Concurrency:
Multi-core Programming
& Data Processing

Lab 3

-- Atomic types & Peterson's algorithm --

Java Atomic data types

- `java.util.concurrent.atomic`
- Lock-free thread-safe programming on single variables
- Atomic equivalent for usual data types: `AtomicBoolean`, `AtomicInteger`, etc.
- Direct use of hardware primitives
- Operations:
  - main: `get()`, `set()`
  - conditional update: `compareAndSet()`
  - other utility functions: `getAndIncrement()`

Atomic vs. Volatile

- `get()` and `set()` -- same as operations on volatile variables
- … but it extends Volatile semantics with conditional update primitives
- ... and introduces Atomic arrays (i.e., array elements can be manipulated in an atomic manner)
- Attention! Declaring an array `volatile`, doesn't make each element `volatile`!

Peterson's mutual exclusion

- Generalized mutual exclusion algorithm for n threads
- As follows...

Thread	1	2	3
Level	1	1	1

```
lock() {
  for (int L = 1; L < n; L++) {
    level[i] = L;
    victim[L] = i;
    while (( exists k != i with level[k] >= L )
           && victim [L] == i ) {};
  }}
```

1. there exists a thread (k) and its not the current one (i) …

2. and that thread is at the same or bigger level(level[k]) than the current thread (L)

3. and the victim at the level where the current thread is (victim[L]) is the current thread (i)

Legend:	Variables:
n – total levels	n = 3
L – current level	L = 1
i – thread id	i = 1,2,3

```
lock() {
  for (int L = 1; L < n; L++) {
    level[i] = L;
    victim[L] = i;
    while (( exists k != i with level[k] >= L)
            && victim [L] == i ) {};
  }
}
```

Thread	1	2	3
Level	1	1	1
	level[1]=1	level[2]=1	level[3]=1

Legend:
n – total levels
L – current level
i – thread id

Variables:
n = 3
L = 1
i = 1,2,3

```
lock() {
  for (int L = 1; L < n; L++) {
    level[i] = L;
    victim[L] = i;
    while (( exists k != i with level[k] >= L)
            && victim [L] == i ) {};
  }
}
```

Thread	1	2	3
Level	1	1	1
	level[1]=1	level[2]=1 victim[1]=2	level[3]=1

Legend:
n – total levels
L – current level
i – thread id

Variables:
n = 3
L = 1
i = 2

```
lock() {
  for (int L = 1; L < n; L++) {
    level[i] = L;
    victim[L] = i;
    while (( exists k != i with level[k] >= L)
            && victim [L] == i ) {};
  }
}
```

Thread	1	2	3
Level	1	1	1
	level[1]=1	level[2]=1 victim[1]=2 **Blocked!**	level[3]=1

Legend:
n – total levels
L – current level
i – thread id

Variables:
n = 3
L = 1
i = 2

```
lock() {
  for (int L = 1; L < n; L++) {
    level[i] = L;
    victim[L] = i;
    while (( exists k != i with level[k] >= L)
              && victim [L] == i ) {};
  }
}
```

Thread	1	2	3
Level	1	1	1
	level[1]=1	level[2]=1 victim[1]=2 **Blocked!**	level[3]=1
	victim[1]=1		

Legend:
n – total levels
L – current level
i – thread id

Variables:
n = 3
L = 1
i = 1

```
lock() {
  for (int L = 1; L < n; L++) {
    level[i] = L;
    victim[L] = i;
    while (( exists k != i with level[k] >= L)
            && victim [L] == i ) {};
  }
}
```

Thread	1	2	3
Level	1	1	1
	level[1]=1	level[2]=1	level[3]=1
		victim[1]=2	
	victim[1]=1		
	Blocked!		
Level		2	

Legend:
n – total levels
L – current level
i – thread id

Variables:
n = 3
L = 1
i = 1

```
lock() {
  for (int L = 1; L < n; L++) {
    level[i] = L;
    victim[L] = i;
    while (( exists k != i with level[k] >= L)
              && victim [L] == i ) {};
  }
}
```

Thread	1	2	3
Level	1	1	1
	level[1]=1	level[2]=1	level[3]=1
		victim[1]=2	
	victim[1]=1	↓	
	Blocked!	↓	
		↓	victim[1]=3
Level		2	

