# Mutexes and Monitors

Daniel Zappala

CS 360 Internet Programming
Brigham Young University

# Mutexes

# Mutex

- lock that allows only one thread into a critical section

```
1  #include <pthread.h>
2
3  pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;
4
5  int pthread_mutex_lock(pthread_mutex_t *mutex);
6  int pthread_mutex_trylock(pthread_mutex_t *mutex);
7  int pthread_mutex_unlock(pthread_mutex_t *mutex);
```

- must initialize the mutex first
- `pthread_mutex_lock()` will block if mutex is already locked
- `pthread_mutex_trylock()` will return EBUSY if mutex is locked

▸ GitHub

# Don’t Use Busy Waiting!

# Busy Waiting

```
1  while running {
2    c = NULL;
3    pthread_mutex_lock(&mutex);
4    if queue.not_empty() {
5      c = queue.dequeue();
6    }
7    pthread_mutex_unlock(&mutex);
8    if c {
9      /* handle connection */
10   }
11 }
```

- must busy wait until a connection is available
- wastes CPU time on a server that does not handle many connections

# Condition Variables

# Condition Variables

```
1  #include <pthread.h>
2  pthread_cond_t cond = PTHREAD_COND_INITIALIZER;
3
4  int pthread_cond_wait(pthread_cond_t *cond,
5                        pthread_mutex_t *mutex);
6
7  int pthread_cond_signal(pthread_cond_t);
```

- must initialize the condition variable first
- `pthread_cond_wait()` will block until the condition is signaled; the thread now owns the mutex as well
- need a corresponding `pthread_cond_signal()` to wake up

# Using Condition Variables

```
1  while running {
2    c = NULL;
3    pthread_mutex_lock(&mutex);
4    while queue.empty() {
5      pthread_cond_wait(&cond,&mutex);
6    }
7    c = queue.dequeue();
8    pthread_mutex_unlock(&mutex);
9    /* handle connection */
10 }
```

- process inserting into queue should signal condition when queue goes from empty to having at least one item
- **must re-check queue status when conditional wait returns**
- no guarantee that queue will be empty when you return

# Timed Wait and Broadcast Signals

```
1  #include <pthread.h>
2
3  int pthread_cond_timedwait(pthread_cond_t *cond,
4                             pthread_mutex_t *mutex,
5                             const struct timespec *abstime);
6
7  int pthread_cond_broadcast(pthread_cond_t *cond);
```

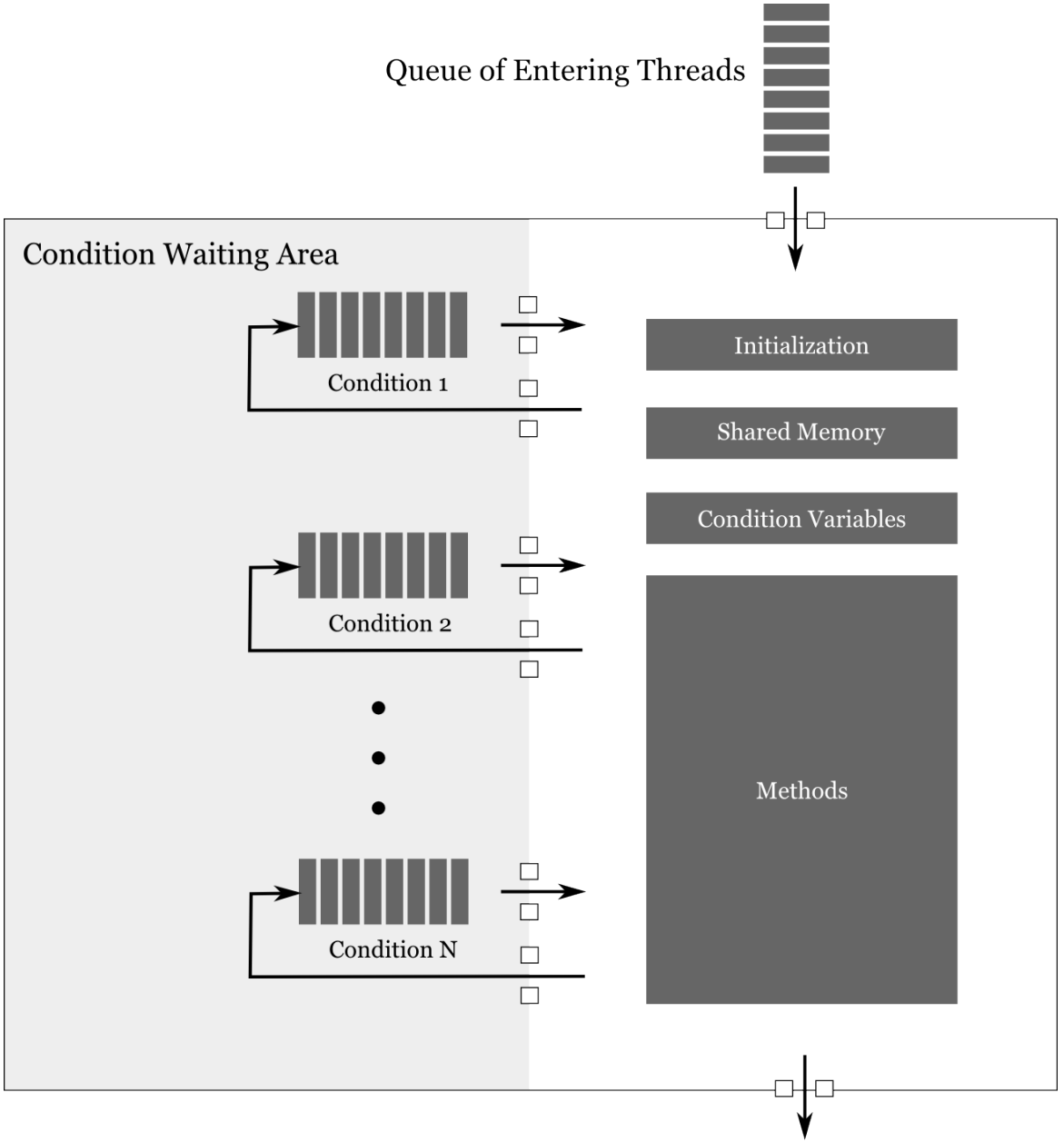
- `pthread_cond_timedwait()` needs an absolute time; use `clock_gettime()` and add the length of time you want to wait
- `pthread_cond_broadcast()` wakes up all threads waiting for a signal

