POSIX THREADS

Ken Wong
Washington University, St. Louis
kenw@arl.wustl.edu
www.arl.wustl.edu/~kenw

POSIX SYNCHRONIZATION PRIMITIVES

- Each synchronization facility has a named data structure called a synchronization variable.
- Counting Semaphores
  - Typically used to coordinate access to resources (e.g., shared variable)
- Mutual Exclusion (mutex) Locks
  - Used to serialize the execution of code
- Condition Variables
  - Enables threads to atomically block until a condition is satisfied
- Multiple Readers, Single Writer Locks
  - Allows many threads to have simultaneous read-only access to data while allowing only one thread to have write access at any given time

THREADS STANDARDS

- Defines an API and behavior of a threads paradigm
  - About 50 function calls
- POSIX Threads
  - IEEE 1003.1c (Pthreads)
  - Portable
  - Implementations on almost all Unix systems
  - Not adopted by Microsoft
- Win32 and OS/2 Threads
  - Not compatible with Pthreads
  - Proprietary (vendor-specific)
- Solaris Threads (UI Threads)
  - Used in Solaris 2 and developed before Pthreads standard was finalized
  - Virtually the same as Pthreads

SCALAR ADDITION ALGORITHM 1

- adder i, m threads
  - Computes the partial sum x[i] , j = i, i + m, i + 2m, i + 3m, ...
  - Sigan's main routine when done
- main sums the partial sums when it has received \# DONE signals.
EXAMPLE 1 (Definitions)

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

#define N 16 // max #threads
#define M 10000 // max n

void *adder (void *id);

int x[M], n, nthreads;
int psum[N];
sem_t work_done; // semaphore used for signaling done
```

EXAMPLE 1 (main)

```c
main (int argc, char *argv[]) {
    int i, n, nthreads;

    nthreads = 2;
    n = 1001;
    for (i=0; i<n; i++) x[i] = i;

    sem_init (&work_done, 0, 0); // counter = 0
    for (i=0; i<nthreads; i++) {
        who[i] = i;
        pthread_create (&(tid[i]), NULL, adder, &(who[i]));
    }
    for (i=0; i<nthreads; i++) sem_wait (&work_done);
    sum = 0;
    for (i=0; i<nthreads; i++) { sum += psum[i]; }
}
```

EXAMPLE 1 (adder)

```c
void *adder (void *id) {
    int whoami;
    int i;
    int nadds;

    whoami = *((int *) id);

    psum[whoami] = 0;
    for (i = whoami; i<n; i += nthreads)
        psum[whoami] += x[i];

    sem_post (&work_done);
}
```

SCALAR ADDITION ALGORITHM 2

- Each adder
  - Computes a partial sum
  - Adds its partial sum to the grand sum
  - Decrement the ndone counter which is initially equal to N.
  - If counter is 0, it signals the main routine.
- main waits for the last adder to signal that it is done.

```
\[
\begin{align*}
\text{x[...]} & \quad \text{adder} \quad \text{DONE} \\
\ldots & \quad \text{sum, ndone} \quad \text{main} \quad \text{sum} \\
\text{x[...]} & \quad \text{adder} \\
\end{align*}
\]
```
EXAMPLE 2 (main)

```c
int x[M] n, nthreads;
struct {
    int sum;     // grand sum
    int ndone;   // number of adder threads finished
    sema_t update_sum; // protect access to sum and ndone
} work;
sema_t all_done;    // semaphore: signal all done
main (int argc, char *argv[]) {
    ... etc ...
    sem_init (&all_done, 0, 0);     // count = 0
    sem_init (&work.update_sum, 0, 1); // count = 1
    work.ndone = 0;  work.sum = 0;
    for (i=0; i<nthreads; i++) id[i] = i;
    for (i=0; i<nthreads; i++)
        pthread_create (&(tid[i]), NULL, adder, &(who[i]));
    sem_wait (&all_done);    ... etc ...
}
```

EXAMPLE 2 (adder)

```c
void *adder (void *id) {
    int whoami;
    int i;
    int nadds;

    whoami = *((int *) id);

    psum[whoami] = 0;
    for (i = whoami; i<n; i += nthreads)
        psum[whoami] += x[i];

    sem_wait (&work.update_sum);    // protect critical region
    work.sum += psum;
    work.ndone++;
    if (ndone == nthreads) sem_post (&all_done);
    sem_post (&work.update_sum);
}
```

EXAMPLE 3 (Main)

```c
#include <pthread.h>

pthread_mutex_t lock;    // protect sum and ndone
pthread_cond_t doneCond; // signal when thread is done
int ndone = 0;  // number of threads done computing
int sum = 0;
main () ... {
    ... declarations and other initialization ... 
    for (i = 0; i < N; i++)
        pthread_create (&(tid[i]), NULL, adder, &(who[i]));
    pthread_mutex_lock (&lock);
    while (ndone < N)     pthread_cond_wait (&doneCond, &lock);
    pthread_mutex_unlock (&lock);
    ... 
}
```

EXAMPLE 3 (Adder)

