

Condition Variable Timelines

- It can help to view timelines
- Next slide shows the following scenario:
 - U wants to update R when C is true
 - It grabs mutex M, checks C (false) and so does a Wait on V, linked to M
 - Wait unlocks M and sleeps...
 - S updates R (with proper M usage) and signals V
 - V locks M, wakes U, but C is still false, so U waits again
 - Wait unlocks M and sleeps...
 - S updates R again and signals V
 - V locks M, wakes U, and C is now true, so U proceeds to modify R and then unlocks M



S

```

pthread_lock(&m) → locked(U)
while (!C) {
    pthread_cond_wait(&v, &m) → unlocked
    (sleeping, Zzzz...) → locked(S) ← pthread_lock(&m)
    Do something to R
    pthread_cond_signal(&v)
    pthread_unlock(&m) → unlocked
    // wakeup; → locked(U)
    // C == False
    pthread_cond_wait(&v, &m) → unlocked
    (sleeping, Zzzz...) → locked(S) ← pthread_lock(&m)
    Do something to R
    pthread_cond_signal(&v)
    pthread_unlock(&m) → unlocked
    // wakeup; → locked(U)
    // C == True
}
... access and modify R ...
pthread_unlock(&m) → unlocked

```

The diagram illustrates the execution flow of a `pthread_cond_wait` loop. It shows the state of a mutex (locked/unlocked) and a condition variable (signaled/not signaled) across different points in the code. Red arrows indicate the flow of execution and state changes. Dotted lines with triangles indicate the thread's state (sleeping/wakeful).

- Initial State:** The mutex is **locked(U)** (unlocked by the caller).
- First Wait:** The thread calls `pthread_cond_wait(&v, &m)`. It becomes **unlocked** and enters a sleeping state (Zzzz...). The mutex becomes **locked(S)** (locked by the thread).
- First Wakeup:** The thread is woken up (indicated by a dotted line with a triangle). The mutex becomes **unlocked**. The thread checks the condition `C` (which is `False`).
- Second Wait:** The thread calls `pthread_cond_wait(&v, &m)` again. It becomes **unlocked** and enters a sleeping state (Zzzz...). The mutex becomes **locked(S)**.
- Second Wakeup:** The thread is woken up. The mutex becomes **unlocked**. The thread checks the condition `C` (which is now `True`).
- Final State:** The thread exits the loop. The mutex becomes **unlocked** after the final `pthread_unlock(&m)` call.

Using Multiple Mutexes

- A typical program may have many critical shared resources, each protected by its own mutex (e.g. *resource_i* protected by mutex *m_i*).
- A thread may want exclusive access to more than one such resource at a time.
- This should be easy:
 - `pthread_mutex_lock(&m1);`
`pthread_mutex_lock(&m2);`
... do stuff to `resource1` and `resource2` ...
`pthread_mutex_unlock(&m1);`
`pthread_mutex_unlock(&m2);`
- Let's see....



2 Mutexes (Preamble)

We have two resources, one mutex each for protection.

```
int resource1, resource2 ; // Two global resources

pthread_mutex_t m1 = PTHREAD_MUTEX_INITIALIZER ; // Protects resource1
pthread_mutex_t m2 = PTHREAD_MUTEX_INITIALIZER ; // Protects resource2
```



2 Mutexes (GoingUp)

```
// Going Up: sets r1 = min(r1,r2), r2 = r1+r2
void *GoingUp(void *a) {

    int tmp;

    pthread_mutex_lock(&m1);
    pthread_mutex_lock(&m2);

    tmp = resource1 + resource2;
    if (resource2 < resource1) { resource1 = resource2 ;}
    resource2 = tmp;

    pthread_mutex_unlock(&m2);
    pthread_mutex_unlock(&m1);

    pthread_exit(NULL);
}
```



2 Mutexes (GoingDown)

```
// Going Down: sets r1 = max(r1,r2), r2 = max(r1,r2)-min(r1,r2)
void *GoingDown(void *a) {
    int tmp;

    pthread_mutex_lock(&m2);
    pthread_mutex_lock(&m1);

    if (resource2 < resource1) {
        resource2 = resource1 - resource2;
    } else {

        tmp = resource2 - resource1;
        resource1 = resource2;
        resource2 = tmp;
    }

    pthread_mutex_unlock(&m1);
    pthread_mutex_unlock(&m2);

    pthread_exit(NULL);
}
```



2 Mutexes (Main Program)

```
int main (int argc, const char * argv[]) {
    static pthread_t goingup,goingdown ;
    long rc;

    resource1 = 13; resource2 = 42;
    printf("r1,r2 = %d,%d\n", resource1, resource2);

    printf("Creating GoingUp:\n");
    rc = pthread_create(&goingup,NULL,GoingUp,(void *)0);
    if (rc) { ... }
    printf("Creating GoingDown:\n");
    rc = pthread_create(&goingdown,NULL,GoingDown,(void *)0);
    if (rc) { ... }

    printf("Waiting to join threads....\n");
    pthread_join( goingdown, NULL);
    pthread_join( goingup, NULL);

    printf("R1,R2 = %d,%d\n", resource1, resource2);
    printf("All Done!\n");

    return 0;
}
```

