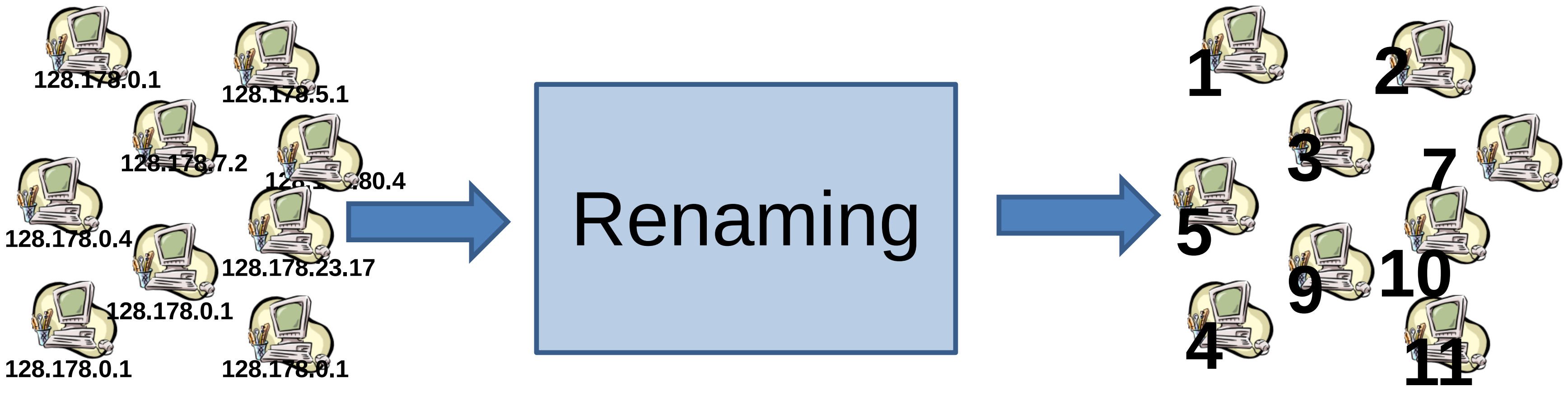
Renaming

Dan Alistarh

EPFL LPD

The Renaming Problem



- N processes, t < N might fail by crashing
- Huge initial ID’s (think IP Addresses)
- Need to get new unique ID’s from a small namespace (e.g., from 1 to N)

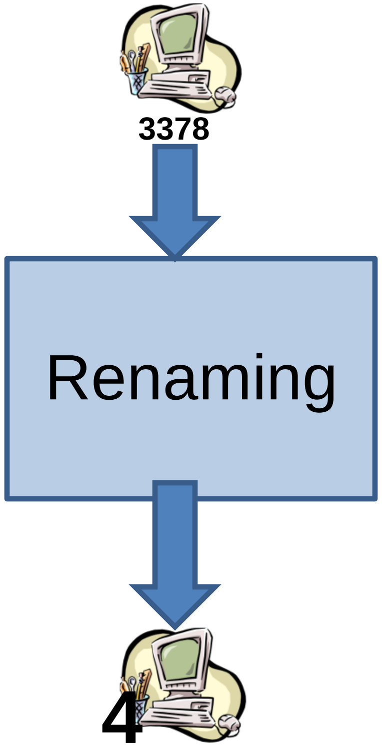
How about Shared Memory?

Example: UNIX process ID's

- Are given sequentially from 1 to MAX_PID (default 32768)
- They wrap around, and are *designed* to be unpredictable
- Commonly, shared-memory processes get *random* id's from 1 to 32767
- …so renaming is also relevant in shared memory

Why is this useful?

