

Condition Variable

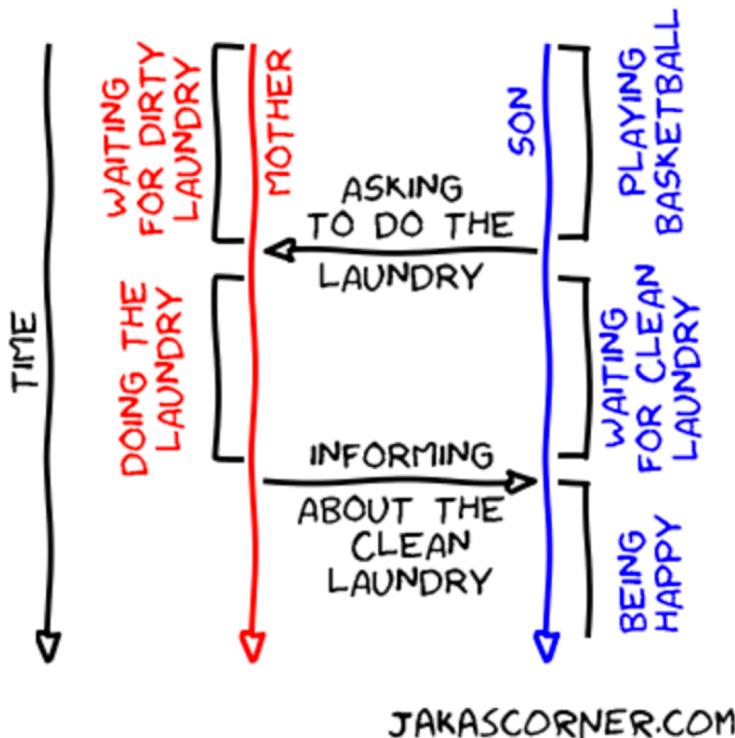
Concurrent Programming

Introduction

- What is Condition Variable?
- Pthread Condition Variable API
- Example

What is Condition Variable?

- Synchronization primitive that can be used to block a thread, or multiple threads at the same time, until another thread both modifies a shared variable (the *condition*), and notifies the condition variable



Pthread Condition Variable API

- `pthread_cond_init`
- `pthread_cond_wait`
- `pthread_cond_signal`
- `pthread_cond_broadcast`
- more APIs, but not today

pthread_cond_init

```
int pthread_cond_init(pthread_cond_t *cond,  
                      const pthread_condattr_t *attr);
```

-
- Initialize the condition variable

@param[in] cond Condition variable to be initialized

@param[in] attr Used for setting attributes of a condition variable, Default 0

@return 0 if initialization success

- You can simply use PTHREAD_COND_INITIALIZER

- ex: `pthread_cond_t cond = PTHREAD_COND_INITIALIZER;`

pthread_cond_wait

```
int pthread_cond_wait(pthread_cond_t *cond,  
                      pthread_mutex_t* mutex);
```

-
- Atomically release the *mutex* and block the calling thread on the *cond*.
 - Always return with the *mutex* acquired

@param[in] cond	Condition variable on which calling thread will block
@param[in] mutex	Mutex to be released
@return	0 if complete successfully

pthread_cond_signal

```
int pthread_cond_signal(pthread_cond_t *cond);
```

-
- Unblock one thread that is blocked on the *cond*
 - When no threads are blocked on the condition variable, it has no effect

@param[in] cond Condition variable that the thread to wake is blocking on
@return 0 if complete successfully

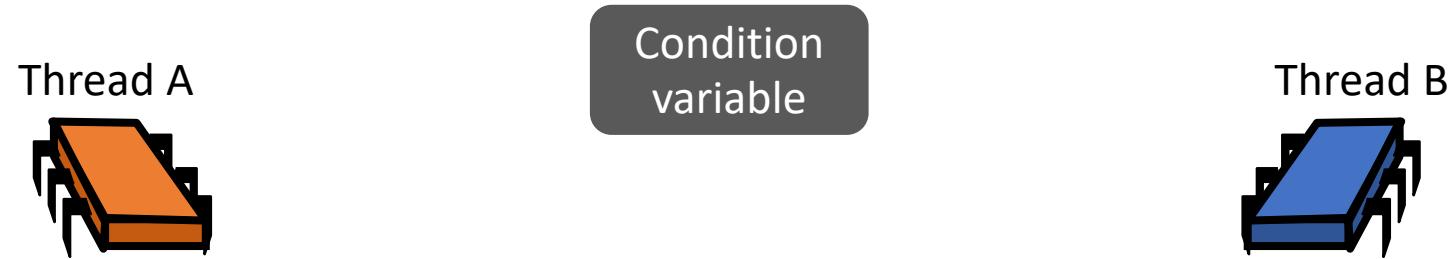
pthread_cond_broadcast

```
int pthread_cond_broadcast(pthread_cond_t *cond);
```

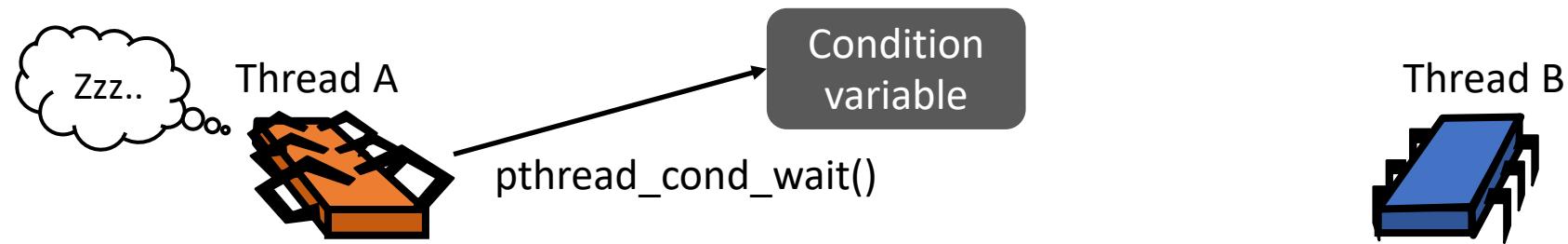
-
- Unblock all threads that is blocked on the *cond*
 - When no threads are blocked on the condition variable, it has no effect

@param[in] cond Condition variable that the threads to wake is blocking on
@return 0 if complete successfully

