6.852: Distributed Algorithms
Fall, 2009

Class 14

Today's plan

- Mutual exclusion with read/write memory:
  - Lamport's Bakery Algorithm
  - Burns' algorithm
  - Lower bound on the number of registers
- Mutual exclusion with read-modify-write operations
- Reading: Sections 10.6-10.8, 10.9

- Next: Lecture by Victor Luchangco (Sun)
  - Practical mutual exclusion algorithms
  - Generalized resource allocation and exclusion problems
  - Reading:
    - Herlihy, Shavit book, Chapter 7
    - Mellor-Crummey and Scott paper (Dijkstra prize winner)
    - (Optional) Magnussen, Landin, Hagersten paper
    - Distributed Algorithms, Chapter 11

Last time

- Mutual exclusion with read/write memory:
  - Dijkstra’s algorithm:
    - Mutual exclusion + progress
  - Peterson’s algorithms
    - Mutual exclusion + progress + lockout-freedom
  - Lamport’s Bakery algorithm (didn’t get to this)
    - Mutual exclusion + progress + lockout-freedom
    - No multi-writer variables.

Lamport's Bakery Algorithm

- Like taking tickets in a bakery.
- Nice features:
  - Uses only single-writer, multi-reader registers.
  - Extends to even weaker registers, in which operations have durations, and a read that overlaps a write receives an arbitrary response.
  - Guarantees lockout-freedom, in fact, almost-FIFO behavior.
- But:
  - Registers are unbounded size.
  - Algorithm can be simulated using bounded registers, but not easily (uses bounded concurrent timestamps).

- Shared variables:
  - For each process i:
    - choosing(i), a Boolean, written by i, read by all, initially 0
    - number(i), a natural number, written by i, read by all, initially 0

Bakery Algorithm

- First part, up to choosing(i) := 0 (the “Doorway”, D):
  - Process i chooses a number number greater than all the numbers it reads for the other processes; writes this in number(i).
  - While doing this, keeps choosing(i) = 1.
  - Two processes could choose the same number (unlike real bakery).
  - Break ties with process ids.

- Second part:
  - Wait to see that no others are choosing, and no one else has a smaller number.
  - That is, wait to see that your ticket is the smallest.
  - Never go back to the beginning of this part---just proceed step by step, waiting when necessary.

Code

Shared variables:

for every $i \in \{1,\ldots,n\}$:

choosing(i) $\in \{0,1\}$, initially 0, writable by i, readable by all $j \neq i$

number(i), a natural number, initially 0, writable by i, readable by $j \neq i$.

try_i

choosing(i) := 1

number(i) := 1 + $\max_{j \neq i}$ number(j)

choosing(i) := 0

for $j \neq i$ do

    waitfor choosing(j) = 0

    waitfor number(j) = 0 or (number(i), i) < (number(j), j)

$crit_i$

$exit_i$

number(i) := 0

rem_i

Correctness: Mutual exclusion

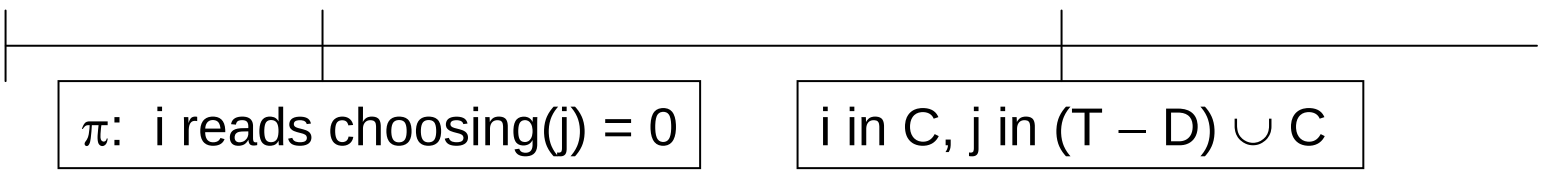
- Key invariant: If process i is in C, and process j $\neq$ i is in $\underbrace{(T - D) \cup C}_{\text{Trying region after doorway, or critical region}}$,

  then (number(i),i) < (number(j),j).

- Proof:
  – Could prove by induction.
  – Instead, give argument based on events in executions.
  – This argument extends to weaker registers, with concurrent accesses.

Correctness: Mutual exclusion

- Invariant: If i is in C, and $j \neq i$ is in $(T - D) \cup C$, then (number(i),i) < (number(j),j).
- Proof:
  - Consider a point where i is in C and $j \neq i$ is in $(T - D) \cup C$.
  - Then before i entered C, it must have read choosing(j) = 0, event π.



  - Case 1: j sets choosing(j) := 1 (starts choosing) after π.
    - Then number(i) is set before j starts choosing.
    - So j sees the "correct" number(i) and chooses something bigger.
  - Case 2: j sets choosing(j) := 0 (finishes choosing) before π.
    - Then when i reads number(j) in its second waitfor loop, it gets the "correct" number(j).
    - Since i decides to enter C anyway, it must have seen (number(i),i) < (number(j),j).

