

# *n*-way Mutual Exclusion

EECS 3/495 “Rust”

Spring 2017

## Filter algorithm for $n$ threads

```
template <int N>
class Filter_lock : public Lock_base
{
    int level_[N] = {};
    int waiting_[N];

    bool exists_competition(int level)
    {
        for (auto k : thread::all_ids())
            if (k != i() && level_[k] >= level)
                return true;
        return false;
    }
    :
}
}
```

```

template <int N>
class Filter_lock : public Lock_base
{
:
public:
    virtual void lock() override
    {
        for (int level = 1; level < N; ++level) {
            level_[i()] = level;
            waiting_[level] = i();
            while (exists_competition(level) &&
                waiting_[level] == i()) {}
        }
    }

    virtual void unlock() override
    { level_[i()] = 0; }
}

```

# Filter lock properties

- Mutual exclusion
  - ▶ By induction, one thread gets stuck in each level...
- Deadlock freedom
  - ▶ Like Peterson—only one thread can wait per level
- Starvation freedom
  - ▶ Like Peterson—every thread advances if any does

# Filter lock properties

- Mutual exclusion
  - ▶ By induction, one thread gets stuck in each level...
- Deadlock freedom
  - ▶ Like Peterson—only one thread can wait per level
- Starvation freedom
  - ▶ Like Peterson—every thread advances if any does
- Fairness?
  - ▶ No—threads can overtake others

## Bounded waiting

Idea: If thread  $A$  “starts before”  $B$ , then  $A$  enters CS before  $B$ .

# Bounded waiting

Idea: If thread  $A$  “starts before”  $B$ , then  $A$  enters CS before  $B$ .

But what is “starts before”?

# Bounded waiting

Divide *lock()* operation into two intervals:

- Doorway ( $D_A$ ), finite steps
- Waiting ( $W_A$ ), possibly unbounded

# Bounded waiting

Divide *lock()* operation into two intervals:

- Doorway ( $D_A$ ), finite steps
- Waiting ( $W_A$ ), possibly unbounded

$r$ -Bounded Waiting Guarantee: If  $D_A^k \rightarrow D_B^j$ , then  $CS_A^k \rightarrow CS_B^{j+r}$ .

# Bounded waiting

Divide *lock()* operation into two intervals:

- Doorway ( $D_A$ ), finite steps
- Waiting ( $W_A$ ), possibly unbounded

*r*-Bounded Waiting Guarantee: If  $D_A^k \rightarrow D_B^j$ , then  $CS_A^k \rightarrow CS_B^{j+r}$ .

“If *A* enters the doorway for the *k*th time before *B* enters it for the *j*th time, then *A*’s *k*th critical section happens before *B*’s (*j* + *r*)th critical section.”

## $r$ -Bounded waiting

- Peterson's Algorithm (for 2) has  $r = 0$  (first-come-first-served)
- Filter algorithm (for  $n$ ) has  $r = \infty$

## $r$ -Bounded waiting

- Peterson's Algorithm (for 2) has  $r = 0$  (first-come-first-served)
- Filter algorithm (for  $n$ ) has  $r = \infty$
- Bakery algorithm (for  $n$ ) has  $r = 0$  (first-come-first-served)

## Helper class: lexicographically-ordered pairs

```
template <typename A, typename B>
struct LP
{
    A x;
    B y;
};
```

```
template <typename A, typename B>
bool operator>(const LP<A, B> & p,
               const LP<A, B> & q)
{
    return p.x > q.x || (p.x == q.x && p.y > q.y);
}
```

## Bakery algorithm for $n$ threads

```
template <int N>
class Bakery_lock : public Lock_base
{
    bool flag_[N] = {false};
    int label_[N] = {0};
    int max_label_ = 0;

    bool someone_is_ahead()
    {
        for (auto k : thread::all_ids())
            if (flag_[k] && LP{label_[i()], i()} > LP{label_[k], k})
                return true;
        return false;
    }
    :
}
```

```
template <int N>
class Bakery_lock : public Lock_base
{
:
public:
    virtual void lock() override
    {
        flag_[i()] = true;
        label_[i()] = ++max_label_;
        while (someone_is_ahead()) {}
    }

    virtual void unlock() override
    { flag_[i()] = false; }
}
```

# Bakery Y2<sup>32</sup>K bug

Does overflow matter?

