Lecture 7
Mutual Exclusion
• We’re exploring the layers of an application running on top of multicore hardware.

- Task-oriented application
- Concurrent Objects
- Tasks and synchronization primitives
Today’s Agenda

• Homework 1
• Lab 0
• Concrete Pthread Creation
• A Simple Mutual Exclusion Algorithm
Homework 1 Q1

Explain the difference between concurrency and parallelism with an example: if an operating system is executing three long-running programs, how would its scheduler execute the programs concurrently on one core, concurrently on three cores, or in parallel on three cores? Comment on running the programs in parallel on one core.

Concurrently on One Core
Concurrently on Three Cores
Parallel on Three Cores
Parallel on One Core
Homework 1 Q1

Explain the difference between concurrency and parallelism with an example: if an operating system is executing three long-running programs, how would its scheduler execute the programs concurrently on one core, concurrently on three cores, or in parallel on three cores? Comment on running the programs in parallel on one core.

Concurrently on One Core
Explain the difference between concurrency and parallelism with an example: if an operating system is executing three long-running programs, how would its scheduler execute the programs concurrently on one core, concurrently on three cores, or in parallel on three cores? Comment on running the programs in parallel on one core.

**Concurrently on Three Cores**
Homework 1 Q1

Explain the difference between concurrency and parallelism with an example: if an operating system is executing three long-running programs, how would its scheduler execute the programs concurrently on one core, concurrently on three cores, or in parallel on three cores? Comment on running the programs in parallel on one core.

Concurrently on Three Cores

![Diagram of concurrent execution on three cores]
Homework 1 Q1

Explain the difference between concurrency and parallelism with an example: if an operating system is executing three long-running programs, how would its scheduler execute the programs concurrently on one core, concurrently on three cores, or in parallel on three cores? Comment on running the programs in parallel on one core.

Parallel on Three Cores

<table>
<thead>
<tr>
<th>PU</th>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td>PU</td>
<td>2</td>
</tr>
<tr>
<td>PU</td>
<td>3</td>
</tr>
</tbody>
</table>
Homework 1 Q1

Explain the difference between concurrency and parallelism with an example: if an operating system is executing three long-running programs, how would its scheduler execute the programs concurrently on one core, concurrently on three cores, or in parallel on three cores? Comment on running the programs in parallel on one core.

Parallel on One Core

Impossible – requires multiple cores. Would have accepted answers involving hyperthreading, but we should still consider those multiple logical cores.
You are implementing a server that sends responses to requests, and requires very small amounts of computation to handle each request. Your application requires many clients and many servers to work together. Which of the programming models discussed in class are you likely to use to model this application, and why?
Homework 1 Q2

You are implementing a server that sends responses to requests, and requires very small amounts of computation to handle each request. Your application requires many clients and many servers to work together. Which of the programming models discussed in class are you likely to use to model this application, and why?

• Small amount of computation
• Lots of communication
• Asynchronous messaging

Best match: LogP
Apply Amdahl’s law to compute the speedup for the following program if you have (a) 1, (b) 2, (c) 4, (d) 8, (e) 12, (f) 16, and (g) ∞ CPUs. In the following diagram, S portions are sequential and P are parallelizable.
Homework 1 Q3

Apply Amdahl’s law to compute the speedup for the following program if you have (a) 1, (b) 2, (c) 4, (d) 8, (e) 12, (f) 16, and (g) ∞ CPUs. In the following diagram, S portions are sequential and P are parallelizable.

<table>
<thead>
<tr>
<th>S</th>
<th>P</th>
<th>S</th>
<th>P</th>
<th>P</th>
<th>S</th>
</tr>
</thead>
<tbody>
<tr>
<td>15%</td>
<td>18%</td>
<td>12%</td>
<td>15%</td>
<td>30%</td>
<td>10%</td>
</tr>
</tbody>
</table>

Serial: 37%
Parallel: 63%

Order doesn’t matter!
Apply Amdahl’s law to compute the speedup for the following program if you have (a) 1, (b) 2, (c) 4, (d) 8, (e) 12, (f) 16, and (g) ∞ CPUs. In the following diagram, S portions are sequential and P are parallelizable.