Correctness: Mutual exclusion

- Invariant: If i is in C, and $j \neq i$ is in $(T - D) \cup C$, then $(\text{number}(i),i) < (\text{number}(j),j)$.
- Proof of mutual exclusion:
  - Apply invariant both ways.
  - Contradictory requirements.

Liveness Conditions

- Progress:
  - By contradiction.
  - If not, eventually region changes stop, leaving everyone in T or R, and at least one process in T.
  - Everyone in T eventually finishes choosing.
  - Then nothing blocks the smallest (number, index) process from entering C.

- Lockout-freedom:
  - Consider any i that enters T
  - Eventually it finishes the doorway.
  - Thereafter, any newly-entering process picks a bigger number.
  - Progress implies that processes continue to enter C, as long as i is still in T.
  - In fact, this must happen infinitely many times!
  - But those with bigger numbers can't get past i, contradiction.

FIFO Condition

- Not really FIFO ($\rightarrow$T vs. $\rightarrow$C), but almost:
  - FIFO after the doorway: if j leaves D before i $\rightarrow$T, then j $\rightarrow$C before i $\rightarrow$C.
- But the “doorway” is an artifact of this algorithm, so this isn’t a meaningful way to evaluate the algorithm!
- Maybe say “there exists a doorway such that”…
- But then we could take D to be the entire trying region, making the property trivial.
- To make the property nontrivial:
  - Require D to be “wait-free”: a process is guaranteed to complete D it if it keeps taking steps, regardless of what other processes do.
  - D in the Bakery Algorithm is wait-free.
- The algorithm is FIFO after a wait-free doorway.

Impact of Bakery Algorithm

- Originated important ideas:
  - Wait-freedom
    - Fundamental notion for theory of fault-tolerant asynchronous distributed algorithms.
  - Weakly coherent memories
    - Beginning of formal study: definitions, and some algorithmic strategies for coping with them.

Space and memory considerations

- All mutual exclusion algorithms use more than n variables.
  - Bakery algorithm could use just n variables. (Why?)
- All but Bakery use multi-writer variables.
  - These can be expensive to implement
- Bakery uses infinite-size variables
  - Difficult (but possible) to adapt to use finite-size variables.
- Q: Can we do better?

Burns’ Algorithm

Burns' algorithm

- Uses just n single-writer Boolean read/write variables.
- Simple.
- Guarantees safety (mutual exclusion) and progress.
  - But not lockout-freedom!

Code

Shared variables:

for every i $\in$ {1,…,n}:

flag(i) $\in$ {0,1}, initially 0, writable by i, readable by all $j \neq i$

Process i:

```
     try_i

L:   flag(i) := 0
     for j ∈ {1,…,i-1} do
         if flag(j) = 1 then go to L
     flag(i) := 1
     for j ∈ {1,…,i-1} do
         if flag(j) = 1 then go to L
M:   for j ∈ {i+1,…,n} do
     if flag(j) = 1 then go to M

     crit_i
```

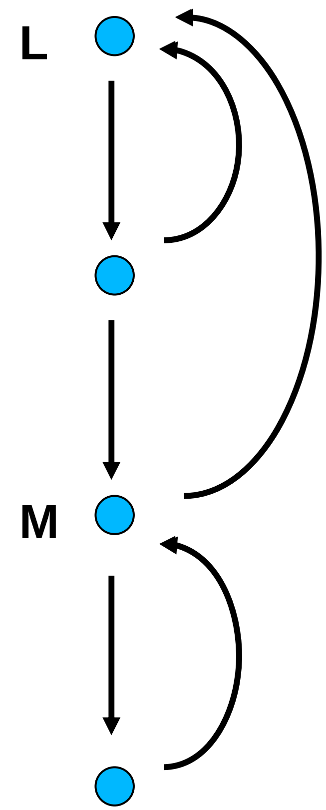
```
exit_i

flag(i) := 0

rem_i
```

That is,...

- Each process goes through 3 loops, sequentially:
  1. Check flags of processes with smaller indices.
  2. Check flags of processes with smaller indices.
  3. Check flags of processes with larger indices.
- If it passes all tests, $\rightarrow$ C.
- Otherwise, drops back:



Correctness of Burns' algorithm

- Mutual exclusion + progress
- Mutual exclusion:
  - Like the proof for Dijkstra's algorithm, but now with flags set to 1 rather than 2.
  - If processes i and j are ever in C simultaneously, both must have set their flags := 1.
  - Assume WLOG that process i sets flag(i) := 1 (for the last time) first.
  - Keeps flag(i) = 1 until process i leaves C.
  - After flag(i) := 1, must have flag(j) := 1, then j must see flag(i) = 0, before j $\rightarrow$ C.
  - Impossible!

Progress for Burns' algorithm

- Consider fair execution α (each process keeps taking steps).
- Assume for contradiction that, after some point in α, some process is in T, no one is in C, and no one $\rightarrow$ C later.
- WLOG, we can assume that every process is in T or R, and no region changes occur after that point in α.
- Call the processes in T the contenders.
- Divide the contenders into two sets:
  - P, the contenders that reach label M, and
  - Q, the contenders that never reach M.
- After some point in α, all contenders in P have reached M; they never drop back thereafter to before M.