Bits	Does it?
16	quite
32	maybe
64	no

# Bakery lock properties

- Mutual exclusion
  - ▶ Between any two  $(\text{label}[k], k)$  pairs, one will defer to the other...
- Deadlock freedom
  - ▶ Must be some least  $(\text{label}[k], k)$  pair
- Starvation freedom
  - ▶ No thread takes the same number twice
- First-come-first-served (0-bounded waiting)
  - ▶ First through the door has lower label, goes first

# Bakery lock properties

- Mutual exclusion
  - ▶ Between any two  $(\text{label}[k], k)$  pairs, one will defer to the other...
- Deadlock freedom
  - ▶ Must be some least  $(\text{label}[k], k)$  pair
- Starvation freedom
  - ▶ No thread takes the same number twice
- First-come-first-served (0-bounded waiting)
  - ▶ First through the door has lower label, goes first
- **Practical?**
  - ▶ Have to read  $n$  variables to lock

# “Registers” (shared memory locations)

Flavors:

- Multi-reader/single-writer (flag[])
- Multi-reader/multi-writer (waiting)
- (Not that interesting: SRMW and SRSW)

# Theorem

At least  $n$  MRSW (multi-reader/single-writer) registers are needed to solve deadlock-free mutual exclusion.

# Theorem

At least  $n$  MRSW (multi-reader/**multi**-writer) registers are needed to solve deadlock-free mutual exclusion.

# Theorem

At least  $n$  MRMW (multi-reader/**multi**-writer) registers are needed to solve deadlock-free mutual exclusion.

# Theorem

At least  $n$  MRMW (multi-reader/multi-writer) registers are needed to solve deadlock-free mutual exclusion.

**Proof sketch.** For  $n = 2$ , one register is insufficient because neither thread necessarily sees the other's write. Then by induction, the record of the first thread to enter always gets obliterated by the rest.

# Summary

For deadlock-free mutual exclusion of  $n$  threads:

- Best known algorithm uses  $2n$  MRSW registers
- Lower bound for MRMW is  $n$

# Summary

For deadlock-free mutual exclusion of  $n$  threads:

- Best known algorithm uses  $2n$  MRSW registers
- Lower bound for MRMW is  $n$

$O(n)$  reads is too inefficient—we need something better, and we'll get it from the hardware

This work is licensed under a Creative Commons “Attribution-ShareAlike 3.0 Unported” license.

These slides are derived from the companion slides for *The Art of Multiprocessor Programming*, by Maurice Herlihy and Nir Shavit. Its original license reads:

*This work is licensed under a Creative Commons Attribution-ShareAlike 2.5 License.*

- **You are free:**
  - ▶ **to Share** — to copy, distribute and transmit the work
  - ▶ **to Remix** — to adapt the work
- **Under the following conditions:**
  - ▶ **Attribution.** You must attribute the work to “The Art of Multiprocessor Programming” (but not in any way that suggests that the authors of that work or this endorse you or your use of the work).
  - ▶ **Share Alike.** If you alter, transform, or build upon this work, you may distribute the resulting work only under the same, similar or a compatible license.
- *For any reuse or distribution, you must make clear to others the license terms of this work. The best way to do this is with a link to*
  - ▶ <http://creativecommons.org/licenses/by-sa/3.0/>.
- *Any of the above conditions can be waived if you get permission from the copyright holder.*
- *Nothing in this license impairs or restricts the author’s moral rights.*