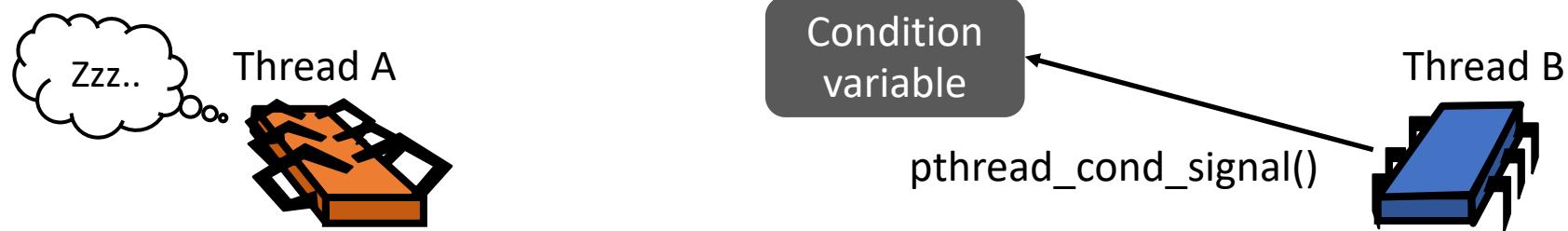
pthread_cond_wait



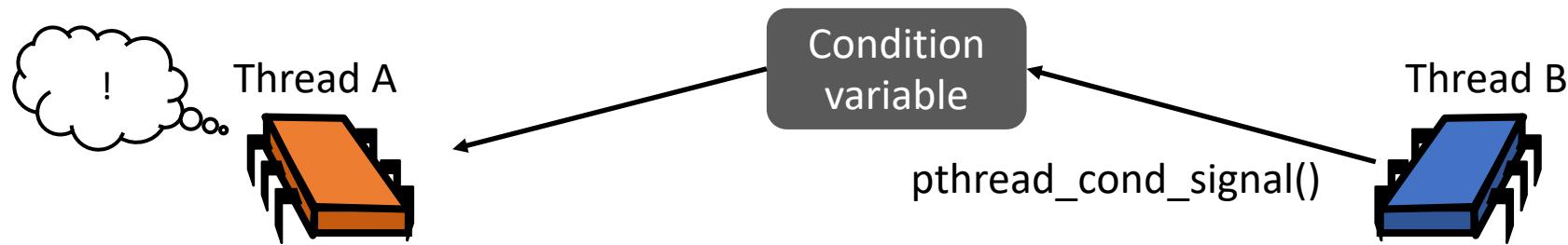
pthread_cond_wait



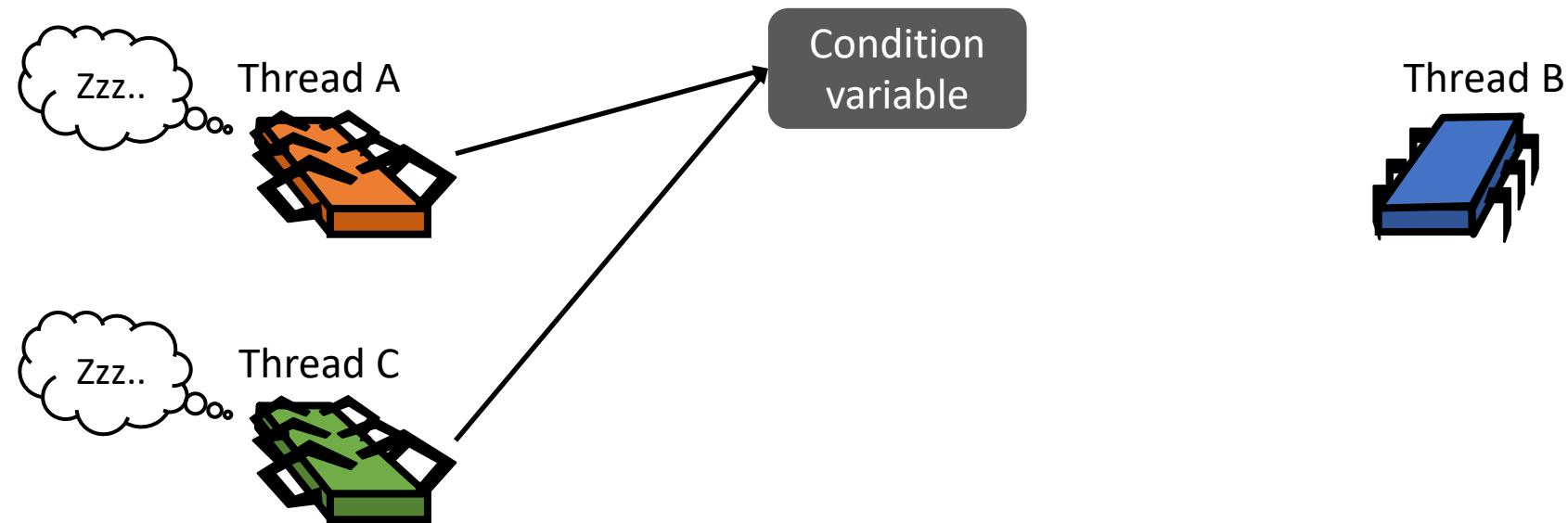
pthread_cond_signal



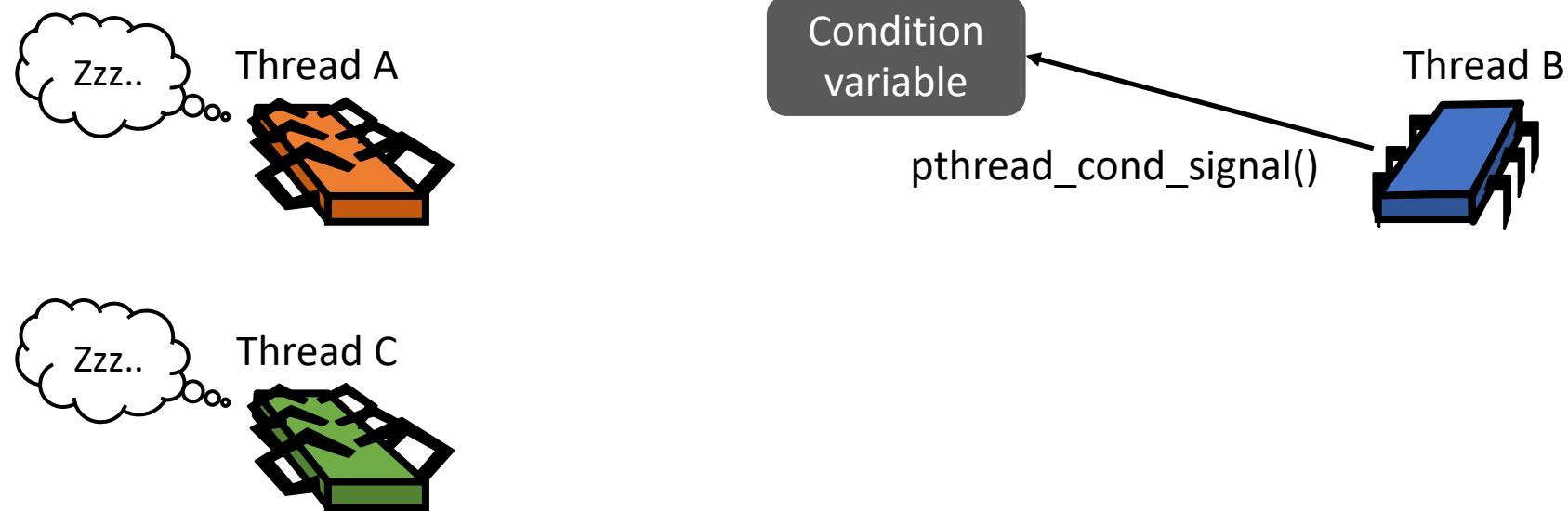
pthread_cond_signal



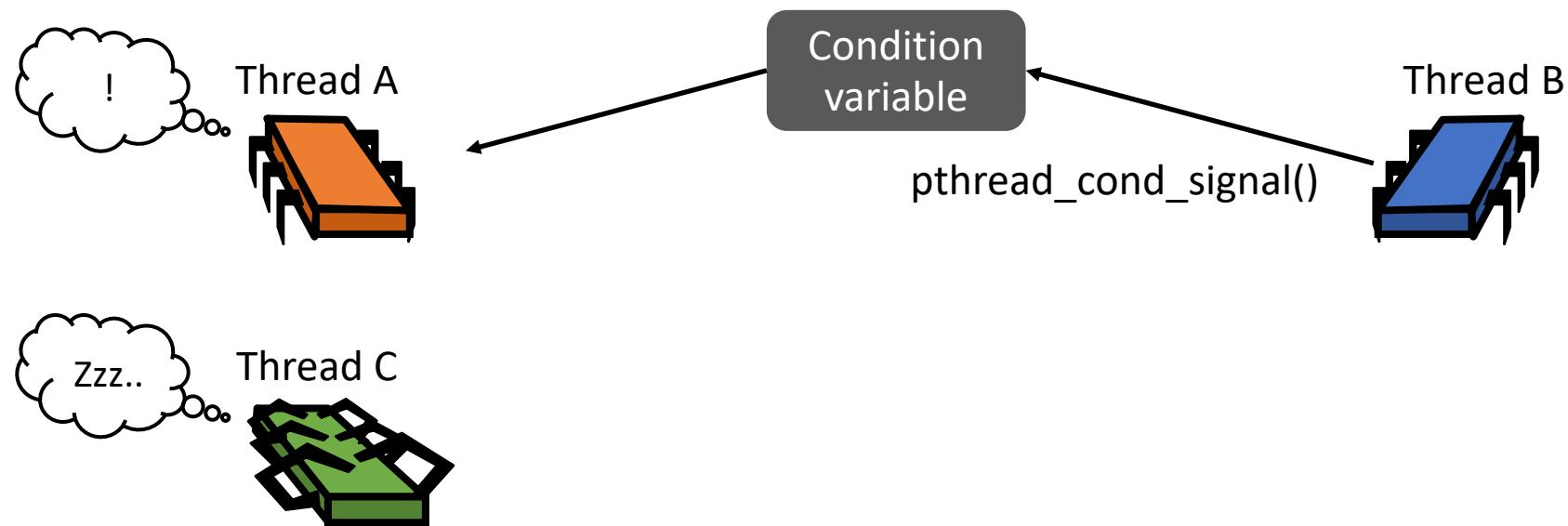
pthread_cond_signal



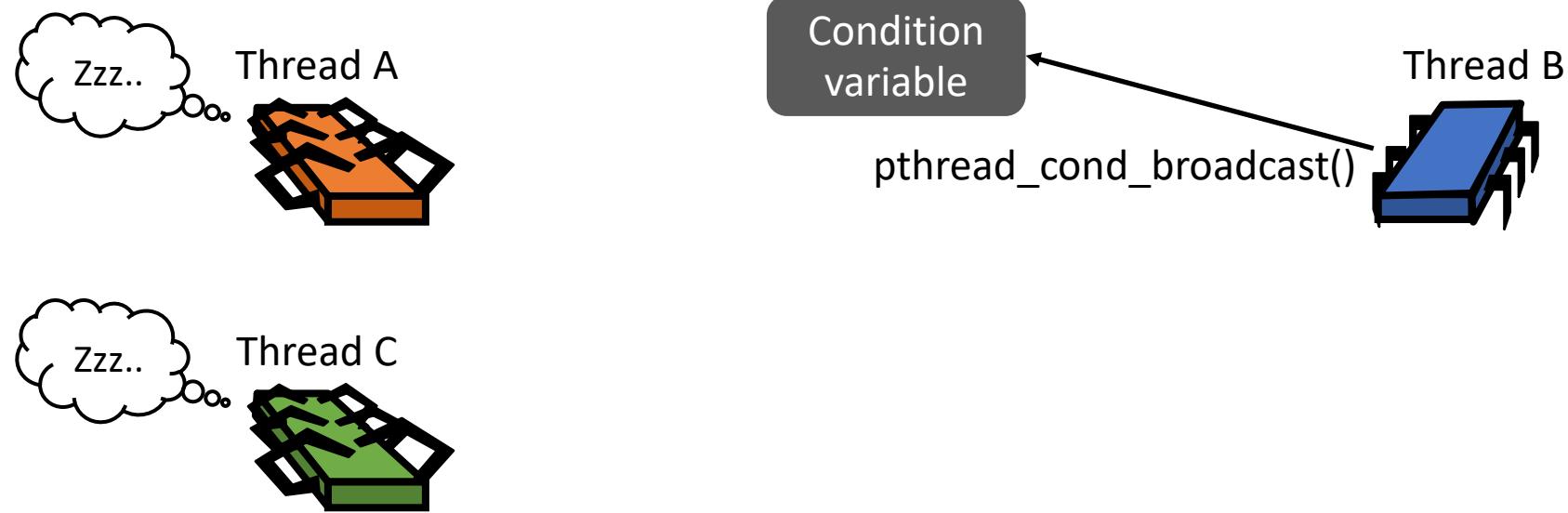
