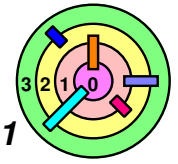
POSIX Condition Variables

Guarded command	*POSIX implementation*

`when (guard) [`
`  statement 1;`
`  ...`
`  statement n;`
`]`
	`pthread_mutex_lock(&mutex);`
	`while(!guard)`
	`  pthread_cond_wait(`
	`  &cv,`
	`  &mutex);`
	`statement 1;`
	`...`
	`statement n;`
	`pthread_mutex_unlock(&mutex);`

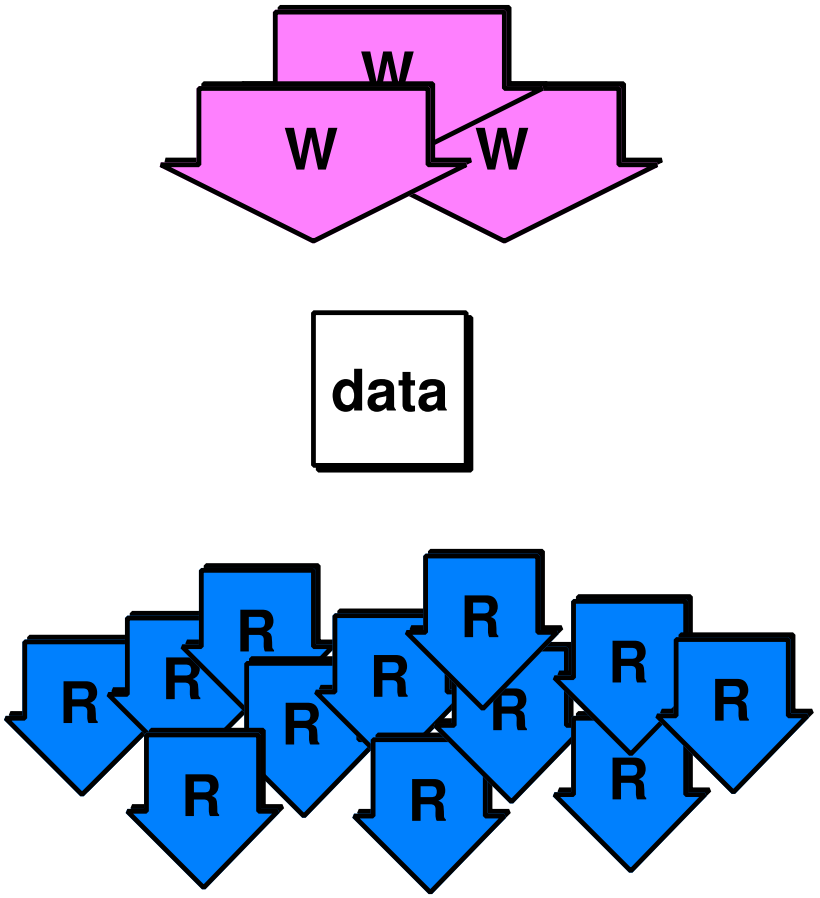
`[ /* code`
`  * modifying`
`  * the guard`
`  */`
`]`
	`pthread_mutex_lock(&mutex);`
	`/*code modifying the guard:*/`
	`...`
	`pthread_cond_broadcast(&cv);`
	`pthread_mutex_unlock(&mutex);`

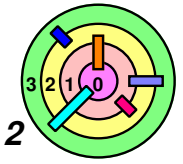
- don't believe that `pthread_cond_signal/broadcast()` can be called without locking the mutex



Copyright © William C. Cheng

Readers-Writers Problem





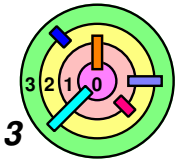
Copyright © William C. Cheng

Readers-Writers Pseudocode

```
reader( ) {
  when (writers == 0) [
    readers++;
  ]
  /* read */
  [readers--;]
}
```

```
writer( ) {
  when ((writers == 0) &&
        (readers == 0)) [
    writers++;
  ]
  /* write */
  [writers--;]
}
```

