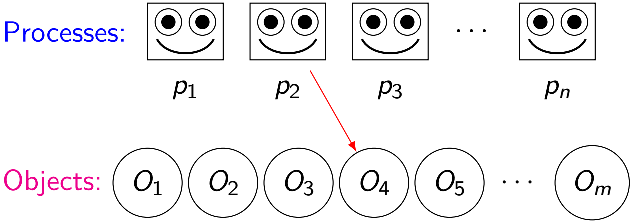
Allocate-On-Use Space Complexity of Shared-Memory Algorithms

James Aspnes Bernhard Haeupler
Alexander Tong Philipp Woelfel

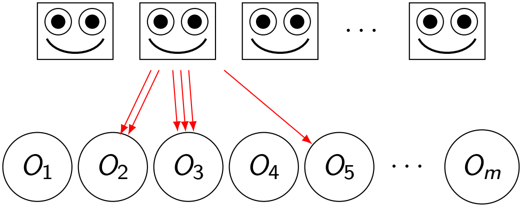
DISC 2018

Asynchronous shared memory model



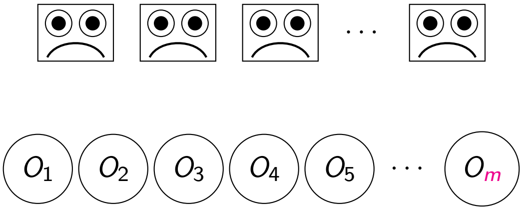
Processes apply atomic operations to objects,
scheduled by an adversary.

Time complexity



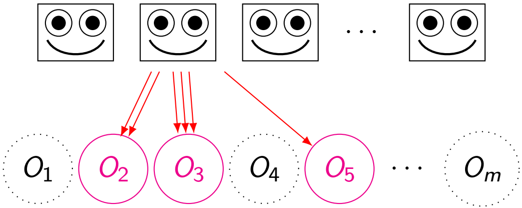
- Many popular time measures:
  - **Total step complexity**: how many operations did we do?
  - **Per-process step complexity**: how many operations did I do?
  - **RMR complexity**: how many times did I see a register change?
- All of these measures are per execution:
  - **Expected** step complexity,
  - **High probability** step complexity,
  - **Adaptive** step complexity,
  - etc.

Space complexity (traditional version)



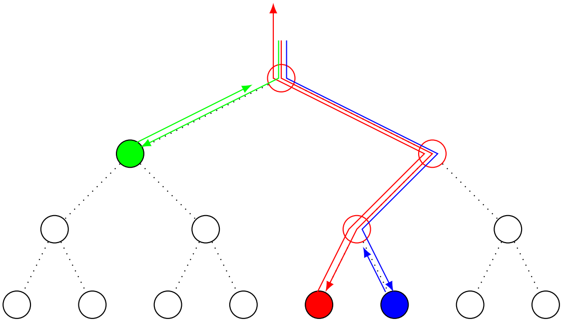
- **Space complexity** = number of objects m.
- Does not change from one execution to the next.
- Linear lower bounds for mutex (Burns-Lynch), perturbable objects (Jayanti-Tan-Toueg), consensus (Zhu).
- Trouble for both theory and practice:
  - Theory: hides effect of randomness.
  - Practice: hides effect of memory management.
- Real systems don't charge you for pages you don't touch.

Space complexity (improved version)



- **Space complexity** = number of objects **used** in some execution.
- An object is **used** when an operation is applied to it.
- Represents an **allocate-on-use** policy.
- Gives a **per-execution** measure.

Example: RatRace (Alistarh *et al.*, DISC 2010)



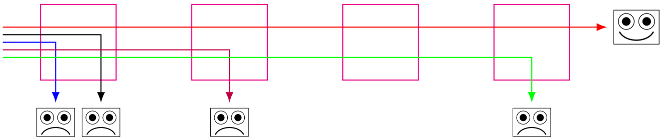
Adaptive test-and-set on a binary tree of depth $3 \log_2 n$.

- **Splitters** allow descending processes to claim nodes.
- Three-process **consensus objects** allow ascending processes to escape subtrees.
- Process that escapes the whole tree wins test-and-set.

Requires $\Theta(n^3)$ objects but only $\Theta(k)$ are used w.h.p.

A hidden trade-off for randomized test-and-set?

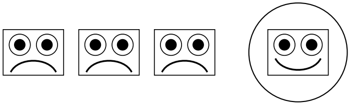
Two algorithms for **randomized test-and-set** with an **oblivious adversary**:



	Time	Space
(Alistarh-Aspnes, DISC 2011)	$\Theta(\log\log n)$	$\Theta(\log\log n)$
(Giakkoupis-Woelfel, PODC 2012)	$\Theta(\log^* n)$	$\Theta(\log n)$

- Both use **sifter** objects to get rid of losing processes quickly.
- Both use $\Theta(n^3)$ worst-case space for backup RatRace.
- Not clear if space-time trade-off is necessary, but without allocate-on-use space complexity it's not even visible.

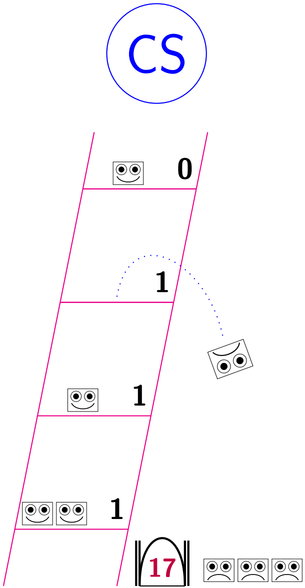
Mutual exclusion


Critical section

- **Mutual exclusion**: at most one process at a time in critical section.
- **Deadlock freedom**: some process reaches critical section eventually.
- (Burns-Lynch, 1993): n registers needed in worst case, by constructing a single bad execution for any given algorithm.