pthread_cond_signal



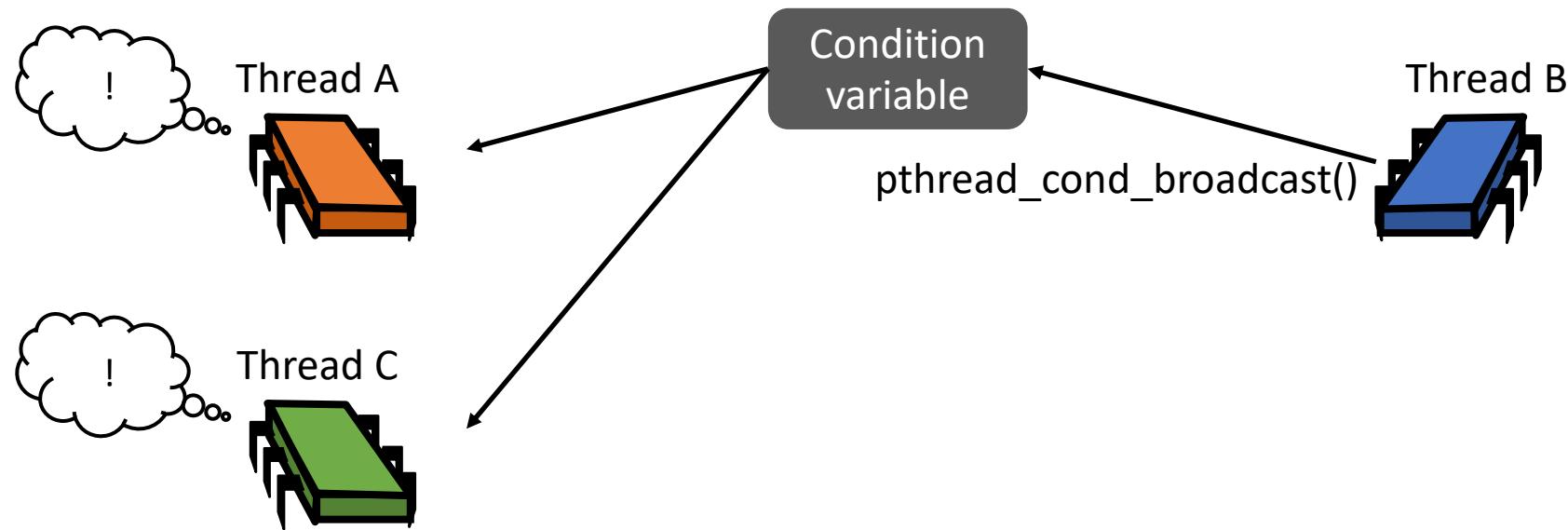
pthread_cond_signal



pthread_cond_broadcast



pthread_cond_broadcast

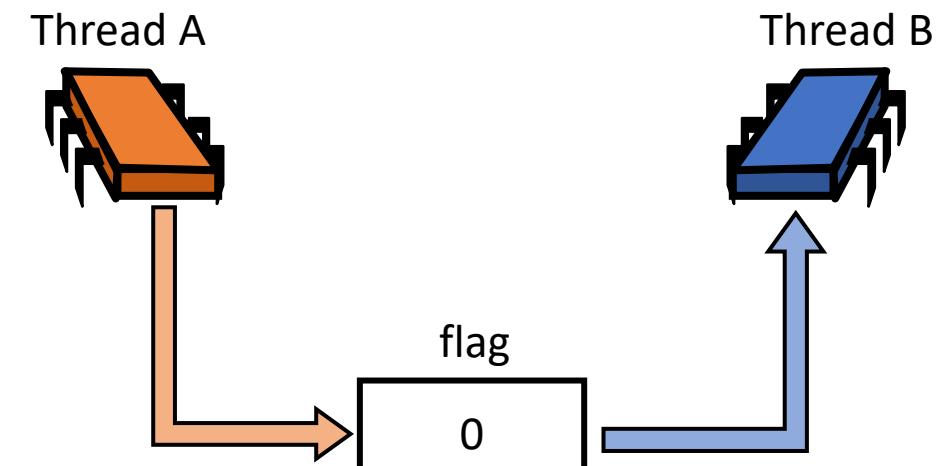


Lost wake-up problem

- Calling **`pthread_cond_signal()`/`pthread_cond_broadcast()`** without holding the corresponding *mutex* of the condition variable can lead to **lost wake-up** problem