- this is synchronization code



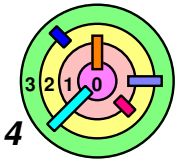
Copyright © William C. Cheng

Pseudocode with Assertions

```
reader( ) {
  when (writers == 0) [
    readers++;
  ]
  // sanity check
  assert((writers == 0) &&
       (readers > 0));
  /* read */
  [readers--;]
}
```

```
writer( ) {
  when ((writers == 0) &&
       (readers == 0)) [
    writers++;
  ]
  // sanity check
  assert((readers == 0) &&
       (writers == 1));
  /* write */
  [writers--;]
}
```

- the sanity checks are really not necessary



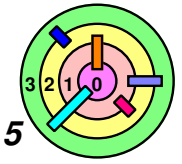
Copyright © William C. Cheng

Readers-Writers Pseudocode

```
reader( ) {
  when (writers == 0) [
    readers++;
  ]
  /* read */
  [readers--;]
}
```

```
writer( ) {
  when ((writers == 0) &&
        (readers == 0)) [
    writers++;
  ]
  /* write */
  [writers--;]
}
```

- since `readers` is part of the guard in the implementation of `[readers--;]`, you may need to signal/broadcast the corresponding condition used to implement that guard
  - in this case, only have to signal if `readers` becomes 0 *(if the guard may become true)*

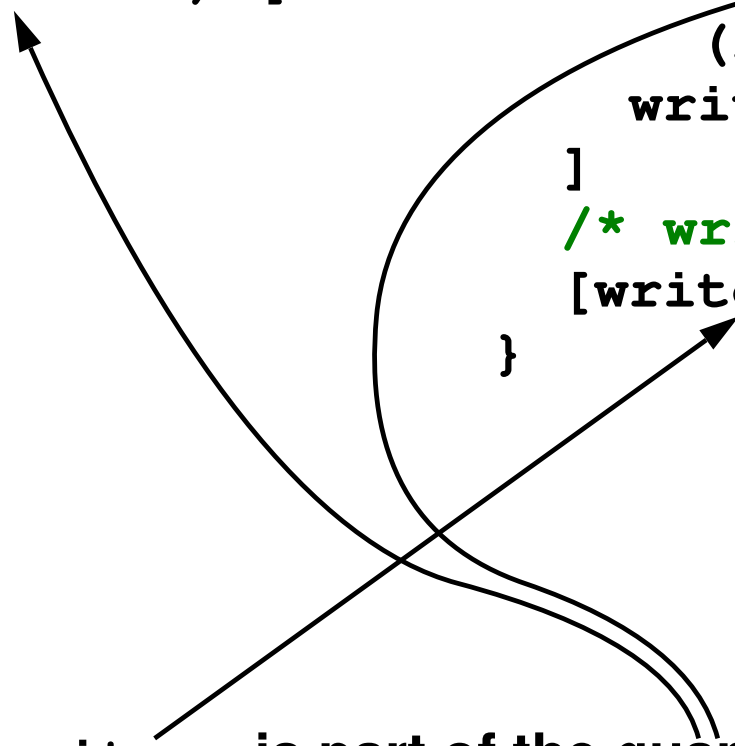


Copyright © William C. Cheng

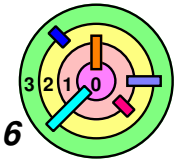
Readers-Writers Pseudocode

```
reader( ) {
  when (writers == 0) [
    readers++;
  ]
  /* read */
  [readers--;]
}
```

```
writer( ) {
  when ((writers == 0) &&
        (readers == 0)) [
    writers++;
  ]
  /* write */
  [writers--;]
}
```



- also, since `writers` is part of the guards (and these two guards are not identical), in the implementation of `[writers--;]`, you may need to signal/broadcast the corresponding conditions used to implement these guards
  - in this case, signal/browdcast if `writers` becomes 0 *(if the guard may become true)*