- Getting a small unique name is important
  - Smaller reads and writes/messages
  - Overall performance
  - Names are a natural prerequisite
- Renaming is related to:
  - Mutual exclusion
  - Test-and-set
  - Counting
  - Resource allocation



Two versions

- “Standard” renaming [Attiya et al.]
  - N = max. number of processes that may participate concurrently
  - N is known in advance
  - Target namespace of size f(N)

- Adaptive renaming [Moir et al.]
  - k is the number of processes that actually participate (contention)
  - k is unknown
  - Namespace size and performance should be f(k)

Renaming specification

- N processes start with unique identifiers from 1 to Y
- $t < N$ processes may fail by crashing
- Read-write shared memory (MWMR atomic)

Properties

1. *Termination*: Every non-faulty process returns an integer y_i
2. *Uniqueness*: for all processes p_i and p_j, $y_i \neq y_j$
3. *Namespace*: the minimal M such that all outputs y_i are in [1, 2,…,M] in all executions.
   Objective: we want to *minimize* the size of the resulting namespace.

Some notation

- *Tight renaming*:
  - Renaming into a namespace of size exactly N (or k)
- *X-renaming*:
  - Renaming into a namespace of size X

The plan for today

- Renaming definition
- Renaming algorithms
  - (2n – 1)-renaming algorithm
  - Can we do better?
  - Adaptive O(k^2)-renaming algorithm
- Renaming versus test-and-set
  - Consensus number

Uniqueness

- Assume processes *p* and *q* get the same name ***s***
- Let $\{<x_1, s_1>, \ldots, <x_n, s_n>\}$ be the result of the snapshot of *p* when deciding ***s***
- Let $\{<x'_1, s'_1>, \ldots, <x'_n, s'_n>\}$ be the result of the snapshot of *q* when deciding ***s***
- Assume that *p* called ***snap*** before *q*
- Then q's snapshot includes $<x_p, s>$, hence *q* cannot propose ***s*** as a name, contradiction
- Same if q called ***snap*** before *p*

Useful tip:
*Of any two linearized **snapshot()** operations, one's results are "included" in the other's results.*

(2n-1)-renaming

```
Shared: array of registers R[1…Y]
each register in R has two components <x, s>
procedure getName (x)
    s ← 1                         // suggested name
    while( true )
        R[x] ← <x, s>
        (<$x_1$, $s_1$>, … ,<$x_n$, $s_n$>) ← R.snap()
        if $s = s_j$ for some $x_j \neq x$      //there is a name clash
            r ← rank of x in { $x_i$ | $x_i \neq$ empty }
            s ← $r^{th}$ positive integer not in
                    { $s_i$ | $i \neq x$ $x_i \neq$ empty }
        else      //no clash
            return s
```

```
//general structure:
while(true)
    try name s
    if ( clash ) s ← new proposal
    else return s
```

Namespace size

```
Shared: array of registers R[1…Y]
each register in R has two components <x, s>
procedure getName (x)
    s ← 1                          // suggested name
    while( true )
        R[x] ← <x, s>
        $(<x_1, s_1>, \ldots, <x_n, s_n>)$ ← R.snap()
        if $s = s_j$ for some $x_j \neq x$      //there is a name clash
            r ← rank of x in { $x_i$ | $x_i \neq$ empty }
            s ← $r^{th}$ positive integer not in
                            { $s_i$ | $i \neq x$, $x_i \neq$ empty }
        else      //no clash
            return s
```

- Claim: $y < 2n$ in all executions
- Step 1: Notice that
  $r \leq n$
- Step 2: Notice that
  $s \leq r$ + # proposals made in this “round” $- 1 < 2n$
- q.e.d.