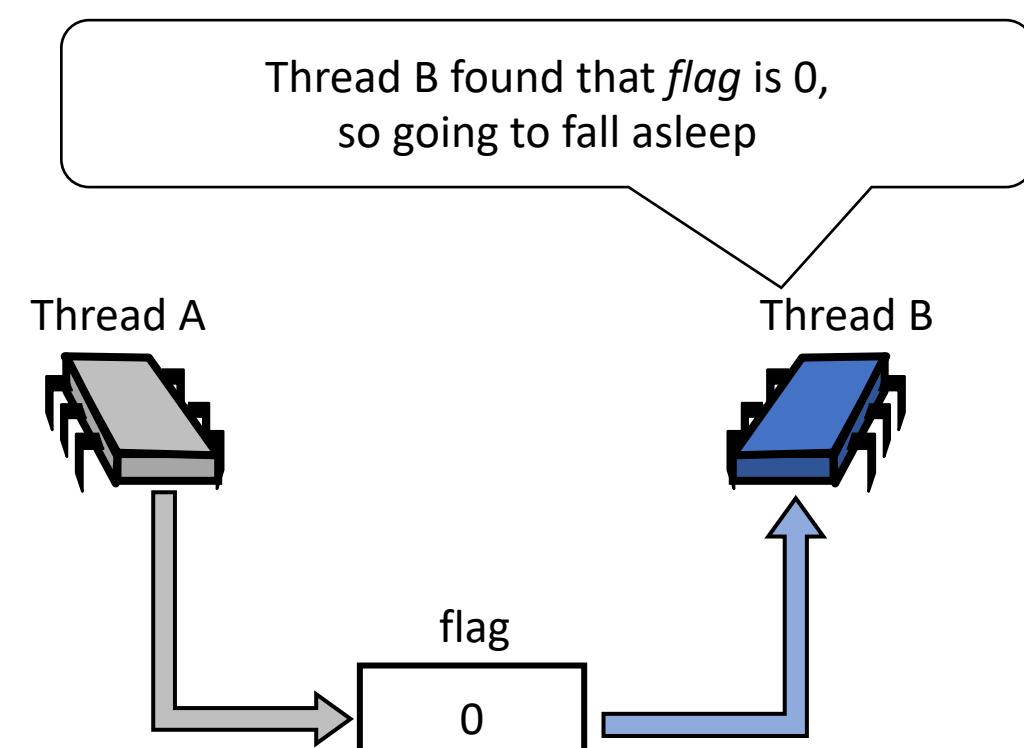
Lost wake-up problem

```
10 pthread_cond_t cond;
11 int flag;
12
→ 13 void func_threadA(void) {
14     flag = 1;
15     pthread_cond_signal(&cond);
16 }
17
→ 18 void func_threadB(void) {
19     while (flag == 0) {
20         pthread_cond_wait(&cond);
21     }
22 }
```



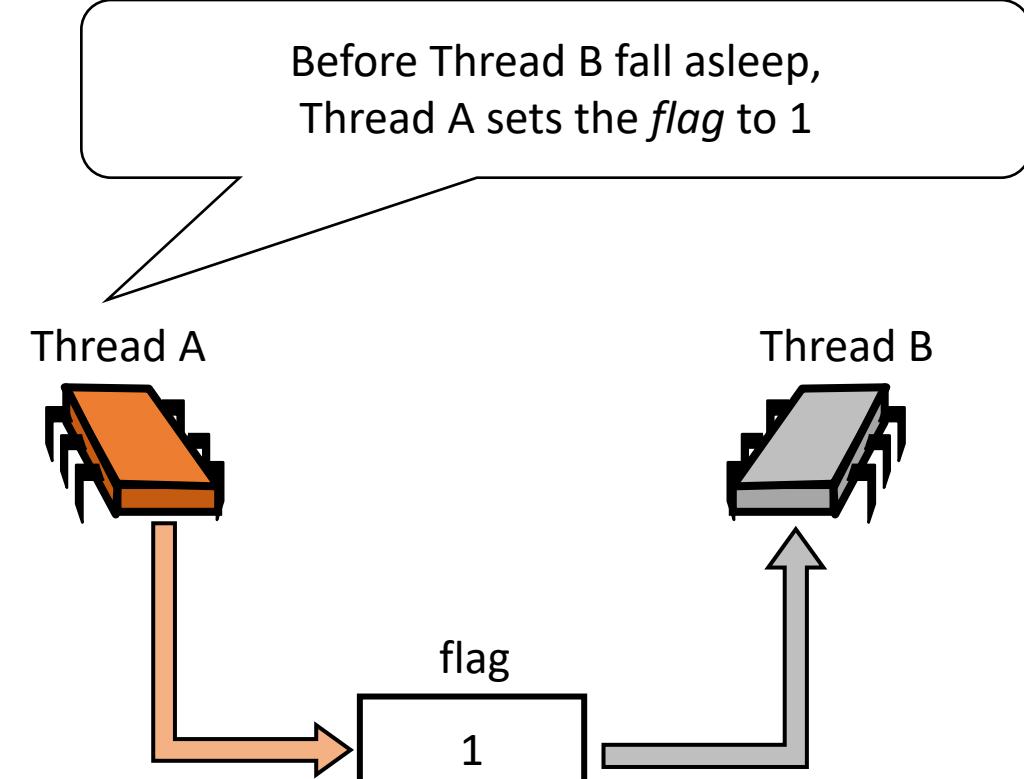
Lost wake-up problem

```
10 pthread_cond_t cond;
11 int flag;
12
13 void func_threadA(void) {
14     flag = 1;
15     pthread_cond_signal(&cond);
16 }
17
18 void func_threadB(void) {
19     while (flag == 0) {
20         pthread_cond_wait(&cond);
21     }
22 }
```



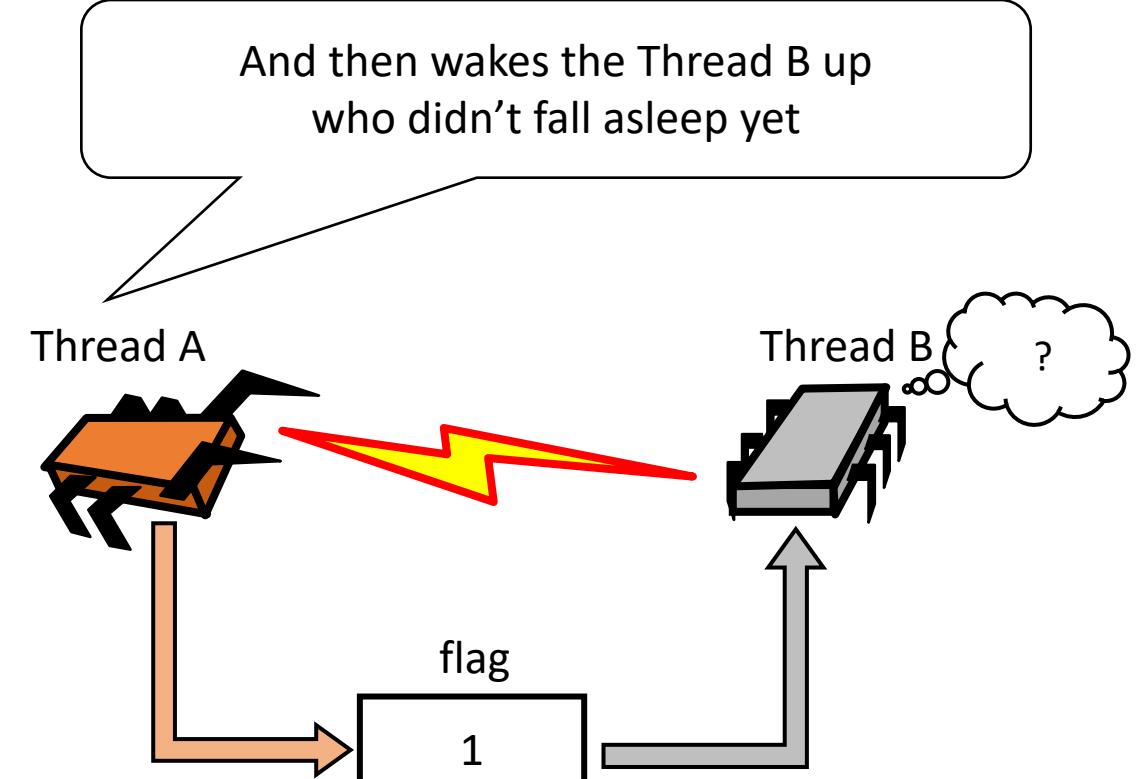
Lost wake-up problem

```
10 pthread_cond_t cond;  
11 int flag;  
12  
13 void func_threadA(void) {  
14     flag = 1;  
15     pthread_cond_signal(&cond);  
16 }  
17  
18 void func_threadB(void) {  
19     while (flag == 0) {  
20         pthread_cond_wait(&cond);  
21     }  
22 }
```



