Condition Variables

Jinkyu Jeong (jinkyu@skku.edu)
Computer Systems Laboratory
Sungkyunkwan University
http://csl.skku.edu
Condition Variables

• Provide a mechanism to wait for events
  – A condition variable (CV) is an explicit queue
  – Threads can put themselves on CV when some state of execution is not met

• Used with mutexes
  – A mutex is a sleeping lock; threads are blocked when it is held by another thread
  – A mutex ensures mutual exclusion for a critical section
  – Manipulating some condition related to a CV should be done inside the critical section
CV Operations

• `wait(cond_t *cv, mutex_t *mutex)`
  – Assumes mutex is held when `wait()` is called
  – Puts the caller to sleep and releases mutex (atomically)
  – When awoken, reacquires mutex before returning
• `signal(cond_t *cv)`
  – Wakes a single thread if there are threads waiting on `cv`
  – Unlike semaphores, CVs have no history; `signal()` is lost if there is no thread waiting for it
• `broadcast(cond_t *cv)`
  – Wakes all waiting threads
  – If there are no waiting thread, just return doing nothing
Pthreads Interface

- Mutexes and CVs are supported in Pthreads

```c
pthread_mutex_t m = PTHREAD_MUTEX_INITIALIZER;
pthread_cond_t c = PTHREAD_COND_INITIALIZER;

void wait_example() {
    pthread_mutex_lock (&m);
    pthread_cond_wait (&c, &m);
    pthread_mutex_unlock (&m);
}

void signal_example() {
    pthread_mutex_lock (&m);
    pthread_cond_signal (&c);
    pthread_mutex_unlock (&m);
}
```
Joining Threads (1)

• An initial attempt

```c
mutex_t m = MUTEX_INITIALIZER;
cond_t c = COND_INITIALIZER;

void *child(void *arg) {
    thr_exit();
    return NULL;
}

void thr_exit() {
    mutex_lock(&m);
    cond_signal(&c);
    mutex_unlock(&m);
}

void thr_join() {
    mutex_lock(&m);
    cond_wait(&c, &m);
    mutex_unlock(&m);
}

t main(int argc, char *argv[]) {
    pthread_t p;
    pthread_create(&p, NULL,
child, NULL);
    thr_join();
    return 0;
}
```
Joining Threads (2)

- Second attempt: keep state in addition to CVs

```c
mutex_t m = MUTEX_INITIALIZER;
cond_t c = COND_INITIALIZER;
int done = 0;

void *child(void *arg) {
    thr_exit();
    return NULL;
}

void thr_exit() {
    done = 1;
    cond_signal(&c);
}

void thr_join() {
    mutex_lock(&m);
    if (done == 0)
        cond_wait(&c, &m);
    mutex_unlock(&m);
    thr_join();
    return 0;
}
```
Joining Threads (3)

• Third attempt: always hold mutex while signaling

```c
mutex_t m = MUTEX_INITIALIZER;
cond_t c = COND_INITIALIZER;
int done = 0;

void *child(void *arg) {
    thr_exit();
    return NULL;
}

int main(int argc, char *argv[]) {
    pthread_t p;
    pthread_create(&p, NULL,
                   child, NULL);
    thr_join();
    return 0;
}

void thr_exit() {
    mutex_lock(&m);
    done = 1;
    cond_signal(&c);
    mutex_unlock(&m);
}

void thr_join() {
    mutex_lock(&m);
    if (done == 0)
        cond_wait(&c, &m);
    mutex_unlock(&m);
}
```
CV Semantics

• **Mesa semantics (used in Pthreads)**
  - `signal()` places a waiter on the ready queue, but signaler continues inside the critical section
  - Condition is not necessarily true when waiter runs again
  - Being woken up is only a hint that something has changed
  - Must recheck the condition

• **Hoare semantics**
  - `signal()` immediately switches from the caller to a waiting thread, blocking the caller
  - The condition that the waiter was anticipating is guaranteed to hold when waiter executes
Bounded Buffer Problem

• What if we use “if” instead of “while”?

```c
mutex_t m;
cond_t notfull, notempty;
int in, out, count;

void produce(data) {
    mutex_lock(&m);
    while (count == N)
        cond_wait(&not_full, &m);
    buffer[in] = data;
in = (in+1) % N;
count++;
    cond_signal(&not_empty);
    mutex_unlock(&m);
}

void consume(data) {
    mutex_lock(&m);
    while (count == 0)
        cond_wait(&not_empty, &m);
data = buffer[out];
out = (out+1) % N;
count--;
    cond_signal(&not_full);
    mutex_unlock(&m);
}
```
Using Broadcast

• Covering condition: when the signaler has no idea on which thread should be woken up
• e.g. memory allocation:

```c
mutex_t m;
cond_t c;
int bytesLeft = MAX_HEAP_SIZE;

void free(void *p, int size) {
    mutex_lock(&m);
    bytesLeft += size;
    cond_broadcast(&c);
    mutex_unlock(&m);
}

void *allocate (int size) {
    mutex_lock(&m);
    while (bytesLeft < size)
        cond_wait(&c, &m);
    void *ptr = ...;
    bytesLeft -= size;
    mutex_unlock(&m);
    return ptr;
}
```
Semaphores vs. Mutexes + CVs

• Both have same expressive power
• Implementing semaphores using mutexes and CVs:

```c
typedef struct sema_t {
    int v;
    cond_t c;
    mutex_t m;
} sema_t;

void sema_init (sema_t *s, int v) {
    s->v = v;
    cond_init(&s->c);
    mutex_init(&s->m);
}

void sema_wait (sema_t *s) {
    mutex_lock(&m);
    while (s->v <= 0)
        cond_wait(&s->c, &s->m);
    s->v--;
    mutex_unlock(&m);
}

void sema_signal (sema_t *s) {
    mutex_lock(&m);
    s->v++;
    cond_signal (&s->c);
    mutex_unlock(&m);
}
```
Summary

• Disabling interrupts
  – Only for the kernel on a single CPU

• Spinlocks
  – Busy waiting, implemented using atomic instructions

• Semaphores
  – Binary semaphore = mutex (≡ lock)
  – Counting semaphore

• Mutexes + condition variables
  – Used in Pthreads