```c
void adder (void *id) ... {
    int whoami;
    int i;
    int nadds;

    whoami = *((int *) id);

    psum[whoami] = 0;
    for (i = whoami; i<n; i += nthreads)
        psum[whoami] += x[i];

    pthread_mutex_lock (&lock);
    sum += psum;
    ++ndone;
    if (ndone == N) pthread_cond_signal (&doneCond);
    pthread_mutex_unlock (&lock);
}
```
**MUTEX LOCKS**

- int pthread_mutex_lock (pthread_mutex_t *M)
  
  * Acquire lock or block calling thread if the lock is already held

- int pthread_mutex_unlock (pthread_mutex_t *M)
  
  * Release lock and allow next thread to enter critical region

- int pthread_mutex_init (pthread_mutex_t *M,
  
  const pthread_mutexattr_t *Attr);
  
  * Return code is 0 if successful, non-zero otherwise.

  * Not needed if mutex variable is statically allocated; e.g.,
  
  pthread_mutex_t lock; // pthread_mutex_t is a macro

**MUTEX LOCK IMPLEMENTATION**

```c
pthread_mutex_lock (L) {
    while (test&set(L)) { // Someone else has the lock
        Put thread on wait queue for L;
        Suspend thread;
    }
    return;
}
```

```c
pthread_mutex_unlock (L) {
    Unsuspend the next thread in the wait queue for L;
    L = 0;
    return;
}
```

- **Variation**: Spin for a short time instead of suspending in hopes of short blocking time.

**SPIN LOCKS**

- Blocking on a mutex lock will cause two context switches (switch out, switch in)

  * 150 µs on SC2000/Solaris 2.4
  
  * 25 µs on 300 MHz Pentium II/NetBSD
  
  * 35 µs on 167 MHz SPARC 5/Solaris 2.5

- A spin lock can be used to avoid the context switching but wastes CPU time

  ```
  while (pthread_mutex_trylock (&mylock) == EBUSY)
    // ... do nothing ... ;
  ... Critical Section ...
  pthread_mutex_unlock (&mylock);
  ```

  * mutex_trylock() returns 0 when the lock is acquired.

- **Disadvantage**: Wasted CPU cycles

- A faster version than the above spin lock exists

**ADVANTAGE OVER SEMAPHORE**

- Uses little memory and is fast.

<table>
<thead>
<tr>
<th>Type of Synchronization</th>
<th>Time (µsec) on 20 Proc. 40 MHz SPARC</th>
</tr>
</thead>
<tbody>
<tr>
<td>Unbound semaphore</td>
<td>292.0</td>
</tr>
<tr>
<td>Bound semaphore</td>
<td>326.0</td>
</tr>
<tr>
<td>Unbound mutex</td>
<td>2.1</td>
</tr>
<tr>
<td>Bound mutex</td>
<td>2.3</td>
</tr>
</tbody>
</table>
CONDITION VARIABLES

- `int pthread_cond_wait (pthread_cond_t *Cv, pthread_mutex_t *M);`
  - Block until the condition is signaled
  - Atomically release mutex lock before blocking and atomically reacquire it before returning
- `int pthread_cond_signal (pthread_cond_t *Cv);`
  - Unblock one thread waiting for the condition.
  - No thread blocked on Cv ⇒ no effect
  - Call under the protection of the mutex associated with Cv
    - Condition should be retested after a thread becomes unblocked.
- `int pthread_cond_init (pthread_cond_t *Cv, const pthread_condattr_t *Attr);`
  - Return code is 0 if successful, non-zero otherwise.
  - Cv: a condition variable pointer
  - Not needed if mutex variable is statically allocated; e.g.,

CONDITION VARIABLE IMPLEMENTATION

```c
pthread_cond_wait (V, L) {
    do {
        Put thread on wait queue for V;
        L = 0;
        Suspend this thread;
    } until (test&set(L) == 0) { // Until I get the lock
        return;
    }
}

pthread_cond_signal (V, L) {
    Unsuspend next thread in wait queue for V; // Could be none
    return;
}
```

CONDITION VARIABLES ARE STATELESS SIGNALS

- **Example** (Another thread on the mylock queue changes n!)

Thread 0:  
```
mutex_lock(&mylock);
if (n == 0)
    cond_signal(&done, &mylock); --> cond_wait(&done, &mylock);
mutex_unlock(&mylock);
```

Thread 1:  
```
mutex_lock(&mylock);
if (n <= 0)
    do {
        cond_signal(&done, &mylock);  
        while (n > 0);
    }
mutex_unlock(&mylock);
```

- Guarantee the condition has not changed

Thread 0:  
```
mutex_lock(&mylock);
if (n <= 0)
    do {
        cond_signal(&done, &mylock);  
        cond_wait(&done, &mylock);
    }
mutex_unlock(&mylock);
```

Thread 1:  
```
mutex_lock(&mylock);
if (n == 0)
```

- Some other thread can change n before Thread 1 gets control.
- n is not protected from change (unlike the counter of a semaphore)