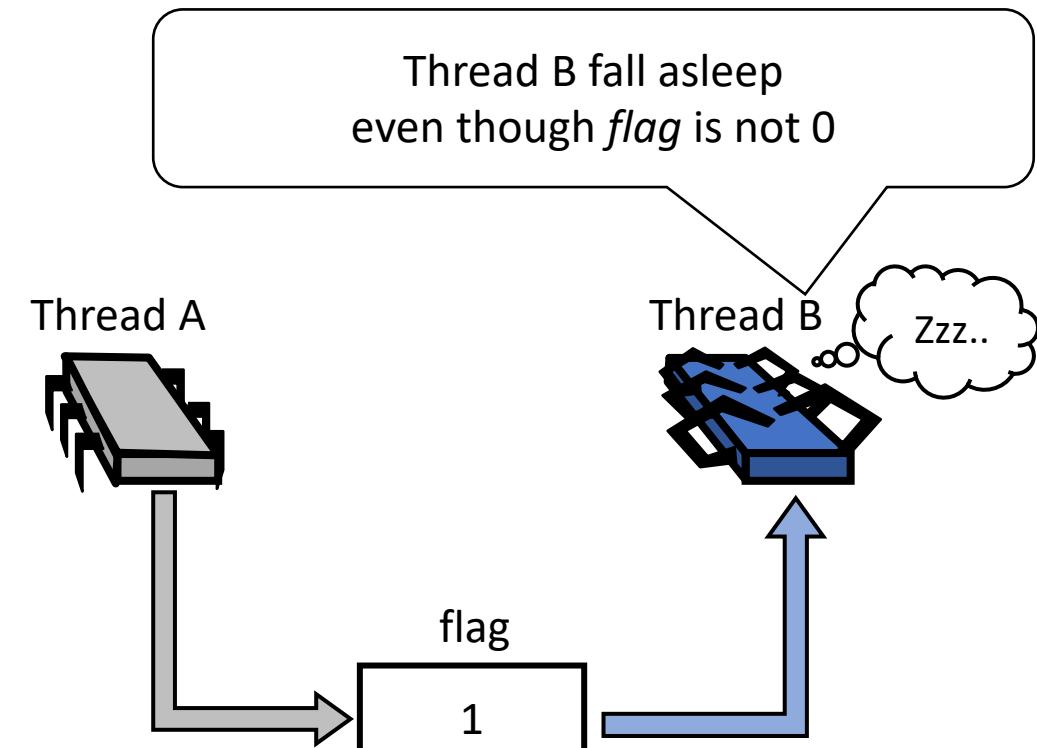
Lost wake-up problem

```
10 pthread_cond_t cond;  
11 int flag;  
12  
13 void func_threadA(void) {  
14     flag = 1;  
15     pthread_cond_signal(&cond);  
16 }  
17  
18 void func_threadB(void) {  
19     while (flag == 0) {  
20         pthread_cond_wait(&cond);  
21     }  
22 }
```



Lost wake-up problem

```
10 pthread_cond_t cond;  
11 int flag;  
12  
13 void func_threadA(void) {  
14     flag = 1;  
15     pthread_cond_signal(&cond);  
16 }  
17  
18 void func_threadB(void) {  
19     while (flag == 0) {  
20         pthread_cond_wait(&cond);  
21     }  
22 }
```

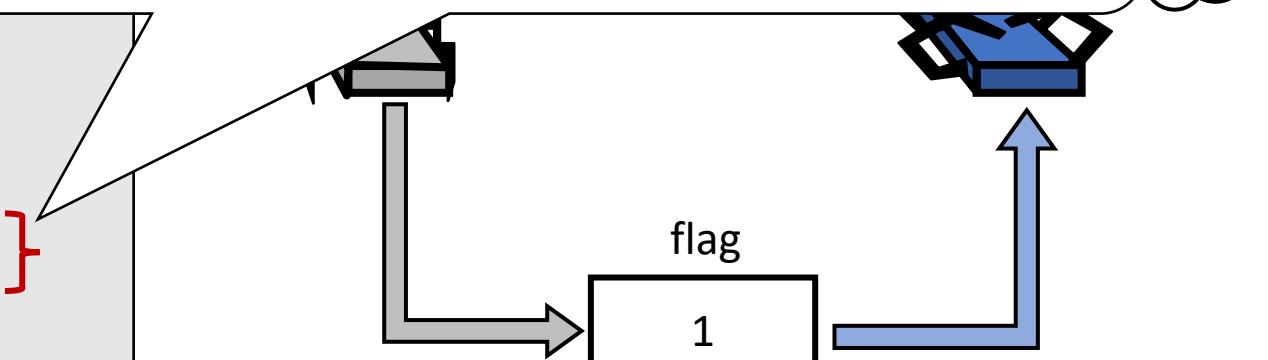


Lost wake-up problem

```
10 pthread_cond_t cond;  
11 int flag;  
12  
13 void func_threadA(void) {  
14     flag = 1;  
15     pthread_cond_signal(&cond)  
16 }  
17  
18 void func_threadB(void) {  
19     while (flag == 0) {  
20         pthread_cond_wait(&cond); }  
21 }  
22 }
```

Lost wake-up occurs because
checking *flag*(line 19) and falling asleep(line 20)
are **not atomic**

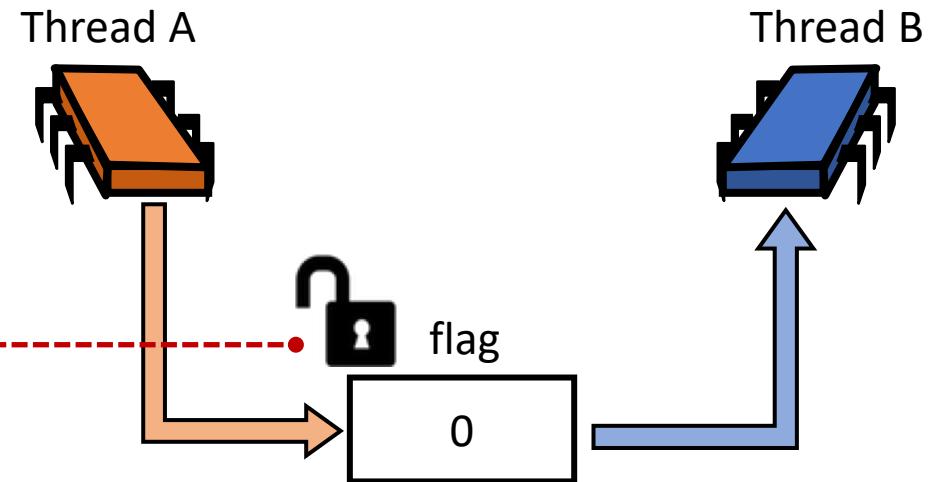
Zzz..