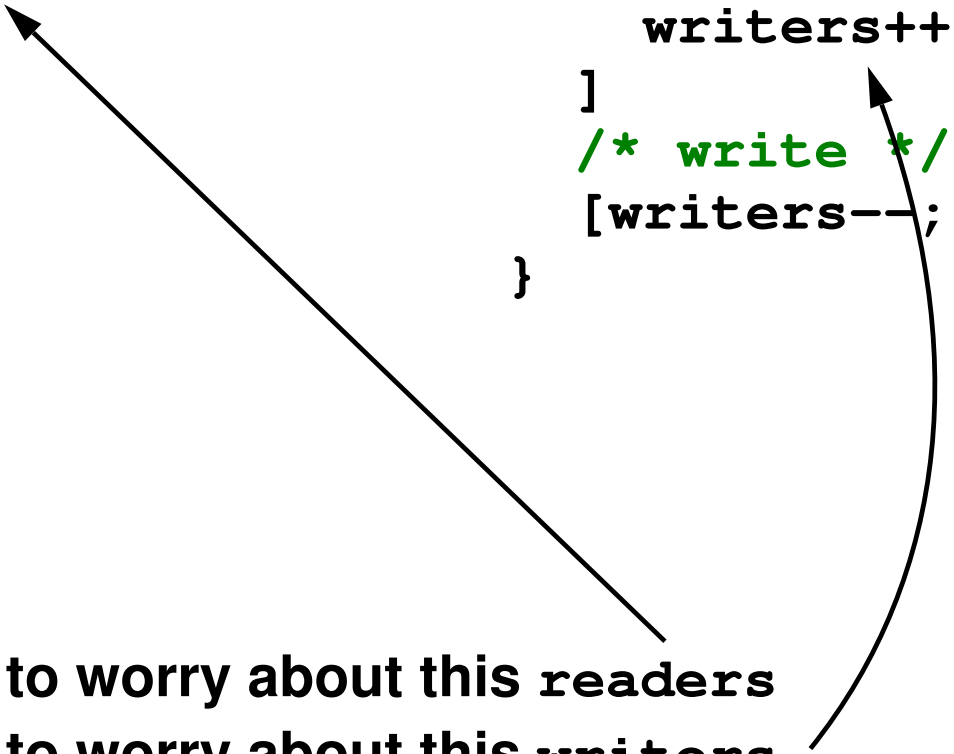


Copyright © William C. Cheng

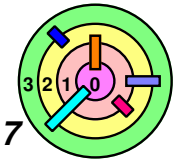
Readers-Writers Pseudocode

```
reader( ) {
  when (writers == 0) [
    readers++;
  ]
  /* read */
  [readers--;]
}
```

```
writer( ) {
  when ((writers == 0) &&
        (readers == 0)) [
    writers++;
  ]
  /* write */
  [writers--;]
}
```



- don't have to worry about this `readers`
- don't have to worry about this `writers`
  - you need to look at your program logic and figure when signal/broadcast conditions won't be useful
    - it's *not wrong* to signal/broadcast here, it's just *wasteful/inefficient*



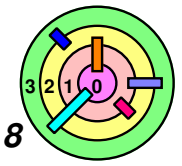
Copyright © William C. Cheng

Readers-Writers Pseudocode

```
reader( ) {
  when (writers == 0) [
    readers++;
  ]
  /* read */
  [readers--;]
}
```

```
writer( ) {
  when ((writers == 0) &&
        (readers == 0)) [
    writers++;
  ]
  /* write */
  [writers--;]
}
```

- `writers` behaves like a binary semaphore
- `readers` behaves like a counting semaphore
- but they are *not semaphores*
  - due to the definition of a semaphore