# Monitors

# Monitor

- difficult to get semaphores, mutexes, condition variables right
  - match wait and signal
  - put in right order
  - scattered throughout code
- monitor: programming language construct
  - equivalent functionality
  - easier to control
  - mutual exclusion constraints can be checked by the compiler
  - used in versions of Pascal, Modula, Mesa
  - Java also has a Monitor object but compliance cannot be checked at compile time

# Hoare Monitor



# Hoare Monitor

- monitor can only be entered through methods
- shared memory can only be accessed by methods
- only one process or thread in monitor at any time
- may suspend and wait on a condition variable
- like object-oriented programming with mutual exclusion added in

# Hoare Synchronization

- `cwait(c)`: suspend on condition c
- `csignal(c)`: wake up one thread waiting for condition c
  - do nothing if no threads waiting (signal is lost)
  - different from semaphore (number of signals represented in semaphore value)

# Producer Consumer with a Hoare Monitor

```
1  vector buffer;
2  condition notfull, notempty;
```

```
1  append(item) {
2    if buffer.full()
3      cwait(notfull);
4    buffer.append(item);
5    csignal(notempty);
6  }
```

```
1  take() {
2    if buffer.empty();
3      cwait(notempty);
4    item = buffer.remove();
5    csignal(notfull);
6    return item;
```

# Producer Consumer with a Hoare Monitor

*producer:*

```
1  while (True) {
2      item = produce();
3      append(item);
4  }
```

*consume:*

```
1  while (True) {
2      item = take();
3      consume(item);
4  }
```

- advantages
  - moves all synchronization code into the monitor
  - monitor handles mutual exclusion
  - programmer handles synchronization (buffer full or empty)
  - synchronization is confined to monitor, so it is easier to check for correctness
  - write a correct monitor, any thread can use it

# Lampson and Redell Monitor

- Hoare monitor requires that signaled thread must run immediately
  - thread that calls `csignal()` must exit the monitor or be suspended
  - for example, when `notempty` condition signaled, thread waiting must be activated immediately or else the condition may no longer be true when it is activated
  - usually restrict `csignal()` to be the last instruction in a method (Concurrent Pascal)
- Lampson and Redell
  - replace `csignal()` with `cnotify()`
  - `cnotify(x)` signals the condition variable, but thread may continue
  - thread at head of condition queue will run at some future time
  - must recheck the condition!
  - used in Mesa, Modula-3

# Producer Consumer with a Lampson Redell Monitor

```
1  vector buffer;
2  condition notfull, notempty;
```

```
1  append() {
2    while buffer.full()
3      cwait(notfull);
4    buffer.append(item);
5    cnotify(notempty);
6  }
```

```
1  take() {
2  while buffer.empty()
3    cwait(notempty);
4    item = buffer.remove();
5    cnotify(notfull);
6    return item;
7  }
```

# Lampson Redell Advantages

- allows processes in waiting queue to awaken periodically and reenter monitor, recheck condition
  - prevents starvation
- can also add `cbroadcast(x)`: wake up all processes waiting for condition
  - for example, append variable block of data, consumer consumes variable amount
  - for example, memory manager that frees $k$ bytes, wake all to see who can go with $k$ more bytes
- less prone to error
  - process always checks condition before doing work

# What Can You Do?

- emulate a Lampson Redell Monitor with semaphores
  - create a class with private data only
  - use the same semaphore to protect all class methods
  - use semaphores to replace `cwait()` and `cnotify()`
- this creates a thread-safe class