Lost wake-up problem

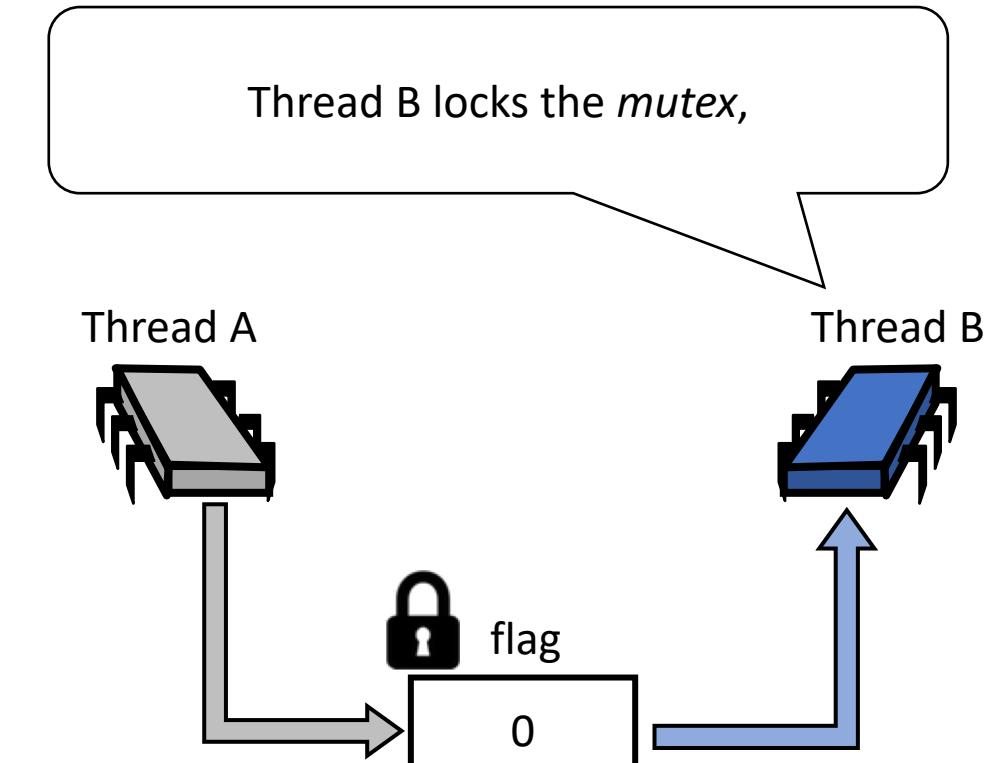
```
10 pthread_cond_t cond;
11 pthread_mutex_t mutex; •-----+
12 int flag;
13
14 void func_threadA(void) {
15     pthread_mutex_lock(&mutex);
16     flag = 1;
17     pthread_cond_signal(&cond);
18     pthread_mutex_unlock(&mutex);
19 }
20
21 void func_threadB(void) {
22     pthread_mutex_lock(&mutex);
23     while (flag == 0) {
24         pthread_cond_wait(&cond);
25     }
26     pthread_mutex_unlock(&mutex);
27 }
```

So, we are going to use a *mutex* to run those operations atomically



Lost wake-up problem

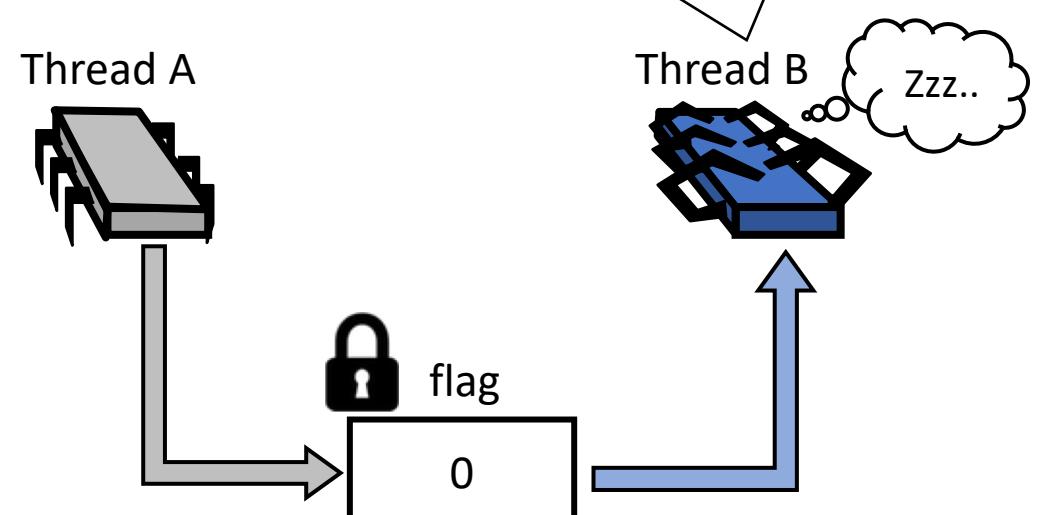
```
10 pthread_cond_t cond;
11 pthread_mutex_t mutex;
12 int flag;
13
14 void func_threadA(void) {
15     pthread_mutex_lock(&mutex);
16     flag = 1;
17     pthread_cond_signal(&cond);
18     pthread_mutex_unlock(&mutex);
19 }
20
21 void func_threadB(void) {
22     pthread_mutex_lock(&mutex);
23     while (flag == 0) {
24         pthread_cond_wait(&cond);
25     }
26     pthread_mutex_unlock(&mutex);
27 }
```



Lost wake-up problem

```
10 pthread_cond_t cond;
11 pthread_mutex_t mutex;
12 int flag;
13
14 void func_threadA(void) {
15     pthread_mutex_lock(&mutex);
16     flag = 1;
17     pthread_cond_signal(&cond);
18     pthread_mutex_unlock(&mutex);
19 }
20
21 void func_threadB(void) {
22     pthread_mutex_lock(&mutex);
23     while (flag == 0) {
24         pthread_cond_wait(&cond);
25     }
26     pthread_mutex_unlock(&mutex);
27 }
```

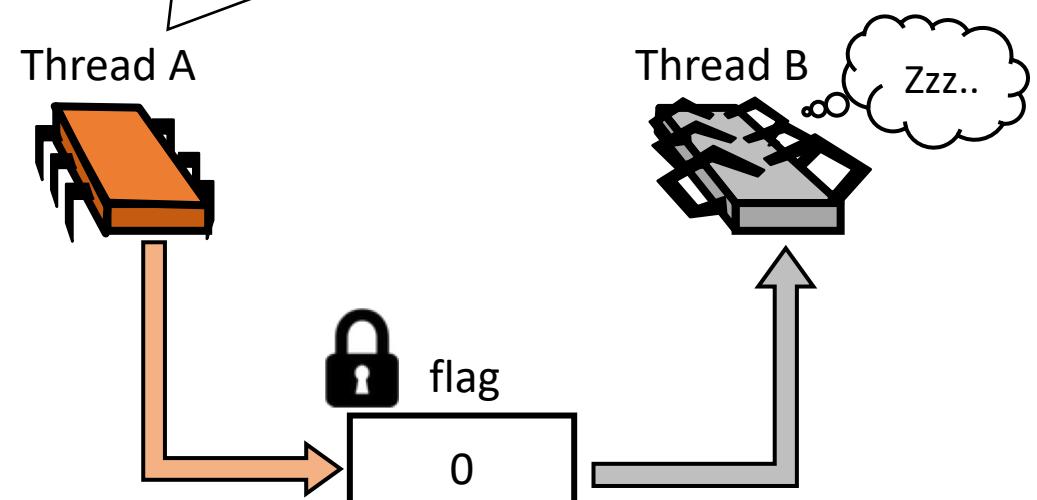
So Thread B can check the *flag* and fall asleep atomically.
Lost wake-up can not occurs, but..



Lost wake-up problem

```
10 pthread_cond_t cond;
11 pthread_mutex_t mutex;
12 int flag;
13
14 void func_threadA(void) {
15     pthread_mutex_lock(&mutex);
16     flag = 1;
17     pthread_cond_signal(&cond);
18     pthread_mutex_unlock(&mutex);
19 }
20
21 void func_threadB(void) {
22     pthread_mutex_lock(&mutex);
23     while (flag == 0) {
24         pthread_cond_wait(&cond);
25     }
26     pthread_mutex_unlock(&mutex);
27 }
```

Thread A waits for Thread B to release *mutex*.
Thread B is waiting for Thread A to wake it up,
so dead-lock have been occurred