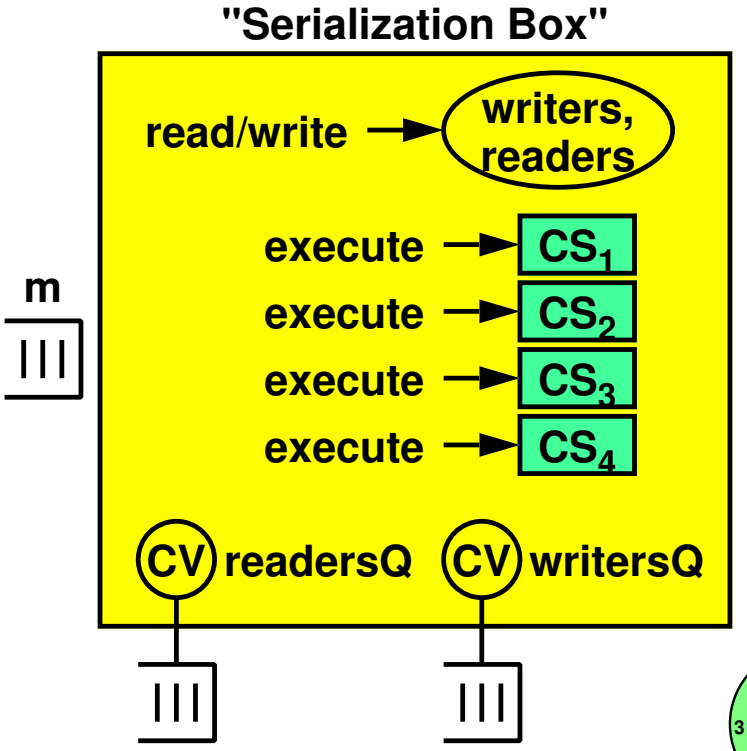
Copyright © William C. Cheng

Solution with POSIX Threads

```
reader( ) {
  when (writers == 0) [
    readers++;
  ]
  /* read */
  [readers--;]
}
```

```
writer( ) {
  when ((writers == 0) &&
        (readers == 0)) [
    writers++;
  ]
  /* write */
  [writers--;]
}
```

- to be even more "efficient", can use multiple CVs
  - so you don't have to wake up a thread unnecessarily
    - here we use one CV for reader's guard and one CV for writer's guard (since we want to wake them up separately)



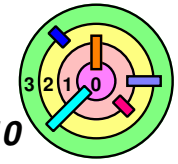
Copyright © William C. Cheng

Recall POSIX Guarded Command Implementation

Guarded command	*POSIX implementation*

`when (guard) [`
`  statement 1;`
`  ...`
`  statement n;`
`]`
	`pthread_mutex_lock(&mutex);`
	`while(!guard)`
	`  pthread_cond_wait(`
	`  &cv,`
	`  &mutex);`
	`statement 1;`
	`...`
	`statement n;`
	`pthread_mutex_unlock(&mutex);`

`[ /* code`
`  * modifying`
`  * the guard`
`  */`
`]`
	`pthread_mutex_lock(&mutex);`
	`/*code modifying the guard:*/`
	`...`
	`pthread_cond_broadcast(&cv);`
	`pthread_mutex_unlock(&mutex);`



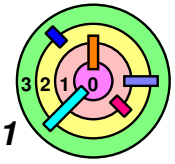
Copyright © William C. Cheng

Solution with POSIX Threads

```
reader( ) {
  pthread_mutex_lock(&m);
  while (!(writers == 0))
    pthread_cond_wait(
        &readersQ, &m);
  readers++;
  pthread_mutex_unlock(&m);
  /* read */
  pthread_mutex_lock(&m);
  if (--readers == 0)
    pthread_cond_signal(
        &writersQ);
  pthread_mutex_unlock(&m);
}
```

```
writer( ) {
  pthread_mutex_lock(&m);
  while(!((readers == 0) &&
      (writers == 0)))
    pthread_cond_wait(
        &writersQ, &m);
  writers++;
  pthread_mutex_unlock(&m);
  /* write */
  pthread_mutex_lock(&m);
  writers--;
  pthread_cond_signal(
      &writersQ);
  pthread_cond_broadcast(
      &readersQ);
  pthread_mutex_unlock(&m);
}
```

- one mutex (`m`) and two condition variables (`readersQ` and `writersQ`)