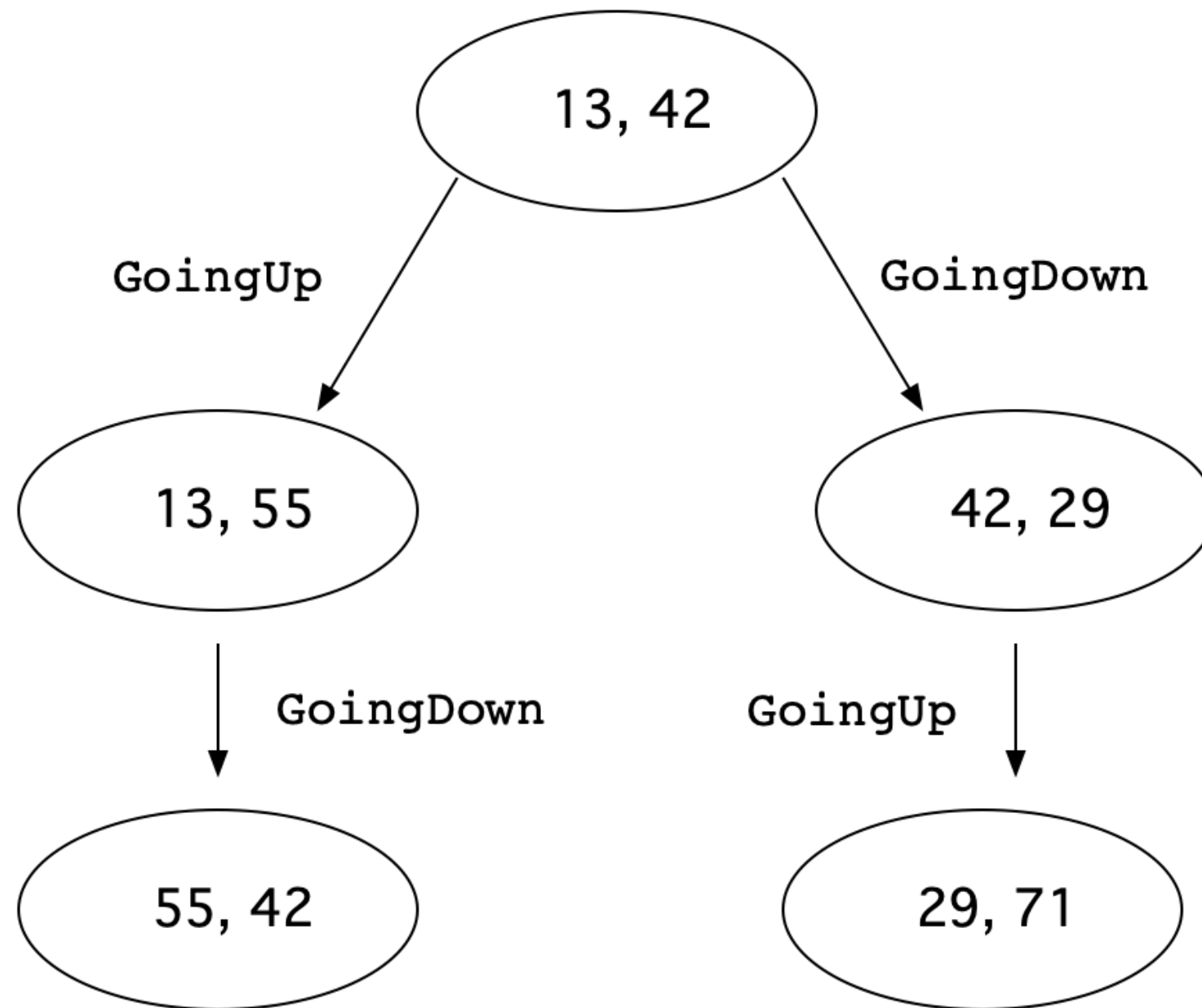


Expected Behaviour

- We have two threads, each of which (GoingUp,GoingDown) uses both resources.
 - `r1,r2`
- We expect to see one of two possibilities:
 - An execution of GoingUp followed by one of GoingDown.
 - 13,42 becomes 13,55 becomes 55,42
 - An execution of GoingDown followed by one of GoingUp.
 - 13,42 becomes 42, 29 becomes 29,71



Expected Behaviour (Diagram)



What actually happens!

- Many times we see the expected behaviour
- But occasionally,
 - it hangs.....
- Another way to cause DEADLOCK !
- The solution:
 - all threads should acquire multiple locks in the same order as each other.