<table>
<thead>
<tr>
<th>S</th>
<th>P</th>
<th>S</th>
<th>P</th>
<th>P</th>
<th>S</th>
</tr>
</thead>
<tbody>
<tr>
<td>15%</td>
<td>18%</td>
<td>12%</td>
<td>15%</td>
<td>30%</td>
<td>10%</td>
</tr>
</tbody>
</table>

Serial: 37%
Parallel: 63%

\[
speedup = \frac{1}{F + \frac{1-F}{P}} = \frac{1}{0.37 + \frac{0.63}{1}} = 1
\]

(a) 1 CPU
Apply Amdahl’s law to compute the speedup for the following program if you have (a) 1, (b) 2, (c) 4, (d) 8, (e) 12, (f) 16, and (g) ∞ CPUs. In the following diagram, S portions are sequential and P are parallelizable.

<table>
<thead>
<tr>
<th>S</th>
<th>P</th>
<th>S</th>
<th>P</th>
<th>P</th>
<th>S</th>
</tr>
</thead>
<tbody>
<tr>
<td>15%</td>
<td>18%</td>
<td>12%</td>
<td>15%</td>
<td>30%</td>
<td>10%</td>
</tr>
</tbody>
</table>

Serial: 37%
Parallel: 63%

\[
speedup = \frac{1}{F + \frac{1 - F}{P}} = \frac{1}{0.37 + \frac{0.63}{2}} = 1.46
\]

(b) 2 CPUs
Apply Amdahl’s law to compute the speedup for the following program if you have (a) 1, (b) 2, (c) 4, (d) 8, (e) 12, (f) 16, and (g) ∞ CPUs. In the following diagram, S portions are sequential and P are parallelizable.

<table>
<thead>
<tr>
<th>S</th>
<th>P</th>
<th>S</th>
<th>P</th>
<th>P</th>
<th>S</th>
</tr>
</thead>
<tbody>
<tr>
<td>15%</td>
<td>18%</td>
<td>12%</td>
<td>15%</td>
<td>30%</td>
<td>10%</td>
</tr>
</tbody>
</table>

Serial: 37%
Parallel: 63%

\[
\text{speedup} = \frac{1}{F + \frac{1-F}{P}} = \frac{1}{0.37 + \frac{0.63}{4}} = 1.90
\]

(c) 4 CPUs
Apply Amdahl’s law to compute the speedup for the following program if you have (a) 1, (b) 2, (c) 4, (d) 8, (e) 12, (f) 16, and (g) ∞ CPUs. In the following diagram, $S$ portions are sequential and $P$ are parallelizable.

<table>
<thead>
<tr>
<th>$S$</th>
<th>$P$</th>
<th>$S$</th>
<th>$P$</th>
<th>$P$</th>
<th>$S$</th>
</tr>
</thead>
<tbody>
<tr>
<td>15%</td>
<td>18%</td>
<td>12%</td>
<td>15%</td>
<td>30%</td>
<td>10%</td>
</tr>
</tbody>
</table>

Serial: 37%
Parallel: 63%

$$speedup = \frac{1}{\frac{1}{F} + \frac{1-F}{P}} = \frac{1}{0.37 + \frac{0.63}{8}} = 2.23$$

(d) 8 CPUs
Homework 1 Q3

Apply Amdahl’s law to compute the speedup for the following program if you have (a) 1, (b) 2, (c) 4, (d) 8, (e) 12, (f) 16, and (g) $\infty$ CPUs. In the following diagram, S portions are sequential and P are parallelizable.

<table>
<thead>
<tr>
<th>S</th>
<th>P</th>
<th>S</th>
<th>P</th>
<th>P</th>
<th>S</th>
</tr>
</thead>
<tbody>
<tr>
<td>15%</td>
<td>18%</td>
<td>12%</td>
<td>15%</td>
<td>30%</td>
<td>10%</td>
</tr>
</tbody>
</table>

Serial: 37%
Parallel: 63%

\[
\text{speedup} = \frac{1}{F + \frac{1-F}{P}} = \frac{1}{0.37 + \frac{0.63}{12}} = 2.37
\]

(e) 12 CPUs
Homework 1 Q3

Apply Amdahl’s law to compute the speedup for the following program if you have (a) 1, (b) 2, (c) 4, (d) 8, (e) 12, (f) 16, and (g) ∞ CPUs. In the following diagram, S portions are sequential and P are parallelizable.