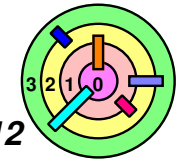
Copyright © William C. Cheng

Solution with POSIX Threads

```
reader( ) {
  when (writers == 0) [
    readers++;
  ]
  /* read */
  [readers--;]
}
```

```
reader( ) {
  pthread_mutex_lock(&m);
  while (!(writers == 0))
    pthread_cond_wait(
        &readersQ, &m);
  readers++;
  pthread_mutex_unlock(&m);
  /* read */
  pthread_mutex_lock(&m);
  if (--readers == 0)
    pthread_cond_signal(
        &writersQ);
  pthread_mutex_unlock(&m);
}
```

- one mutex (`m`) and two condition variables (`readersQ` and `writersQ`)



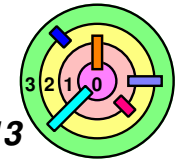
Copyright © William C. Cheng

Solution with POSIX Threads

```
writer( ) {
  when ((writers == 0) &&
       (readers == 0)) [
    writers++;
  ]
  /* write */
  [writers--;]
}
```

```
writer( ) {
  pthread_mutex_lock(&m);
  while(!((readers == 0) &&
       (writers == 0)))
    pthread_cond_wait(
         &writersQ, &m);
  writers++;
  pthread_mutex_unlock(&m);
  /* write */
  pthread_mutex_lock(&m);
  writers--;
  pthread_cond_signal(
       &writersQ);
  pthread_cond_broadcast(
       &readersQ);
  pthread_mutex_unlock(&m);
}
```

- one mutex (`m`) and two condition variables (`readersQ` and `writersQ`)



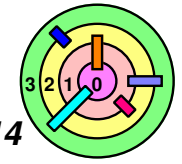
Copyright © William C. Cheng

The Starvation Problem

```
reader( ) {
  when (writers == 0) [
    readers++;
  ]
  /* read */
  [readers--;]
}
```

```
writer( ) {
  when ((writers == 0) &&
       (readers == 0)) [
    writers++;
  ]
  /* write */
  [writers--;]
}
```

- Can the writer never get a chance to write?
  - yes, if there are always readers
  - so, this implementation can be unfair to writers
- Solution
  - once a writer arrives, shut the door on new readers
    - `writers` now means the number of writers *wanting* to write
    - use `active_writers` to make sure that only one writer can do the actual writing at a time



Copyright © William C. Cheng

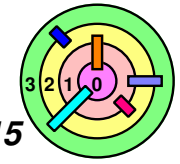
Solving The Starvation Problem

```
reader( ) {
  when (writers == 0) [
    readers++;
  ]
  /* read */
  [readers--;]
}
```

```
writer( ) {
  [writers++;]
  when ((readers == 0) &&
       (active_writers == 0))
  [
    active_writers++;
  ]
  /* write */
  [ writers--;
    active_writers--;
  ]
}
```

- now it's unfair to the readers
- isn't writing more important than reading anyway?

⇨ This is an example of how to give threads priority *without* assigning priorities to threads!

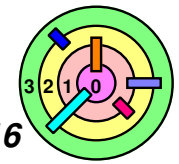


Copyright © William C. Cheng

Improved Reader

```
reader( ) {
  pthread_mutex_lock(&m);
  while (!(writers == 0))
    pthread_cond_wait(
        &readersQ, &m);
  readers++;
  pthread_mutex_unlock(&m);
  /* read */
  pthread_mutex_lock(&m);
  if (--readers == 0)
    pthread_cond_signal(
        &writersQ);
  pthread_mutex_unlock(&m);
}
```

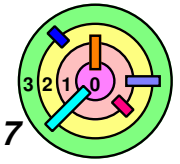
- exactly the same as before!