α

α': All processes in T or R; someone in T; no region changes, all processes in P in final loop.

Progress for Burns' algorithm

- P, the contenders that reach label M, and
- Q, the contenders that never reach M.

α

α': All processes in T or R; someone in T; no region changes, all processes in P in final loop.

- Claim P contains at least one process:
  - Process with the lowest index among all the contenders is not blocked from reaching M.
- Let i = largest index of a process in P.
- Claim process i eventually $\rightarrow$ C: All others with larger indices eventually see a smaller-index contender and drop back to L, setting their flags := 0 (and these stay = 0).
- So i eventually sees all these = 0 and $\rightarrow$ C.
- Contradiction.

Lower Bound on the Number of Registers

Lower Bound on the Number of Registers

- All the mutual exclusion algorithms we've studied:
  - Use read/write shared memory, and
  - Use at least n read/write shared variables.
- That's one variable per potential contender.

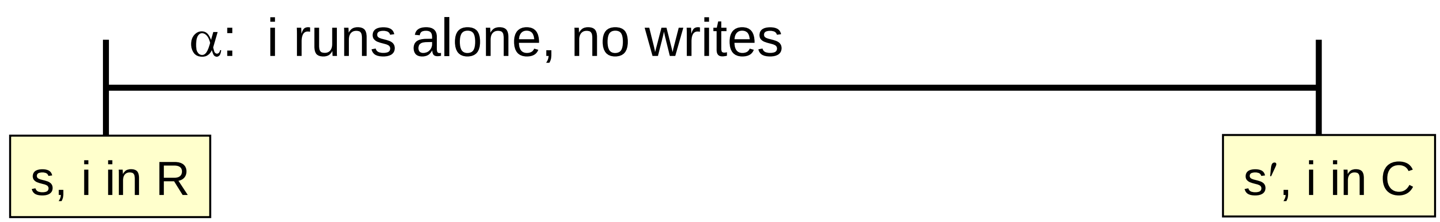
- Q: Can we use fewer than n r/w shared variables?
- Not single-writer. (Why?)
- Not even multi-writer!

Lower bound on number of registers

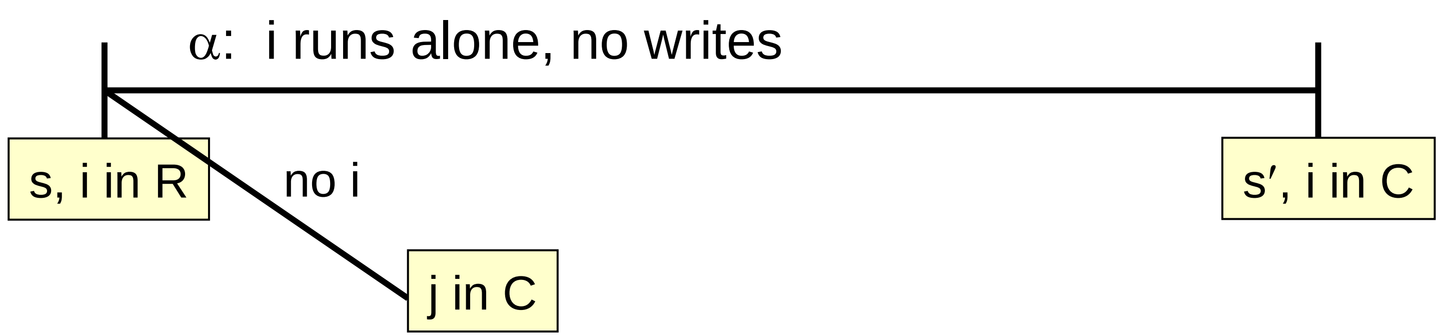
- Lower bound of n holds even if:
  - We require only mutual exclusion + progress (no stronger liveness properties).
  - The variables can be any size.
  - Variables can be read and written by all processes.
- Start with basic facts about any mutex algorithm A using r/w shared variables.
- Lemma 1: If s is a reachable, idle system state (meaning all processes are in R), and if process i runs alone from s, then eventually $i \rightarrow C$.
- Proof: By the progress requirement.
- Corollary: If i runs alone from a system state s′ that is indistinguishable from s by i, $s' \sim^i s$, then eventually $i \rightarrow C$.
- Indistinguishable: Same state of i and same shared variable values.

Lower bound on registers

- Lemma 2: Suppose that s is a reachable system state in which i $\in$ R. Suppose process i $\rightarrow$ C on its own, from s. Then along the way, process i writes to some shared variable.
- Proof:
  - By contradiction; suppose it doesn't.
  - Then:



  - Then $s' \sim^j s$ for every $j \neq i$.
  - Then there is some execution fragment from s in which process i takes no steps, and in which some other process j $\rightarrow$ C.
    - By repeated use of the progress requirement.