<table>
<thead>
<tr>
<th>S</th>
<th>P</th>
<th>S</th>
<th>P</th>
<th>P</th>
<th>S</th>
</tr>
</thead>
<tbody>
<tr>
<td>15%</td>
<td>18%</td>
<td>12%</td>
<td>15%</td>
<td>30%</td>
<td>10%</td>
</tr>
</tbody>
</table>

Serial: 37%
Parallel: 63%

\[
speedup = \frac{1}{F + \frac{1 - F}{P}} = \frac{1}{0.37 + \frac{0.63}{16}} = 2.44
\]

(f) 16 CPUs
Homework 1 Q3

Apply Amdahl’s law to compute the speedup for the following program if you have (a) 1, (b) 2, (c) 4, (d) 8, (e) 12, (f) 16, and (g) ∞ CPUs. In the following diagram, S portions are sequential and P are parallelizable.

<table>
<thead>
<tr>
<th>S</th>
<th>P</th>
<th>S</th>
<th>P</th>
<th>P</th>
<th>S</th>
</tr>
</thead>
<tbody>
<tr>
<td>15%</td>
<td>18%</td>
<td>12%</td>
<td>15%</td>
<td>30%</td>
<td>10%</td>
</tr>
</tbody>
</table>

Serial: 37%
Parallel: 63%

\[
\text{speedup} = \frac{1}{F + \frac{1-F}{P}} = \frac{1}{0.37 + \frac{0.63}{\infty}} = 2.70
\]

(g) ∞ CPUs
What are hazards in a CPU pipeline? Why and how do they reduce the effective IPC (instructions per clock)? Feel free to explain whatever concepts of how pipelines work are relevant.
Homework 1 Q4

What are hazards in a CPU pipeline? Why and how do they reduce the effective IPC (instructions per clock)? Feel free to explain whatever concepts of how pipelines work are relevant.

- Dependencies between sequential instructions
- Can stall the pipeline while instruction $n+1$ waits for the result of instruction $n$
- Reduces IPC because now fewer than 1 instruction finishing per clock cycle
Draft the skeleton of a family of C++ classes (declarations, no definitions) that could be used to represent struct and class concepts in C++. For example, you might want an Object class that is the parent of the Struct and Class classes, which would contain other things. Think about storing the things that can go in either one (member variables, methods), and what we need to know (type or return type, visibility, arguments, etc.). An exhaustive representation of every relevant C++ is not necessary, but show you understand the concepts.

Was intended to be not easy, but straightforward.
struct Method {
    Visibility visibility;
    Type returnType;
    std::string name;
    bool const;
    std::vector<MethodArg> args;
};

struct Field {
    Visibility visibility;
    Type type;
    std::string name;
    bool const;
};

struct MethodArg {
    Type type;
    std::string name;
    bool const;
};

class Object {
    std::vector<Field> fields_;  
    std::vector<Method> methods_;  
};

class Struct : public Object { }

class Class : public Object { }
### Lab 0: Templated Multimap

**Vector of Pairs of (K, V):**

<table>
<thead>
<tr>
<th>Key</th>
<th>Val</th>
</tr>
</thead>
<tbody>
<tr>
<td>Key1</td>
<td>Val1</td>
</tr>
<tr>
<td>Key2</td>
<td>Val2</td>
</tr>
<tr>
<td>Key1</td>
<td>Val3</td>
</tr>
<tr>
<td>Key1</td>
<td>Val4</td>
</tr>
<tr>
<td>Key3</td>
<td>Val5</td>
</tr>
<tr>
<td>Key4</td>
<td>Val6</td>
</tr>
<tr>
<td>Key3</td>
<td>Val7</td>
</tr>
</tbody>
</table>

**Vector of Pairs of (K, vector of V):**

<table>
<thead>
<tr>
<th>Key</th>
<th>Vector of Vals</th>
</tr>
</thead>
<tbody>
<tr>
<td>Key1</td>
<td>Val1, Val3, Val4</td>
</tr>
<tr>
<td>Key2</td>
<td>Val2</td>
</tr>
<tr>
<td>Key3</td>
<td>Val5, Val7</td>
</tr>
</tbody>
</table>
template <class K, class V>
class Multimap {
    typedef std::pair<K, V> kv_pair;
    std::vector<kv_pair> data_; 
    public:
        bool insert(const K& key, const V& val) {
            data_.push_back(std::make_pair(key, val));
            return true;
        }
        bool find(const K& key, std::list<V>& values) {
            for(const auto& item : data_) {
                if (key == item.first) {
                    values.push_back(item.second);
                }
            }
            return values.size() > 0;
        }
};
Lab 0: Templated Multimap: Vector of Pairs of (K, vector of V)