Termination

- Main idea of the proof (full proof is homework!)
- By contradiction: assume exists *p* that takes ∞ steps in an execution
- Fix an execution prefix *E* in which every process has executed “R[x] ← <x, s>” at least once or crashed.
  Let F = $\{z_1, z_2, \ldots\}$ be the names that are still free after *E*
- Let *q* be the process with smallest initial name x, that hasn’t decided or crashed so far

Claim: *q* decides within a finite number of steps, or crashes

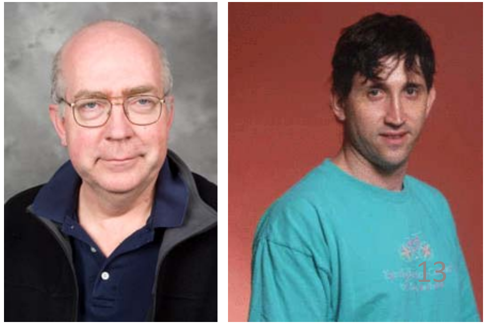
- Step 1: Let *r* be the rank of *q*’s initial value x_q among all initial values.
  Eventually, no process other than *q* proposes names in $\{z_1, \ldots, z_r\}$ (prove it!)
- Step 2: Process *q* eventually suggests name z_r or crashes. (prove it!)
- Step 3: 1 + 2 implies *q* is eventually successful in getting name z_r

Wrap-up

- We have an algorithm that returns names from 1 to 2n – 1 in an *asynchronous* system
- Can we do better?

Theorem [HS, RC] In an asynchronous system with $t < N$ crashes, Deterministic Renaming is impossible in $N + t - 1$ or less names.

- Both Shared-Memory and Message-Passing
- Uses Algebraic Topology!
- Gödel Prize 2004



There’s a problem

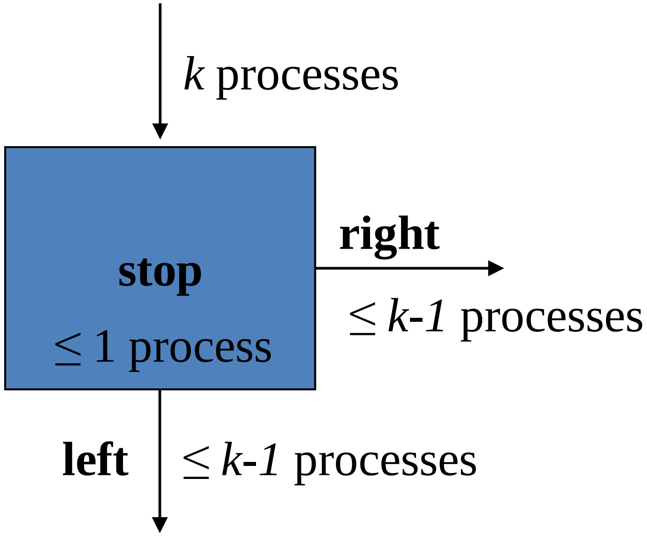
- In the previous algorithm, the size of the proposal array R[ ] is Θ(Y)!
  - Huge memory cost
  - Huge complexity for the snap() operation
- We need to make the size of the array depend on *k* = the number of *participating* processes
- An application of *adaptive renaming*

An adaptive renaming algorithm

- Each process starts with a unique initial name from 1 to Y
- Will return an integer y from 1 to k^2
- k is the contention in the current execution, i.e. the number of *active* processes in the execution

The splitter

[Moir & Anderson, 1995]



k processes

stop

≤ 1 process

right

$\leq k-1$ processes

left

$\leq k-1$ processes

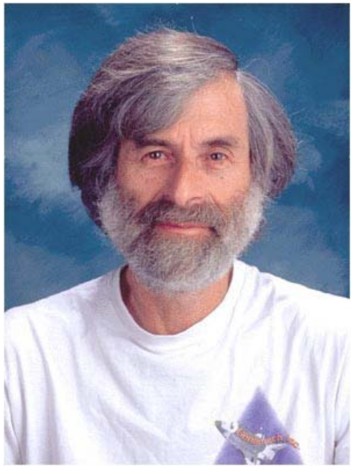
Solo-winner:

A process stops if it is alone in the splitter.