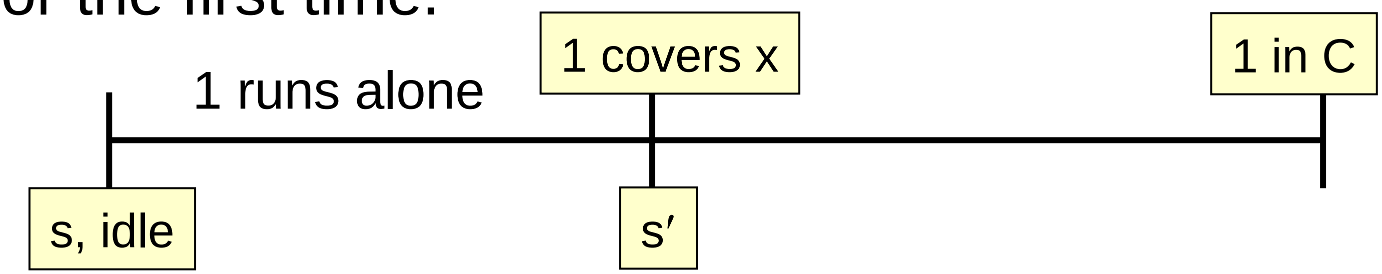
Lower bound on registers

- Lemma 2: Suppose that s is a reachable system state in which i $\in$ R. Suppose process i $\to$ C on its own, from s. Then along the way, process i writes to some shared variable.
- Proof, cont'd:
  - There is some execution fragment from s in which process i takes no steps, and in which some other process j $\to$ C.

  - Then there is also such a fragment from s′.
  - Yields a counterexample execution:
    - System gets to s, then i alone takes it to s′, then others get j in C.
    - Contradiction because i,j are in C at the same time.

Lower bound on registers

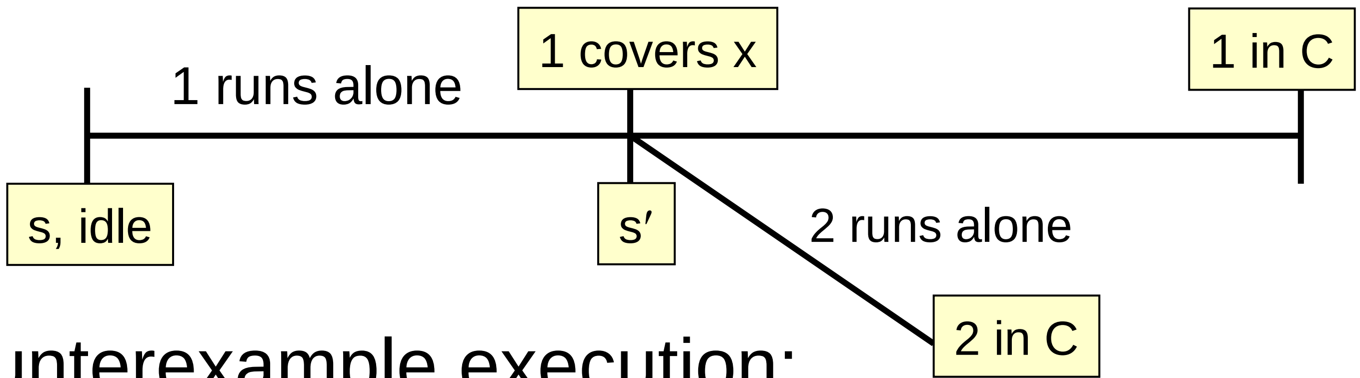
- Back to showing ≥ n shared variables needed…
- Special case: 2 processes and 1 variable:
  - Suppose A is a 2-processes mutex algorithm using 1 r/w shared variable x.
  - Start in initial (idle) state s.
  - Run process 1 alone, → C, writes x on the way.
    - By Lemmas 1 and 2.
  - Consider the point where process 1 is just about to write x, i.e., covers x, for the first time.



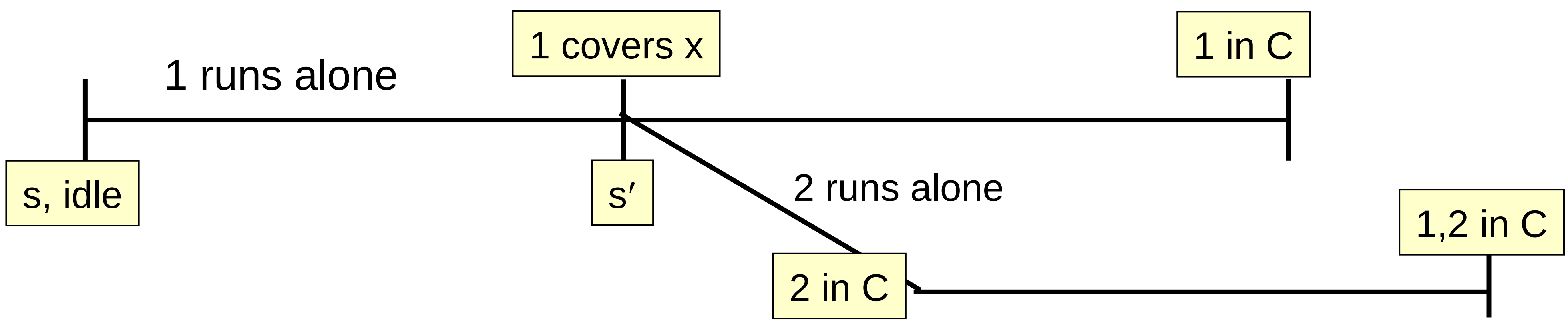
  - Note that $s' \sim^2 s$, because 1 doesn't write between s and s′.
  - So process 2 can reach C on its own from s′.
    - By Corollary to Lemma 1.