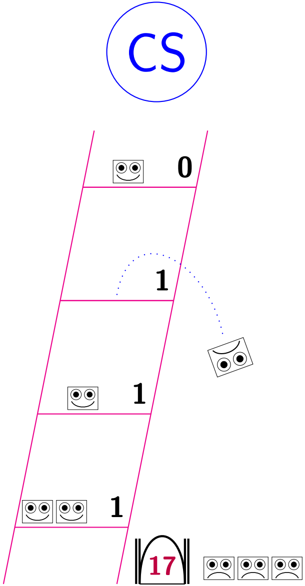
We want to beat this bound for expected space complexity.

Monte Carlo mutual exclusion



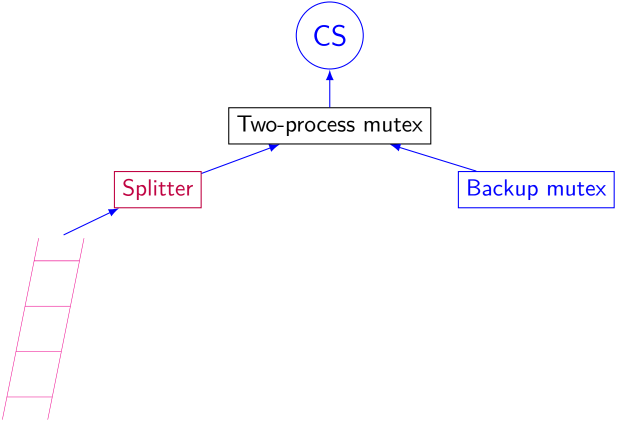
- Processes climb a slippery ladder of one-bit register rungs to reach the critical section (CS).
- Process flips a coin at each rung:
  - Heads: Write 1 and climb.
  - Tails: Read:
    - $0 \Rightarrow$ stay at same rung.
    - $1 \Rightarrow$ fall to holding pen.
- About half fall from each rung.
- $O(\log n)$ rungs leave one process in CS with high probability.
- After finishing CS, winner resets rungs and increments gate.

Monte Carlo mutual exclusion: Analysis



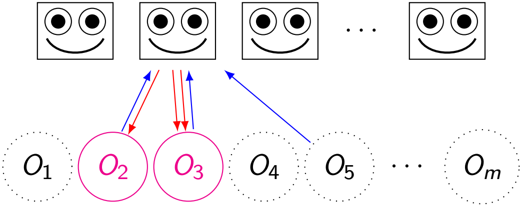
- Deadlock freedom: some process is first to write 1.
- Mutual exclusion:
  - Potential function Φ sums
    - 2^{height} for processes.
    - $-w^{\text{height}}$ for 1 registers.
    - Plus a few extra terms.
  - Φ increases slowly on average.
  - Φ is big when two processes in CS.
  - whp have mutual exclusion in polynomially-long executions.
- $O(n)$ amortized RMRs per CS.

Mutual exclusion in $O(\log n)$ expected space



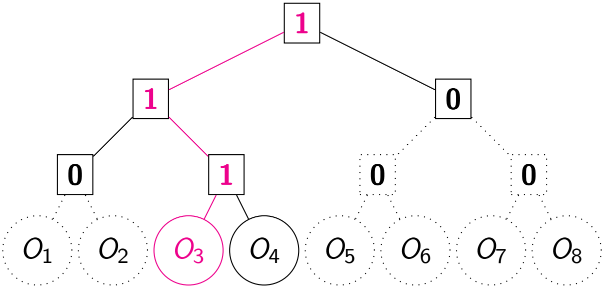
- Splitter detects when Monte Carlo algorithm violates mutex.
- In this case, switch to $O(n)$-space backup mutex.
- Gives mutex always, $O(\log n)$ space whp,
  $O(n)$ amortized RMRs per critical section.

Allocate-on-update space complexity



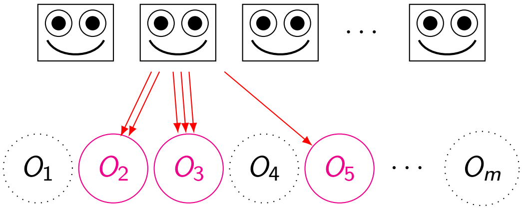
- Many systems allocate pages only on write.
- Analogous notion is **allocate-on-update**.
    - Reading an object is free!
    - Changing an object is not.
- How does this compare to allocate-on-use?

Simulating allocate-on-update with allocate-on-use



- Idea: Use one-bit registers to mark which ranges have changed.
- Balanced binary tree gives $O(\log m)$ overhead.
- Unbalanced tree gives $O(\log(\text{max address updated}))$.
- So models are equivalent up to log factor.

Conclusion and open problems



Allocate-on-use space complexity reveals differences in algorithms that are hidden by worst-case space complexity.

- What other problems allow low allocate-on-use space?
- Space lower bounds for allocate-on-use?
- What happens with an adaptive adversary?
- Low-space mutex with better RMR complexity?
- Implement allocate-on-use in a model that doesn't provide it?