthread_attr_init(&attr);
pthread_attr_setdetachstate(&attr, PTHREAD_CREATE_JOINABLE);

pthread_create(&threads[0], &attr, inc_count, (void *)
&thread_ids[0]);
pthread_create(&threads[1], &attr, inc_count, (void *)
&thread_ids[1]);
pthread_create(&threads[2], &attr, watch_count, (void *)
&thread_ids[2]);

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

/* Clean up and exit */
pthread_attr_destroy(&attr);
pthread_mutex_destroy(&count_mutex);
pthread_cond_destroy(&count_threshold_cv);
pthread_exit(NULL);
} /* End of main */
Example: Condition Variables

```c
void *inc_count(void *idp) {
    int j,i;
    double result=0.0;
    int *my_id = idp;

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

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

        /* Do some work so threads can alternate on mutex lock */
        for (j=0; j<1000; j++)
            result = result + (double)random();
    }
    pthread_exit(NULL);
}
```
Example: Condition Variables

```c
void *watch_count(void *idp) {
    int *my_id = idp;

    printf("Starting watch_count(): thread %d\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.*/
    pthread_mutex_lock(&count_mutex);
    while (count<COUNT_LIMIT) {
        pthread_cond_wait(&count_threshold_cv, &count_mutex);
        printf("watch_count(): thread %d Condition signal
            received.\n", *my_id);
    }
    pthread_mutex_unlock(&count_mutex);
    pthread_exit(NULL);
}
```
Example: Condition Variables
(output)

inc_count(): thread 0, count = 1, unlocking mutex
Starting watch_count(): thread 2
inc_count(): thread 1, count = 2, unlocking mutex
inc_count(): thread 0, count = 3, unlocking mutex
inc_count(): thread 1, count = 4, unlocking mutex
inc_count(): thread 0, count = 5, unlocking mutex
inc_count(): thread 0, count = 6, unlocking mutex
inc_count(): thread 1, count = 7, unlocking mutex
inc_count(): thread 0, count = 8, unlocking mutex
inc_count(): thread 1, count = 9, unlocking mutex
inc_count(): thread 0, count = 10, unlocking mutex
inc_count(): thread 1, count = 11, unlocking mutex
inc_count(): thread 0, count = 12 Threshold reached.
inc_count(): thread 0, count = 12, unlocking mutex
watch_count(): thread 2 Condition signal received.
inc_count(): thread 1, count = 13, unlocking mutex
inc_count(): thread 0, count = 14, unlocking mutex
inc_count(): thread 1, count = 15, unlocking mutex
inc_count(): thread 0, count = 16, unlocking mutex
inc_count(): thread 1, count = 17, unlocking mutex
inc_count(): thread 0, count = 18, unlocking mutex
inc_count(): thread 1, count = 19, unlocking mutex
inc_count(): thread 1, count = 20, unlocking mutex
Main(): Waited on 3 threads. Done.
• Some online References:
  • http://yolinux.com/TUTORIALS/LinuxTutorialPosixThreads.html
  • http://users.actcom.co.il/~choo/lupg/tutorials/multi-thread/multi-thread
  • http://math.arizona.edu/~swig/documentation/pthreads/
  • http://www.linuxselfhelp.com/HOWTO/C++Programming-HOWTO-18
  • http://www.cs.nmsu.edu/~jcook/Tools/pthreads/pthreads.html