2 processes, 1 variable

- Process 2 can reach C on its own from s′:



- Counterexample execution:
  - Run 1 until it covers x, then let 2 reach C.
  - Then resume 1, letting it write x and then → C.
  - When it writes x, it overwrites anything 2 might have written there on its way to C; so 1 never sees any evidence of 2.

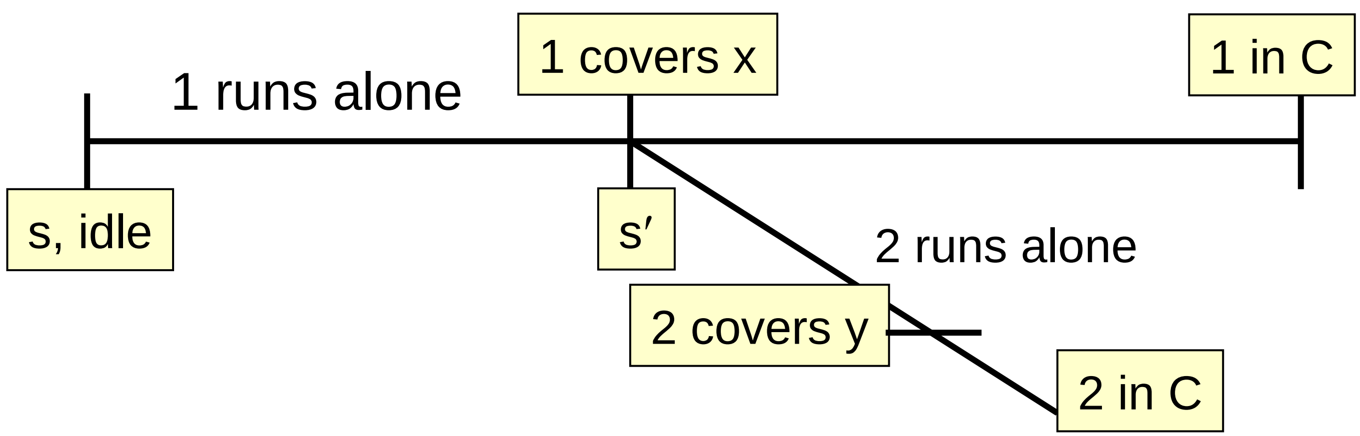


Another special case: 3 processes, 2 variables

- Processes 1, 2, 3; variables x,y.
- Similar construction, with a couple of twists.
- Start in initial (idle) state s.
- Run processes 1 and 2 until:
  - Each covers one of x,y---both variables covered.
  - Resulting state is indistinguishable by 3 from a reachable idle state.
- Q: How to do this?
  - For now, assume we can.
- Then run 3 alone, $\rightarrow$ C.
- Then let 1 and 2 take one step each, overwriting both variables, and obliterating all traces of 3.
- Continue running 1 and 2; they run as if 3 were still in R.
- By progress requirement, one eventually $\rightarrow$ C.
- Contradicts mutual exclusion.