template <class K, class V>
class Multimap {
    typedef std::pair<K, std::vector<V>> kv_pair;
    std::vector<kv_pair> data_

public:
    bool insert(const K& key, const V& val) {
        for(auto& item : data_) {
            if (item.first == key) {
                item.second.push_back(val);
                return true;
            }
        }
        data_.push_back(std::make_pair(key, {val}));
        return true;
    }

    bool find(const K& key, std::list<V>& values) {
        for(const auto& item : data_) {
            if (key == item.first) {
                values = item.second;
                return true;
            }
        }
        return false;
    }
};
Concrete Thread Creation

```cpp
#include <iostream>
#include <thread>
#include <memory>               // unique_ptr
#include <exception>

const size_t NUM_THREADS = 4;

void printHello(const size_t thread_id) {
    printf("Hello World! It's me, thread #%ld\n", thread_id);
    return;
}

int main(int argc, char *argv[]) {
    std::unique_ptr<std::thread> threads[NUM_THREADS];
    for(size_t t = 0; t < NUM_THREADS; t++) {
        printf("In main: creating thread %ld\n", t);
        threads[t].reset(new std::thread(printHello, t));

        threads[t]->detach();           // try join in a loop below!
    }
    //for(size_t t = 0; t < NUM_THREADS; t++) {
    //    threads[t]->join();
    //}
    return 0;
}
```
The literature on mutual exclusion algorithms is extensive.

In practice, mutual exclusion is implemented with some form of hardware and OS support (coming soon!)

But we’ll first develop our algorithmic and correctness evaluation skills by looking at some very elegant solutions that use only shared memory.
Understanding Mutices: Review

- Mutual exclusion algorithms will provide two methods, `lock()` and `unlock()`, that allow us to mark the beginning and the end of critical sections in our code.
  - `lock()` will block the caller until there is no other thread in the critical section.
  - `unlock()` would allow other thread in the critical section if there’s one waiting.
Peterson’s Algorithm

- Mutual exclusion solution in “three” lines of code.

```c
int victim;
bool flag[2] = { false, false };

void lock(void) {
    int me = my_tid % 2
    int other = 1 - me
    flag[me] = true
    victim = me
    while (flag[other] &&
        victim == me) {
        no-op;
    }
}

void unlock(void) {
    int me = my_tid % 2
    flag[me] = false
}
```
Uncontested Case

- Solved in two simple tests and no wait.

```c
int victim = 0;
bool flag[2] = { true, false };

void lock(void) {
    int me = my_tid % 2
    int other = 1 - me
    flag[me] = true
    victim = me
    while (flag[other] &&
        victim == me) {
        no-op;
    }
}

void unlock(void) {
    int me = my_tid % 2
    flag[me] = false
}
```

Contested Case

- Thread 1 will busy-wait until Thread 0 executes unlock

```c
int victim = 1;
bool flag[2] = { true, true };

void lock(void) {
    int me = my_tid % 2
    int other = 1 - me
    flag[me] = true
    victim = me
    while (flag[other] &&
        victim == me) {
        no-op;
    }
}

void unlock(void) {
    int me = my_tid % 2
    flag[me] = false
}
```
Simultaneous Case

- Assumes both cannot write to same mem addr at once.

```c
int victim = ?;
bool flag[2] = { true, true };

void lock(void) {
    int me = my_tid % 2
    int other = 1 - me
    flag[me] = true
    victim = me
    while (flag[other] &&
          victim == me) {
        no-op;
    }
}

void unlock(void) {
    int me = my_tid % 2
    flag[me] = false
}
```
Analyzing Peterson’s Algorithm

• Correct mutex needs three criteria
  1. Mutual exclusion
  2. Progress (deadlock-free)
  3. Bounded waiting (fairness: starvation-free)
Analyzing Peterson’s Algorithm

• Does Peterson’s Algorithm guarantee mutual exclusion?
• Assume that both threads could pass the tests:
  while (flag[other] && victim == me)
• This would have meant that each would have set the
  victim to be itself and each would have seen the victim as
  the other thread
• Is this possible?
Analyzing Peterson’s Algorithm

