

Jin-Soo Kim
(jinsoo.kim@snu.ac.kr)

Systems Software &
Architecture Lab.

Seoul National University

Fall 2023

Condition Variables and Mutexes



Condition Variables and Mutex

- Yet another synchronization construct
- Condition variables can be also used without monitors in conjunction with mutexes
- Think of a monitor as a language feature
 - Under the covers, compiler knows about monitors
 - Compiler inserts a mutex to control entry and exit of processes to the monitor's procedures
 - But can be done anywhere in procedure, at finer granularity
- With condition variables, the module methods may wait and signal on independent conditions

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 _____ 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, `signal()` is lost if there is no thread waiting for it
 - _____ semantics: thread continues after sending `signal()`
- `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

```
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: An Initial Attempt

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

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

int main(int argc, char *argv[]) {
    thread_t p;
    thread_create(&p, NULL, child, NULL);
    thread_join();
    return 0;
}
```

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

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

Joining Threads: Second Attempt

- Keep state in addition to CVs

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

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

int main(int argc, char *argv[]) {
    thread_t p;
    thread_create(&p, NULL, child, NULL);
    thread_join();
    return 0;
}
```

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

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

Joining Threads: Third Attempt

- Always hold mutex while signaling

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

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

int main(int argc, char *argv[]) {
    thread_t p;
    thread_create(&p, NULL, child, NULL);
    thread_join();
    return 0;
}
```

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

void thread_join() {
    mutex_lock(&m);
    while (done == 0)
        cond_wait(&c, &m);
    mutex_unlock(&m);
}
```


Bounded Buffer with CVs/Mutexes

```
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:

```
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:

```
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(&s->m);
    while (s->v <= 0)
        cond_wait(&s->c, &s->m);
    s->v--;
    mutex_unlock(&s->m);
}

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

Xv6: Sleeplock

```
struct sleeplock {
    uint locked;
    struct spinlock lk;
    char *name;
    int pid;
};

void initsleeplock(struct sleeplock *lk,
                  char *name) {
    initlock(&lk->lk, "sleep lock");
    lk->name = name;
    lk->locked = 0;
    lk->pid = 0;
}
```

```
void acquiresleep(struct sleeplock *lk) {
    acquire(&lk->lk);
    while (lk->locked) {
        sleep(lk, &lk->lk);
    }
    lk->locked = 1;
    lk->pid = myproc()->pid;
    release(&lk->lk);
}

void releasesleep(struct sleeplock *lk) {
    acquire(&lk->lk);
    lk->locked = 0;
    lk->pid = 0;
    wakeup(lk);
    release(&lk->lk);
}
```

Xv6: Sleep & Wakeup

```
void sleep(void *chan,
           struct spinlock *lk) {
    struct proc *p = myproc();

    if (lk != &p->lock) {
        acquire(&p->lock);
        release(lk);
    }
    p->chan = chan;
    p->state = SLEEPING;
    sched();

    p->chan = 0;
    if (lk != &p->lock) {
        release(&p->lock);
        acquire(lk);
    }
}
```

```
void wakeup(void *chan) {
    struct proc *p;

    for (p = proc; p < &proc[NPROC]; p++) {
        acquire(&p->lock);
        if (p->state == SLEEPING &&
            p->chan == chan) {
            p->state = RUNNABLE;
        }
        release(&p->lock);
    }
}
```

Summary

- **Disabling interrupts**
 - Only for the kernel on a single CPU
- **Spinlocks**
 - Busy waiting, implemented using atomic instructions
- **Semaphores**
 - Binary semaphore = mutex (\cong lock)
 - Counting semaphore
- **Monitors**
 - Language construct with condition variables
- **Mutexes + condition variables**
 - Used in Pthreads