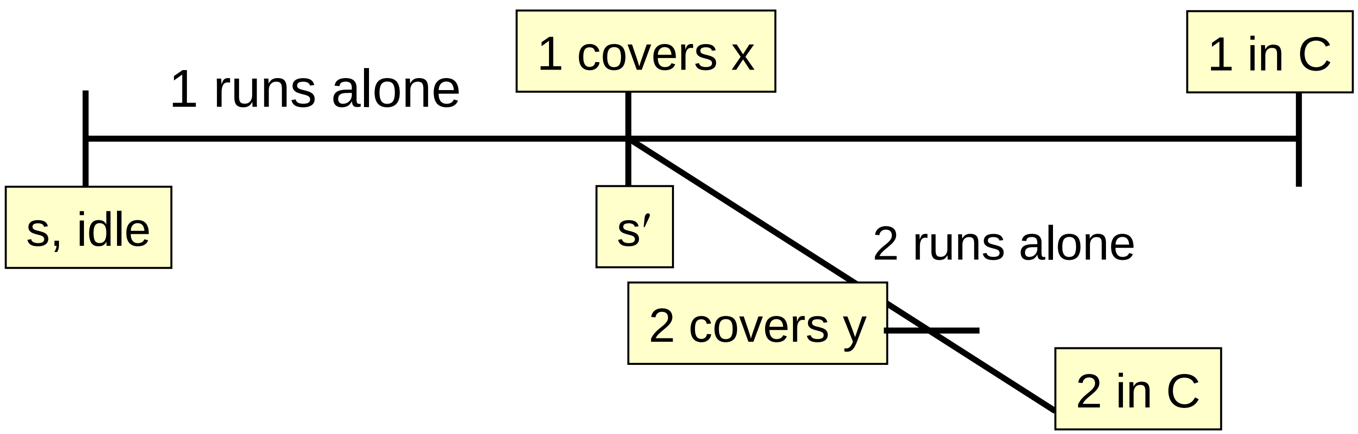
3 processes, 2 variables

- It remains to show how to maneuver 1 and 2 so that:
  - Each covers one of x,y.
  - Resulting state is indistinguishable by 3 from a reachable idle state.
- First try:
  - Run 1 alone until it first covers a shared variable, say x.
  - Then run 2 alone until $\rightarrow$ C.
  - Claim: Alone the way, it must write the other shared variable y.
    - If not, then after 2 $\rightarrow$ C, 1 could take one step, overwriting anything 2 wrote to x, and thus obliterating all traces of 2.
    - Then 1 continues $\rightarrow$ C, violating mutual exclusion.
  - Stop 2 just when it first covers y; then 1 and 2 cover x and y.



3 processes, 2 variables

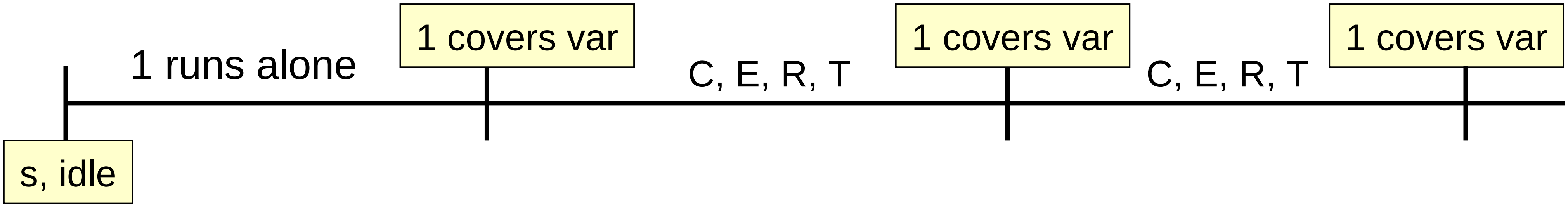
- Maneuver 1 and 2 so that:
  - Each covers one of x,y.
  - Resulting state is indistinguishable by 3 from a reachable idle state.



- But this is not quite right… resulting state might not be indistinguishable by 3 from an idle state.
- 2 could have written x before writing y.

3 processes, 2 variables

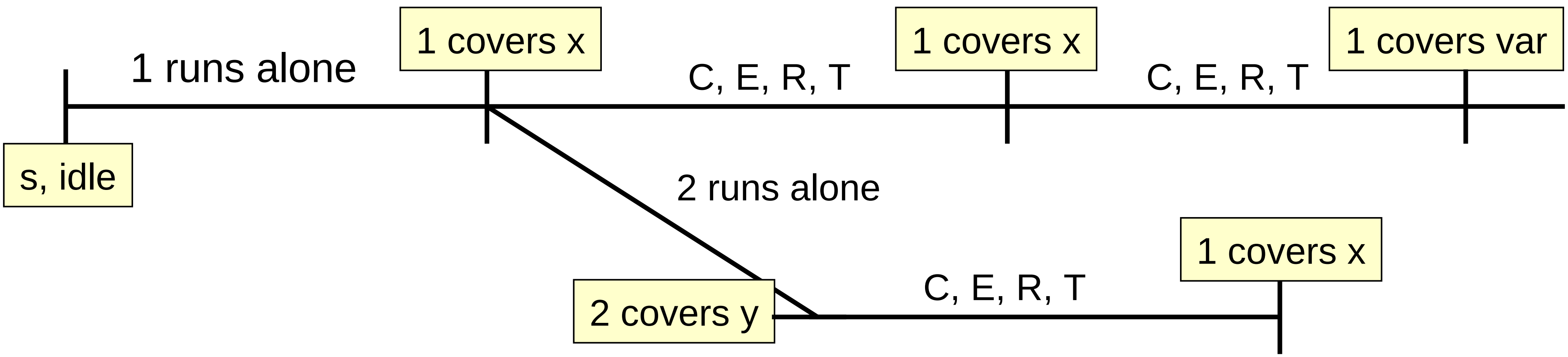
- Maneuver 1 and 2 so that:
  - Each covers one of x,y.
  - Resulting state is indistinguishable by 3 from a reachable idle state.
- Second (successful) try:
  - Run 1 alone until it first covers a shared variable.
  - Continue running 1, through C, E, R, back in T, until it again first covers a variable.
  - And once again.



  - In two of the three covering states, 1 must cover the same variable.
  - E.g., suppose in first two states, 1 covers x (other cases analogous).

3 processes, 2 variables

- Counterexample execution:
  - Run 1 until it covers x the first time.
  - Then run 2 until it first covers y (must do so).



  - Then let 1 write x and continue until it covers x again.
  - Now both variables are (again) covered.
  - This time, the final state is indistinguishable by 3 from an idle state.
  - As needed.