• Is it starvation free? Fair?
• If a thread `unlock()`s then it would set victim to be itself
• The contenting thread would have the chance to pass `lock()` then.
• Now, nothing prevents the `unlock()` thread from going ahead and trying a `lock()` again.
  • But in that case, that thread changes the victim to be itself before getting in the loop.
Deadlock on Simultaneous Case

- Using flag[] alone doesn’t work
- Could it be simpler? Instead of flag[] and victim, let’s have one or the other

```c
bool flag[2] = { true, true };  

void lock(void) {
    int me = my_tid % 2
    int other = 1 - me
    flag[me] = true
    while (flag[other]) {
        no-op;
    }
}

void unlock(void) {
    int me = my_tid % 2
    flag[me] = false
}
```
Deadlock on the Uncontested Case

- Peterson’s is as simple as it gets.
- Now with ‘victim’ only

```c
void lock(void) {
    int me = my_tid % 2
    int other = 1 - me
    victim = me
    while (victim == me) {
        no-op;
    }
}

void unlock(void)
```

- Mutices
- Semaphores
- Cond Vars
Observations

- We’ll be using the reasoning we developed here in other algorithms.
- Peterson’s lock is important because it was arguably the first one to show 2-thread mutual exclusion can be solved in a simple way.
- It makes several assumptions, though, and these turn out not to be so in practice:
  - Number of threads is known a priori
  - Program execution is strictly sequential and each instruction is atomic
- The algorithm uses busy waiting
Filter Algorithm

- Peterson’s Algorithm generalized for \( n \geq 2 \) threads
- Adds \( n-1 \) waiting rooms ("levels")
  - Every thread is not in any room (or you could think of it as "level 0") when it doesn’t want the lock
  - Every thread enters algorithm at level 1
  - Must reach level \( n-1 \) to have the lock
- One fewer thread can be at each level
  - Level 1: Up to \( n-1 \) threads
  - Level 2: Up to \( n-2 \) threads
  - Level \( n-2 \): Up to 2 threads
  - Level \( n-1 \): Up to 1 thread
Filter Algorithm: New State

- Per-thread: `int level;`
  - The current level that this thread *wants* to enter (not necessarily the one that it has succeeded in entering)
  - Must be able to read every other thread’s level variable (CREW)

- Global: `int victim[n];`
  - Could also be called `last_to_enter`: last thread to enter each level
  - Replaces flag array
  - CRCW
Filter Algorithm: Pseudocode

def lock(thread i):
    for L in range(0, n-1):
        L → level[i]
        i → victim[L]
        while victim[L] = i and there exists some k ≠ i, such that level[k] ≥ L:
            wait()

def unlock(thread i):
    -1 → level[i]
Filter Algorithm: Pseudocode

def lock(thread i):
    for L in range(1, n-1):
        L \rightarrow level[i]
        i \rightarrow victim[L]
        while victim[L] = i and there exists some k \neq i, such that level[k] \geq L:
            wait()

Must proceed through every level L, 1 to \(n - 1\), to get the lock, no matter what.
Filter Algorithm: Pseudocode

```python
def lock(thread i):
    for L in range(1, n-1):
        L → level[i]
        i → victim[L]
        while victim[L] = i and there exists some k ≠ i, such that level[k] ≥ L:
            wait()
```

Make globally visible that this thread \(i\) \textit{intends} to enter level \(L\) when it’s able
def lock(thread i):
    for L in range(1, n-1):
        L \rightarrow \text{level}[i]
        i \rightarrow \text{victim}[L]
        while \text{victim}[L] = i and there exists some k \neq i, such that \text{level}[k] \geq L:
            wait()

Update the victim for this level L to this thread i (i.e., this i is the last thread to attempt to enter level L now).
Filter Algorithm: Pseudocode

```python
def lock(thread i):
    for L in range(1, n-1):
        L → level[i]
        i → victim[L]
        while victim[L] = i and there exists some k ≠ i, such that level[k] ≥ L:
            wait()
```

Can’t proceed until either:
1. Every thread has left the higher, more restricted waiting rooms, *or*
2. Another thread attempts to joins the current level after us
Filter Algorithm: Example

- Threads A, B, and C (of 4 total threads) want to lock

```c
int victim[3] = {-1, -1, -1};
int level[4] = { 0, 0, 0, 0};
```
Filter Algorithm: Example

• B wants to add itself to level 1

```
int victim[3] = { 1, -1, -1};
int level[4] = { 0, 1, 0, 0};
```
Filter Algorithm: Example