REMIND - pthread_cond_wait

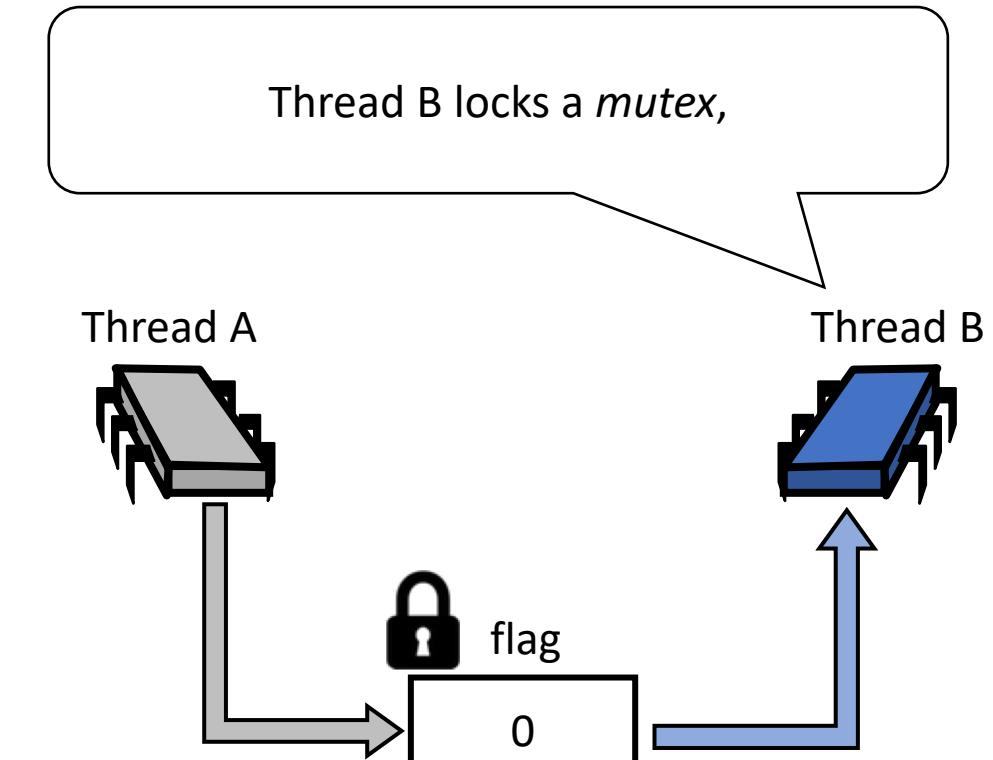
```
int pthread_cond_wait(pthread_cond_t *cond,  
                      pthread_mutex_t* mutex);
```

-
- Atomically release the **mutex** and block the calling thread on the *cond*.
 - Always return with the **mutex** acquired

@param[in] cond	Condition variable on which calling thread will block
@param[in] mutex	Mutex to be released
@return	0 if complete successfully

Lost wake-up problem

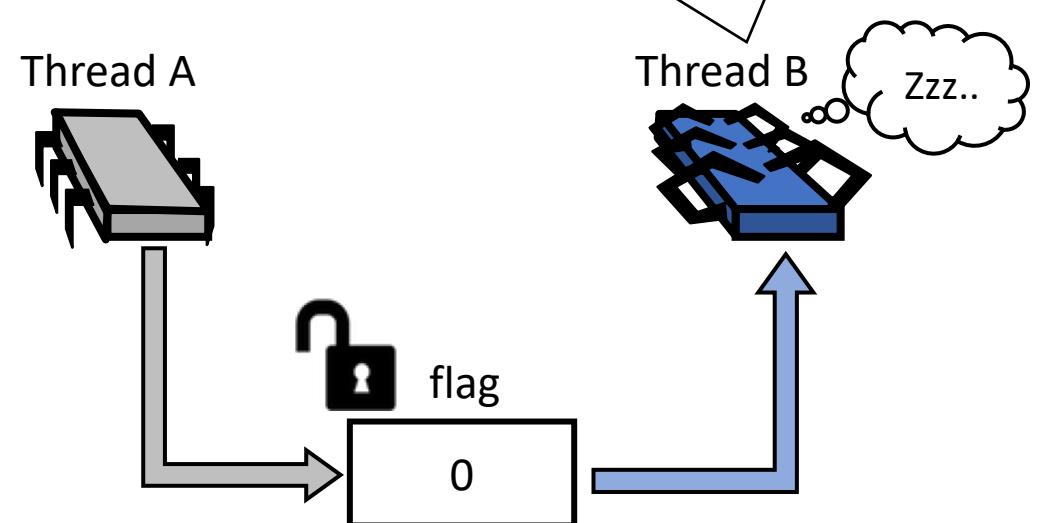
```
10 pthread_cond_t cond;
11 pthread_mutex_t mutex;
12 int flag;
13
14 void func_threadA(void) {
15     pthread_mutex_lock(&mutex);
16     flag = 1;
17     pthread_cond_signal(&cond);
18     pthread_mutex_unlock(&mutex);
19 }
20
21 void func_threadB(void) {
22     pthread_mutex_lock(&mutex);
23     while (flag == 0) {
24         pthread_cond_wait(&cond, &mutex);
25     }
26     pthread_mutex_unlock(&mutex);
27 }
```



Lost wake-up problem

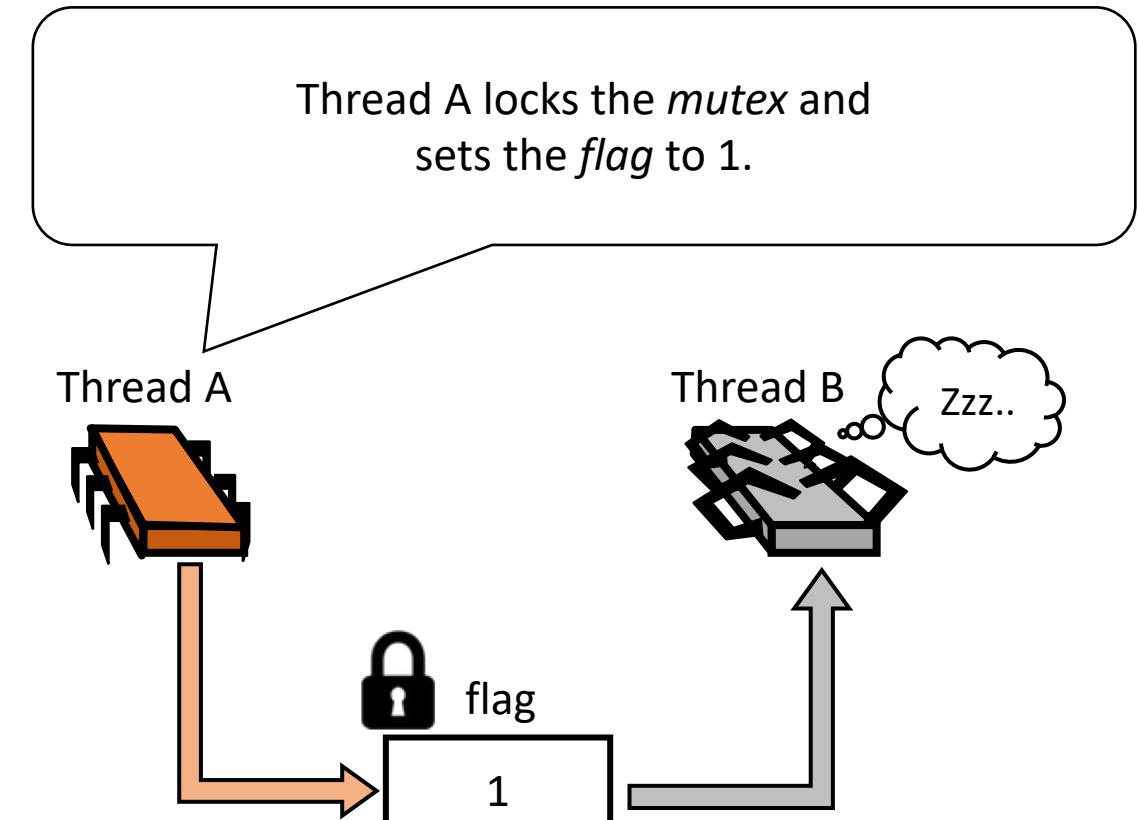
```
10 pthread_cond_t cond;
11 pthread_mutex_t mutex;
12 int flag;
13
14 void func_threadA(void) {
15     pthread_mutex_lock(&mutex);
16     flag = 1;
17     pthread_cond_signal(&cond);
18     pthread_mutex_unlock(&mutex);
19 }
20
21 void func_threadB(void) {
22     pthread_mutex_lock(&mutex);
23     while (flag == 0) {
24         pthread_cond_wait(&cond, &mutex);
25     }
26     pthread_mutex_unlock(&mutex);
27 }
```

Thread B checks the *flag*. After that, sleep on *cond* and unlock *mutex* atomically.