Copyright © William C. Cheng

Improved Writer

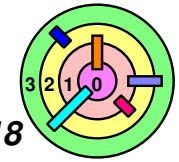
```
writer( ) {
  pthread_mutex_lock(&m);
  writers++;
  while (!((readers == 0) &&
       (active_writers == 0))) {
    pthread_cond_wait(&writersQ, &m);
  }
  active_writers++;
  pthread_mutex_unlock(&m);
  /* write */
  pthread_mutex_lock(&m);
  writers--;
  active_writers--;
  if (writers > 0)
    pthread_cond_signal(&writersQ);
  else
    pthread_cond_broadcast(&readersQ);
  pthread_mutex_unlock(&m);
}
```



Copyright © William C. Cheng

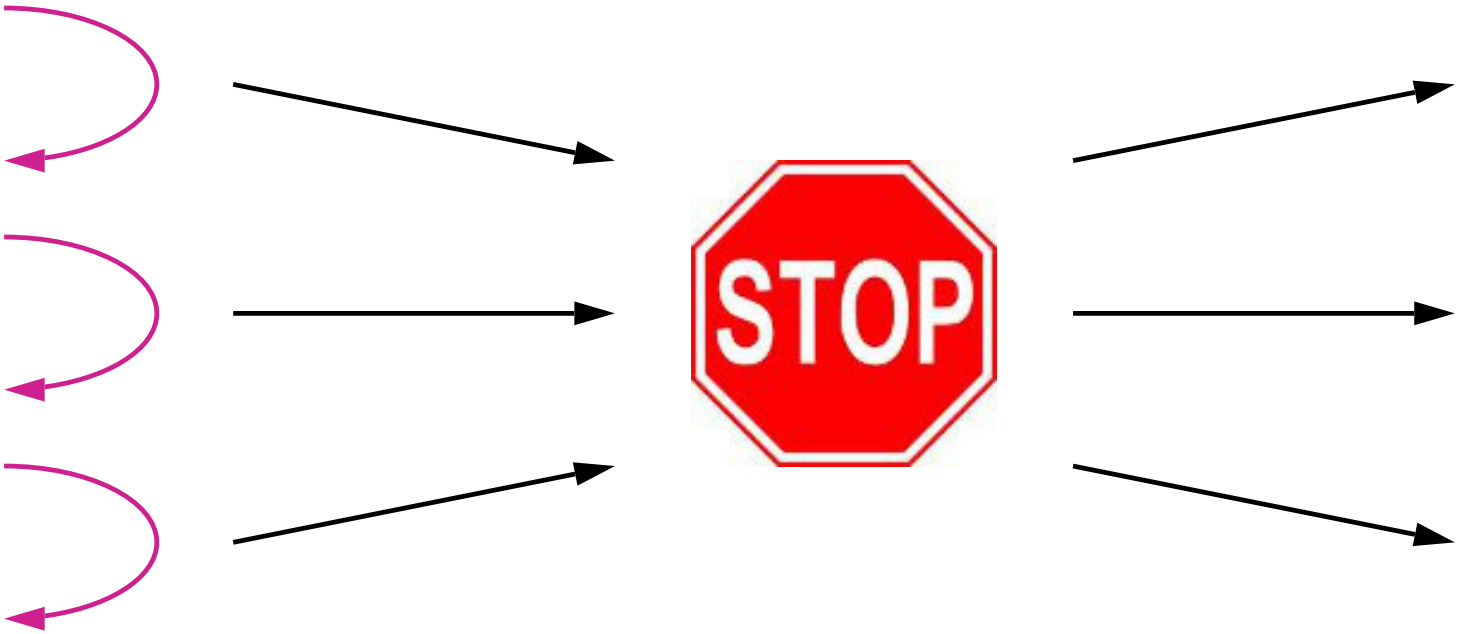
New, From POSIX!

```
int pthread_rwlock_init(
        pthread_rwlock_t *lock,
        pthread_rwlockattr_t *att);
int pthread_rwlock_destroy(
        pthread_rwlock_t *lock);
int pthread_rwlock_rdlock(
        pthread_rwlock_t *lock);
int pthread_rwlock_wrlock(
        pthread_rwlock_t *lock);
int pthread_rwlock_tryrdlock(
        pthread_rwlock_t *lock);
int pthread_rwlock_trywrlock(
        pthread_rwlock_t *lock);
int pthread_timedrwlock_rdlock(
        pthread_rwlock_t *lock, struct timespec *ts);
int pthread_timedrwlock_wrlock(
        pthread_rwlock_t *lock, struct timespec *ts);
int pthread_rwlock_unlock(
        pthread_rwlock_t *lock);
```