• B succeeds in joining level 1

```
int victim[3] = { 1, -1, -1};
int level[4] = { 0, 1, 0, 0};
```
Filter Algorithm: Example

- No thread at level 1 or 2, so B tries to advance, advances

```c
int victim[3] = { 1, 1, -1};
int level[4] = { 0, 2, 0, 0};
```
Filter Algorithm: Example

- C tries to add itself to level 1, can’t proceed because of B

```c
int victim[3] = { 2, 1, -1};
int level[4] = { 0, 2, 1, 0};
```
• A tries to add itself to level 1. This frees C to enter level 1.

```c
int victim[3] = { 0, 1, -1};
int level[4] = { 1, 2, 1, 0};
```
Filter Algorithm: Example

- Because victim[0] is no longer 2, C tries to advance to level 2

```
int victim[3] = { 0, 2, -1};
int level[4] = { 1, 2, 2, 0};
```
Filter Algorithm: Example

- Because victim[1] is no longer 1 (even though there’s another thread attempting to join a level $\geq 2$), B advances to level 3, and has the lock.

```
int victim[3] = { 0, 2, 1};
int level[4] = { 1, 3, 2, 0};
```
• Once B finishes, C can proceed down to level 3 and get the lock

```c
int victim[3] = { 0, 2, 2};
int level[4] = { 1, 0, 3, 0};
```
Filter Algorithm: Example

- Once C finishes, A can proceed down to level 3 and get the lock

```c
int victim[3] = { 0, 0, 0};
int level[4] = { 3, 0, 0, 0};
```
Filter Algorithm: Proving Mutual Exclusion

Claim: if all threads start at level $L = 0$, at most $n - L$ threads can actually enter level $L$, and at most 1 thread can enter level $n - 1$.

1. Let $L \leq n - 1$

2. At most $n - L' \leq n - (L + 1)$ threads can concurrently proceed to any level $L' \geq L + 1$

3. Specifically:
   - At most one thread is at a level $L' \geq L$, or
   - A thread is waiting at each level 0, ..., $L - 1$

4. In each case in (3), (2) holds

5. Therefore, if $L = n - 2$, At most $n - L' = n - (n - 2 + 1) = 1$ thread can proceed to $L = n - 1$ (and get the lock)
Filter Algorithm: Proving Starvation-Free

- Consider a thread $i$ waiting to go to level $L \geq 0$ (having already announced their intention to enter $L$).

- If other threads keep on entering/level their critical sections, eventually a thread $j \neq i$ wants to enter $L$ en route to level $n - 1$, sets $j \rightarrow \text{victim}[L]$.

- Then thread $i$ must be able to proceed to level $L$ and then attempt to enter $L + 1$ before thread $j$. 
Filter Algorithm: Why Have victim?

- Let any thread $i$ immediately grab the lock if $\text{level}[j] < L$ for all $j \neq i$. We have 3 threads, thus 2 levels.

```python
def lock(thread i):
    for L in range(0, n-1):
        L \rightarrow level[i]
        while there exists some $k \neq i$, such that $\text{level}[k] \geq L$:
            wait()
```

```python
int level[3] = {0, 0, 0};
```
Filter Algorithm: Why Have victim?

- A and B both attempt to add themselves to level 1

```python
def lock(thread i):
    for L in range(0, n-1):
        L → level[i]
        while there exists some k ≠ i, such that level[k] ≥ L:
            wait()
```

```python
int level[3] = { 1, 1, 0};
```
Filter Algorithm: Why Have victim?

- Neither A nor B can proceed into level 1: deadlock

```python
def lock(thread i):
    for L in range(0, n-1):
        L → level[i]
        while there exists some k ≠ i, such that level[k] ≥ L:
            wait()
```

```plaintext
int level[2] = { 1, 1, 0};
```
Conclusion

- Ideal Mutual Exclusion
  - Deadlock-free
    - Some thread eventually gets the lock
  - Starvation-free
    - Every (waiting) thread eventually gets the lock

- Working towards mutual exclusion
  - Peterson’s Algorithm
  - Filter Algorithm

- Real-Life Mutices
  - Peterson’s & Filter need memory barriers
  - More efficient approaches
    - CPUs have building blocks for mutual exclusion (TAS, CAS)
    - CPUs have atomic instructions for building primitives