Splitter Implementation

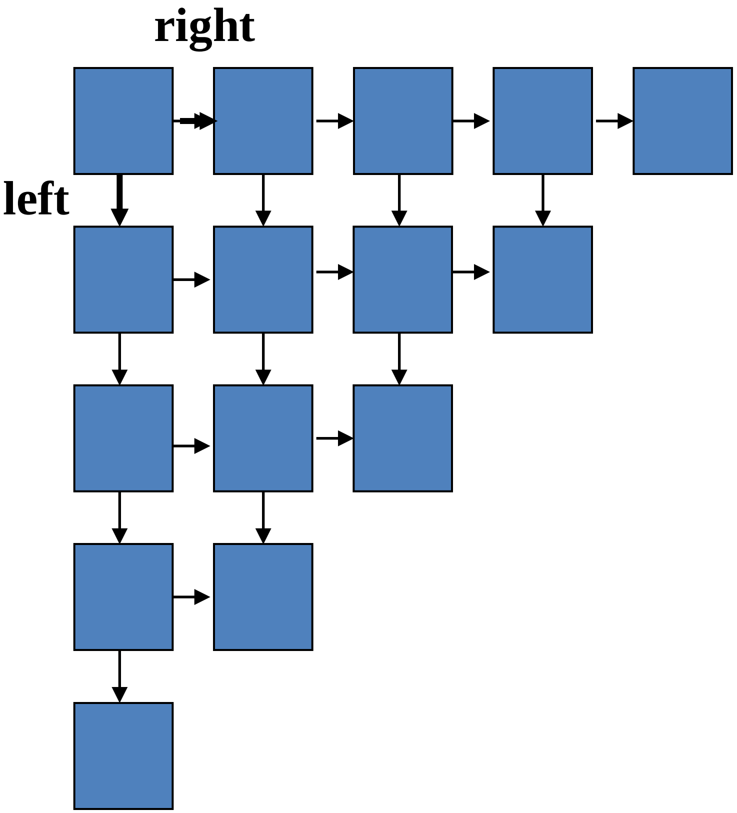
[Moir & Anderson, 1995][Lamport, 1986]

```
1. X = id_i              // write your identifier
2. if Y then return( right )
3. Y = true
4. if ( X == id_i )   // check identifier
       then return( stop )
5.else return( left )
```



Splitters -> Renaming

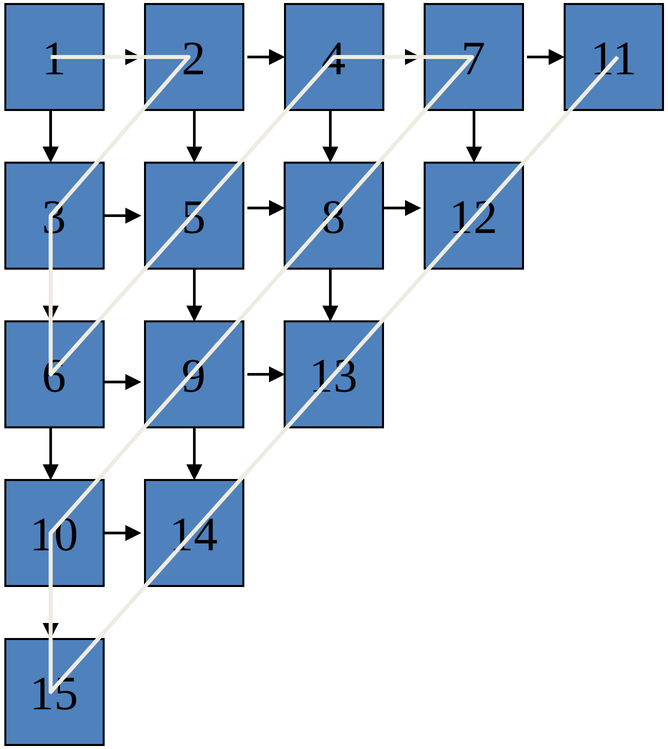
- A triangular matrix of splitters
- Traverse matrix, starting top left, according to the values returned by splitters
- Until process stops in some splitter.



Putting Splitters Together:
k^2 -Renaming

Diagonal association of names with splitters.