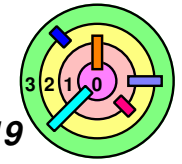


Copyright © William C. Cheng

Barriers



- When a thread reaches a barrier, it must stop (do nothing) and simply wait for other threads to arrive at the same barrier
  - when all the threads that were suppose to arrive at the barrier have all arrived at the barrier, they are all given the signal to proceed forward
    - the barrier is then reset
- Ex: fork/join (fork to create parallel execution)

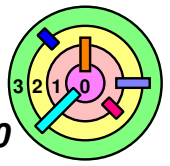


Copyright © William C. Cheng

A Solution?

```
int count = 0;
pthread_mutex_t m;
pthread_cond_t BarrierQueue;
void barrier_sync() {
  pthread_mutex_lock(&m);
  if (++count < n) {
    pthread_cond_wait(&BarrierQueue, &m);
  } else {
    count = 0;
    pthread_cond_broadcast(&BarrierQueue);
  }
  pthread_mutex_unlock(&m);
}
```

- the idea here is to have the last thread broadcast the condition while all the other threads are blocked at waiting for the condition to be signaled
- as it turns out, `pthread_cond_wait()` might return ***spontaneously***, so this won't work
  - http://pubs.opengroup.org/onlinepubs/009604599/functions/pthread_cond_signal.html

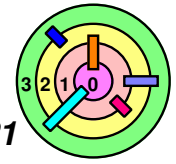


Copyright © William C. Cheng

A Solution?

```
int count = 0;
pthread_mutex_t m;
pthread_cond_t BarrierQueue;
void barrier_sync() {
  pthread_mutex_lock(&m);
  if (++count < n) {
    while (count < n)
      pthread_cond_wait(&BarrierQueue, &m);
  } else {
    pthread_cond_broadcast(&BarrierQueue);
    count = 0;
  }
  pthread_mutex_unlock(&m);
}
```

- if the `n` th thread wakes up all the other blocked threads, most likely, *none* of these threads will see `count == n`

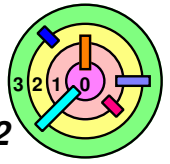


Copyright © William C. Cheng

A Solution?

```
int count = 0;
pthread_mutex_t m;
pthread_cond_t BarrierQueue;
void barrier_sync() {
  pthread_mutex_lock(&m);
  if (++count < n) {
    while (count < n)
      pthread_cond_wait(&BarrierQueue, &m);
  } else {
    pthread_cond_broadcast(&BarrierQueue);
  }
  pthread_mutex_unlock(&m);
  count = 0;
}
```

- if the `n` th thread wakes up all the other blocked threads, most likely, ***none*** of these threads will see `count == n`
- moving `count = 0` around won't help
  - cannot guarantee all `n` threads will exit the barrier

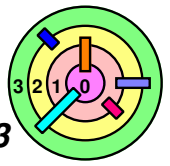


Copyright © William C. Cheng

Barrier in POSIX Threads

```
int count = 0;
pthread_mutex_t m;
pthread_cond_t BarrierQueue;
void barrier_sync() {
  pthread_mutex_lock(&m);
  if (++count < number) {
    int my_generation = generation;
    while(my_generation == generation)
      pthread_cond_wait(&BarrierQueue, &m);
  } else {
    count = 0;
    generation++;
    pthread_cond_broadcast(&BarrierQueue);
  }
  pthread_mutex_unlock(&m);
}
```

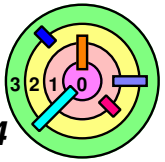
- don't use `count` in the guard since its problematic!
- introduce a new guard (with a new variable)



Copyright © William C. Cheng

More From POSIX!

```
int pthread_barrier_init(
        pthread_barrier_t *barrier,
        pthread_barrierattr_t *attr,
        unsigned int count);
int pthread_barrier_destroy(
        pthread_barrier_t *barrier);
int pthread_barrier_wait(
        pthread_barrier_t *barrier);
```



Copyright © William C. Cheng