General case: n processes, n-1 variables

- Extends 3-process 2-variable case, using induction.
- Need strengthened version of Lemma 2:
- Lemma 2′: Suppose that s is a reachable system state in which $i \in R$. Suppose process $i \rightarrow C$ on its own, from s. Then along the way, process i writes to some shared variable that is not covered (in s) by any other process.
- Proof:
  - Similar to Lemma 2.
  - Contradictory execution fragment begins by overwriting all the covered variables, obliterating any evidence of i.

n processes, n-1 variables

- Definition: s′ is k-reachable from s if there is an execution fragment from s to s′ involving only steps by processes 1 to k.

n processes, n-1 variables

- Now suppose (for contradiction) that A solves mutual exclusion for n processes, with n-1 shared variables.
- Main Lemma: For any $k \in \{1,\ldots,n-1\}$ and from any idle state, there is a k-reachable state in which processes 1,…,k cover k distinct shared variables, and that is indistinguishable by processes k+1,…,n from some k-reachable idle state.
- Proof: In a minute…
- Now assume we have this, for k = n-1.
- Then run n alone, $\rightarrow$ C.
  - Can do this, by Corollary to Lemma 1.
- Along the way, it must write some variable that isn't covered by 1,…,n-1.
  - By Lemma 2′.
- But all n-1 variables are covered, contradiction.

- It remains to prove the Main Lemma…

Proof of the Main Lemma

- Main Lemma: For any $k \in \{1,\ldots,n-1\}$ and from any idle state, there is a k-reachable state in which processes 1 to k cover k distinct shared variables, and that is indistinguishable by processes k+1 to n from some k-reachable idle state.

- Proof: Induction on k.
  - Base case (k=1):
    - Run process 1 alone until just before it first writes a shared variable.
    - 1-reachable state, process 1 covers a shared variable, indistinguishable by the other processes from initial state.
  - Inductive step (Assume for $k \leq n-2$, show for k+1):
    - By inductive hypothesis, get a k-reachable state t_1 in which processes 1,…,k cover k variables, and that is indistinguishable by processes k+1,…,n from some k-reachable idle state.

Proof of the Main Lemma

- Main Lemma: For any $k \in \{1,\ldots,n-1\}$ and from any idle state, there is a k-reachable state in which processes 1 to k cover k distinct shared variables, and that is indistinguishable by processes k+1 to n from some k-reachable idle state.
- Proof: Inductive step (Assume for $k \leq n-2$, show for k+1):
  - By I.H., get a k-reachable state t_1 in which 1,…,k cover k variables, and that is indistinguishable by k+1,…,n from some k-reachable idle state.
  - Let each of 1,…,k take one step, overwriting covered variables.
  - Run 1,…,k until all are back in R; resulting state is idle.
  - By I.H. get another k-reachable state t_2 in which 1,…, k cover k variables, and that is indistinguishable by k+1,…,n from some k-reachable idle state.
  - Repeat, getting t_3, t_4,…, until we get t_i and t_j ($i < j$) that cover the same set X of variables. (Why is this guaranteed to happen?)
  - Run k+1 alone from t_i until it first covers a variable not in X.
  - Then run 1,…,k as if from t_i to t_j (they can't tell the difference).
  - Now processes 1,…,k+1 cover k+1 different variables.
  - And result is indistinguishable by k+2,…,n from an idle state.

Discussion

- Bell Labs research failure:
  - At Bell Labs (many years ago), Gadi Taubenfeld found out that the Unix group was trying to develop an asynchronous mutual exclusion algorithm for many processes that used only a few r/w shared registers.
  - He told them it was impossible.

Discussion

- New research direction:
  - Develop “space-adaptive” algorithms that potentially use many variables, but are guaranteed to use only a few if only a few processes are contending.
  - Also “time-adaptive” algorithms.
  - See work by [Moir, Anderson], [Attiya, Friedman]
  - Time-adaptive and space-adaptive algorithms often yield better performance, lower overhead, in practice.

Mutual Exclusion with Read-Modify-Write Shared Variables

Mutual exclusion with RMW shared variables

- Stronger memory primitives (synchronization primitives):
  - Test-and-set, fetch-and-increment, swap, compare-and-swap, load-linked/store-conditional,…
- All modern computer architectures provide one or more of these, in addition to read/write registers.
- Generally support reads and writes, as well as more powerful operations.
- More expensive (cost of hardware, time to access) than variables supporting just reads and writes.
- Not all the same strength; we'll come back to this later.

- Q: Do such stronger memory primitives enable better algorithms, e.g., for mutual exclusion?

Mutual exclusion with RMW: Test-and-set algorithm

- test-and-set operation: Sets value to 1, returns previous value.
  - Usually for binary variables.
- Test-and-set mutual exclusion algorithm (trivial):
  - One shared binary variable x, 0 when no one has been granted the resource (initial state), 1 when someone has.
  - Trying protocol: Repeatedly test-and-set x until get 0.
  - Exit protocol: Set x := 0.

try_i
  waitfor(test-and-set(x) = 0)
$crit_i$

$exit_i$
  x := 0
rem_i

- Guarantees mutual exclusion + progress.
- No fairness. To get fairness, we can use a more expensive queue-based algorithm:

Mutual exclusion with RMW: Queue-based algorithm

- queue shared variable
  - Supports enqueue, dequeue, head operations.
  - Can be quite large!
- Queue mutual exclusion algorithm:
  - One shared variable Q: FIFO queue.
  - Trying protocol: Add self to Q, wait until you're at the head.
  - Exit protocol: Remove self from Q.

try_i
  enqueue(Q,i)
  waitfor(head(Q) = i)
$crit_i$

$exit_i$
  dequeue(Q)
rem_i

- Fairness: Guarantees bounded bypass (indeed, no bypass = 1-bounded bypass).