⇨Take a name $\leq k^2$.



Correctness

Termination: Every process stops after $O(k)$ read and write steps.

- Follows from the solo-winner splitter property

Uniqueness: No two processes return the same name.

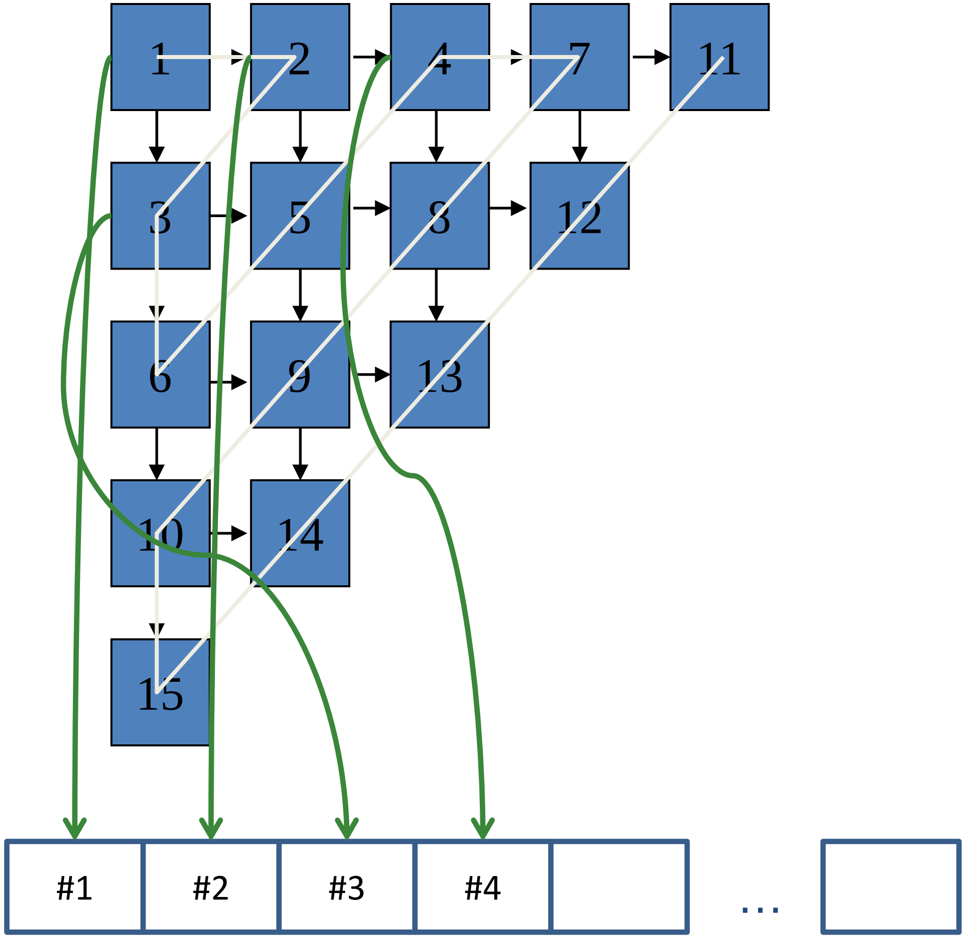
- Since no two processes win the same splitter

Namespace size: Every process returns a name between 1 and $k^2 / 2$.

- Follows since no process makes more than k steps.

How does this help?

- Adaptive Snapshot
- Each name awards a slot in the vector
- So now the memory used is $O(k^2)$
- The **snap()** operation complexity also becomes $f(k)$ (how?)