Legend:
n – total levels
L – current level
i – thread id

Variables:
n = 3
L = 1
i = 3

```
lock() {
  for (int L = 1; L < n; L++) {
    level[i] = L;
    victim[L] = i;
    while (( exists k != i with level[k] >= L)
            && victim [L] == i ) {};
  }
}
```

Thread	**1**	**2**	**3**
Level	1	1	1
	level[1]=1	level[2]=1	level[3]=1
		victim[1]=2	
	victim[1]=1		
			victim[1]=3
			Blocked!
Level	2	2	

Legend:
n – total levels
L – current level
i – thread id

Variables:
n = 3
L = 1
i = 3

```
lock() {
  for (int L = 1; L < n; L++) {
    level[i] = L;
    victim[L] = i;
    while (( exists k != i with level[k] >= L)
              && victim [L] == i ) {};
  }
}
```

Thread	1	2	3
Level	1	1	1
	level[1]=1	level[2]=1	level[3]=1
		victim[1]=2	
	victim[1]=1	↓	
	↓		victim[1]=3
			Blocked!
Level	2	2	
	level[1]=2	level[2]=2	

Legend:
n – total levels
L – current level
i – thread id

Variables:
n = 3
L = 2
i = 1,2

```
lock() {
  for (int L = 1; L < n; L++) {
    level[i] = L;
    victim[L] = i;
    while (( exists k != i with level[k] >= L)
              && victim [L] == i ) {};
  }
}
```

Thread	1	2	3
Level	1	1	1
	level[1]=1	level[2]=1	level[3]=1
		victim[1]=2	
	victim[1]=1		
			victim[1]=3
			Blocked!
Level	2	2	
	level[1]=2	level[2]=2	
		victim[2]=2	

Legend:
n – total levels
L – current level
i – thread id

Variables:
n = 3
L = 2
i = 2

```
lock() {
  for (int L = 1; L < n; L++) {
    level[i] = L;
    victim[L] = i;
    while (( exists k != i with level[k] >= L)
            && victim [L] == i ) {};
  }
}
```

Thread	1	2	3
Level	1	1	1
	level[1]=1	level[2]=1	level[3]=1
		victim[1]=2	
	victim[1]=1		
			victim[1]=3
			Blocked!
Level	2	2	
	level[1]=2	level[2]=2	
		victim[2]=2	
		Blocked!	

Legend:
n – total levels
L – current level
i – thread id

Variables:
n = 3
L = 2
i = 2

```
lock() {
  for (int L = 1; L < n; L++) {
    level[i] = L;
    victim[L] = i;
    while (( exists k != i with level[k] >= L)
              && victim [L] == i ) {};
  }
}
```

Thread	1	2	3
Level	1	1	1
	level[1]=1	level[2]=1	level[3]=1
		victim[1]=2	
	victim[1]=1		
			victim[1]=3
			Blocked!
Level	2	2	
	level[1]=2	level[2]=2	
		victim[2]=2	
		Blocked!	
	victim[2]=1		

Legend:
n – total levels
L – current level
i – thread id

Variables:
n = 3
L = 2
i = 1

```
lock() {
  for (int L = 1; L < n; L++) {
    level[i] = L;
    victim[L] = i;
    while (( exists k != i with level[k] >= L)
            && victim [L] == i ) {};
  }
}
```

Thread	1	2	3
Level	1	1	1
	level[1]=1	level[2]=1	level[3]=1
		victim[1]=2	
	victim[1]=1	↓	
	↓		victim[1]=3
			Blocked!
Level	2	2	
	level[1]=2	level[2]=2	
		victim[2]=2	
	victim[2]=1	↓	
	Blocked!		
Level		**CriticalSection**	

Legend:
n – total levels
L – current level
i – thread id

Variables:
n = 3
L = 2
i = 1

Exercise: Atomic type application

- Last time: started threads manually, assigned static IDs, differentiated based on ID between master and workers
- **This time:** use thread pool, define different tasks for the master and workers
- But... ID was important for splitting the image between workers!
- Use atomic type primitives to generate a sequence of unique IDs for computing each slice, without depending on which thread executes the task
- Skeleton on Lab3/exercises (also code from lab2 – you can copy the master and worker code from there)