Lost wake-up problem

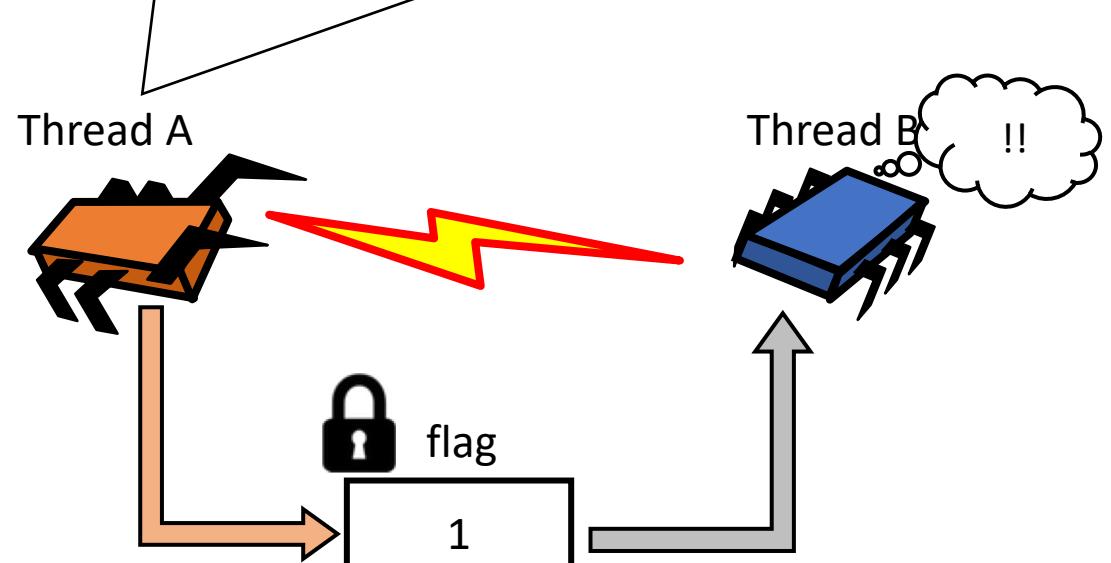
```
10 pthread_cond_t cond;
11 pthread_mutex_t mutex;
12 int flag;
13
14 void func_threadA(void) {
15     pthread_mutex_lock(&mutex);
16     flag = 1;
17     pthread_cond_signal(&cond);
18     pthread_mutex_unlock(&mutex);
19 }
20
21 void func_threadB(void) {
22     pthread_mutex_lock(&mutex);
23     while (flag == 0) {
24         pthread_cond_wait(&cond, &mutex);
25     }
26     pthread_mutex_unlock(&mutex);
27 }
```



Lost wake-up problem

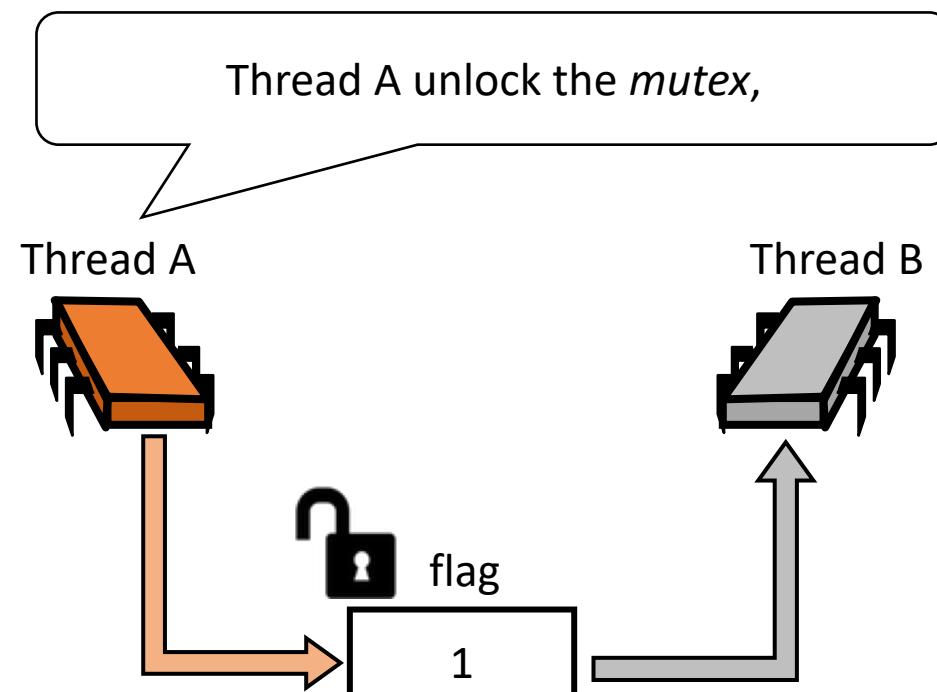
```
10 pthread_cond_t cond;
11 pthread_mutex_t mutex;
12 int flag;
13
14 void func_threadA(void) {
15     pthread_mutex_lock(&mutex);
16     flag = 1;
17     pthread_cond_signal(&cond);
18     pthread_mutex_unlock(&mutex);
19 }
20
21 void func_threadB(void) {
22     pthread_mutex_lock(&mutex);
23     while (flag == 0) {
24         pthread_cond_wait(&cond, &mutex);
25     }
26     pthread_mutex_unlock(&mutex);
27 }
```

And then, wakes up the Thread B sleeping on *cond*.
Thread B try to re-lock the *mutex* right after be awaken,
but *mutex* is still locked by Thread A now.



Lost wake-up problem

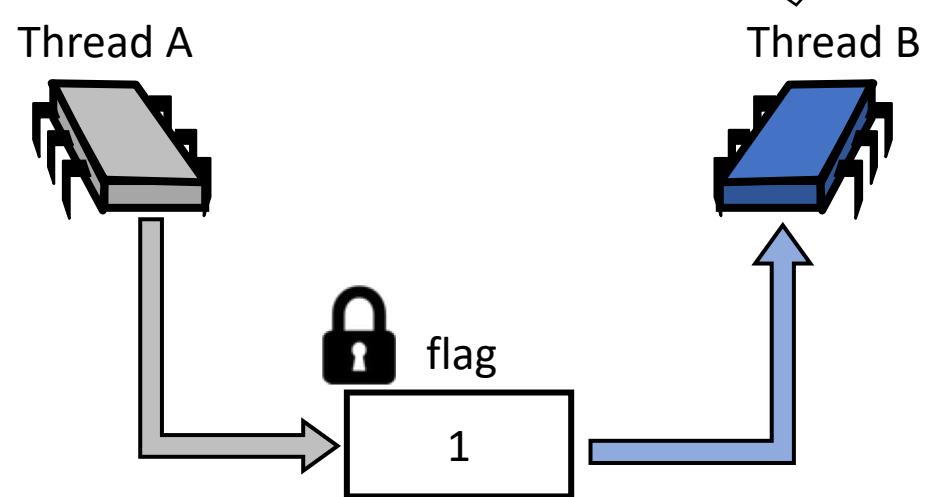
```
10 pthread_cond_t cond;
11 pthread_mutex_t mutex;
12 int flag;
13
14 void func_threadA(void) {
15     pthread_mutex_lock(&mutex);
16     flag = 1;
17     pthread_cond_signal(&cond);
18     pthread_mutex_unlock(&mutex);
19 }
20
21 void func_threadB(void) {
22     pthread_mutex_lock(&mutex);
23     while (flag == 0) {
24         pthread_cond_wait(&cond, &mutex);
25     }
26     pthread_mutex_unlock(&mutex);
27 }
```



Lost wake-up problem

```
10 pthread_cond_t cond;
11 pthread_mutex_t mutex;
12 int flag;
13
14 void func_threadA(void) {
15     pthread_mutex_lock(&mutex);
16     flag = 1;
17     pthread_cond_signal(&cond);
18     pthread_mutex_unlock(&mutex);
19 }
20
21 void func_threadB(void) {
22     pthread_mutex_lock(&mutex);
23     while (flag == 0) {
24         pthread_cond_wait(&cond, &mutex);
25     }
26     pthread_mutex_unlock(&mutex);
27 }
```

So Thread B now able to re-lock the *mutex*.
After that, checks the *flag* and go out.



Practice

- Prepare the *prime_mt.cpp* (*fixed version of prime_mt_bug.cpp*), *workload.txt* from the Piazza resource page
- Improve the code to *prime_cond.cpp*
 - Create worker threads at once
 - Wake up the threads when job is comes in
 - Put the threads to sleep after a job done
 - Compare the performance with *prime_mt* using *workload.txt*

Thank You