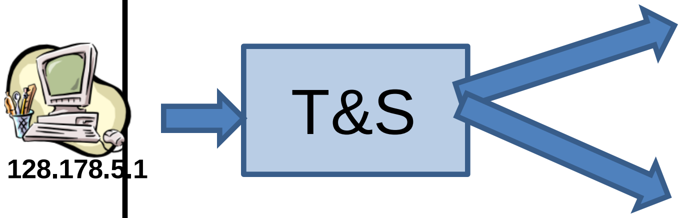


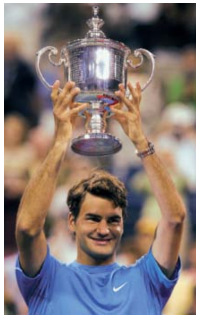
The plan for today

- Renaming definition
- Renaming algorithms
  - (2n – 1)-renaming algorithm
  - Can we do better?
  - Adaptive O(k^2)-renaming algorithm
- Renaming versus test-and-set
  - Consensus number

Test-and-set

```
Shared: V, a binary MWMR
    atomic register, initially 0
procedure Test-and-Set()
    if V = 0 then V ← 1
        return winner
    else return loser
```




winner


loser

Test-and-Set from Adaptive *Tight* Renaming

```
Shared: AdRen, an adaptive tight
    renaming object
procedure Test-and-Set$_i$()
    name ← AdRen.getName(i)
    if name = 1 then
        return winner
    else return loser
```

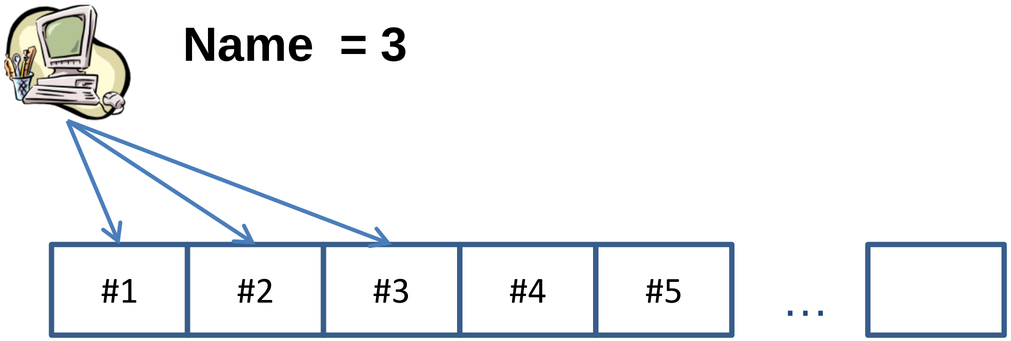
Exactly one process
gets name 1
(or crashes)

- Adaptive tight renaming returns names from 1 to *k* when *k* processes are active
- What goes wrong when renaming is not tight? What if it's not adaptive?

Adaptive Tight Renaming from Test-and-Set

Shared: *V*, an infinite vector of test-and-set objects

```
procedure getName(i)
    j ← 1
    while( true )
        res ← V[j].$\text{Test-and-set}_i$ ()
        if res = winner then
            return j
        else j ← j + 1
```



Tight adaptive renaming

- Using read-write registers, tight adaptive renaming is impossible
- By Herlihy-Shavit [HS], we can't even get close to *k* names!
- It all changes when adding test-and-set
- How many operations per process does the algorithm have?
  - We can get O(log k) operations per process using randomization

Consensus number?

- Consider *adaptive tight* renaming
- Three steps
  1. We can implement it with test-and-set + registers
  2. We can implement test-and-set *from* it
  3. Test-and-set has consensus number 2
- Adaptive tight renaming has consensus number 2!
- Weaker variants ("standard") have consensus number ≤ 2

References (use Google Scholar)

- For definitions + “standard” renaming algorithm
  - Hagit Attiya, Jennifer Welch: “Distributed Computing”, pages 356-359
- For topology [HS], see here http://www.cs.brown.edu/~mph/topology.html
- For adaptive renaming
  - Deterministic – Mark Moir: “Fast, Long-Lived Renaming Improved and Simplified”
  - Randomized -- Dan Alistarh, Hagit Attiya, Seth Gilbert, Andrei Giurgiu, and Rachid Guerraoui: “Fast Randomized Test-and-Set and Renaming”