Mutual exclusion with RMW: Ticket-based algorithm

- Modular fetch-and-increment operation, $f\&i_n$
  - Variable values are integers mod n.
  - Increments variable mod n, returns the previous value.
- Ticket mutual exclusion algorithm:
  - Like Bakery algorithm: Take a number, wait till it's your turn.
  - Guarantees bounded bypass (no bypass).
  - Shared variables: next, granted: integers mod n, initially 0
    - Support modular fetch-and-increment.
  - Trying protocol: Increment next, wait till granted.
  - Exit protocol: Increment granted.

try_i
  ticket := $f\&i_n$(next)
  waitfor(granted = ticket)
$crit_i$

$exit_i$
  $f\&i_n$(granted)
rem_i

Ticket-based algorithm

- Space complexity:
  - Each shared variable takes on at most n values.
  - Total number of variable values: n^2
  - Total size of variables in bits: 2 log n
- Compare with queue:
  - Total number of variable values:

    n! + (n choose (n-1)) (n-1)! + (n ch (n-2)) (n-2)! +…+ (n ch 1) 1!

    = n! (1 + 1/1! + 1/2! + 1/3! +…+ 1/(n-1)!)

    $\leq$ n! e = $O(n^n)$
  - Size of variable in bits: O(n log n)

try_i
  ticket := $f\&i_n$(next)
  waitfor(granted = ticket)
$crit_i$

$exit_i$
  $f\&i_n$(granted)
rem_i

Variable Size for Mutual Exclusion with RMW

- Q: How small could we make the RMW variable?
- 1 bit, for just mutual exclusion + progress (simple test and set algorithm).
- With fairness guarantees?
- O(n) values (O(log n) bits) for bounded bypass.
  - Can get n+k values, for small k.

In practice, on a real shared-memory multiprocessor,
we want a few variables of size O(log n).
So ticket algorithm is pretty good (in terms of space).

- Theoretical lower bounds:
  - $\Omega(n)$ values needed for bounded bypass, $\Omega(\sqrt{n})$ for lockout-freedom.

Variable Size for Mutual Exclusion with RMW

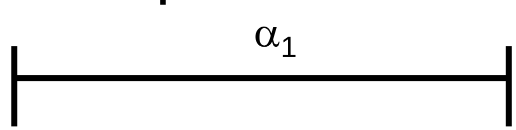
- Theoretical lower bound:
  - $\Omega(n)$ values needed for bounded bypass, $\Omega(\sqrt{n})$ for lockout-freedom.
- Significance:
  - Achieving mutual exclusion + lockout freedom is not trivial, even though we assume that the processes get fair access to the shared variables.
  - Thus, fair access to the shared variables does not immediately translate into fair access to higher-level critical sections.
- For example, consider bounded bypass:…

Lower bound on variable size for mutual exclusion + bounded bypass

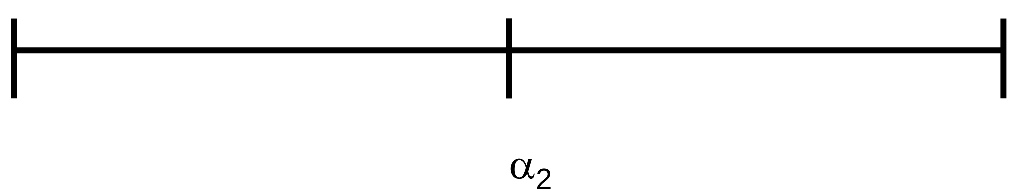
- Theorem: In any mutual exclusion algorithm guaranteeing progress and bounded bypass, using a single RMW shared variable, the variable must be able to take on at least n distinct values.
- Essentially, need enough space to keep a process index, or a counter of the number of active processes, in shared memory.
- General RMW shared variable: Allows read, arbitrary computation, and write, all in one step.
- Proof: By contradiction.
  - Suppose Algorithm A achieves mutual exclusion + progress + k-bounded bypass, using one RMW variable with $< n$ values.
  - Construct a bad execution, which violates k-bounded bypass:

Lower bound on variable size for mutual exclusion + bounded bypass

- Theorem: In any mutual exclusion algorithm guaranteeing progress and bounded bypass, using a single RMW shared variable, the variable must be able to take on at least n distinct values.
- Proof: By contradiction.
  - Suppose Algorithm A achieves mutual exclusion + progress + k-bounded bypass, using one RMW variable with < n values.
  - Run process 1 from initial state, until $\rightarrow$ C, execution α_1:



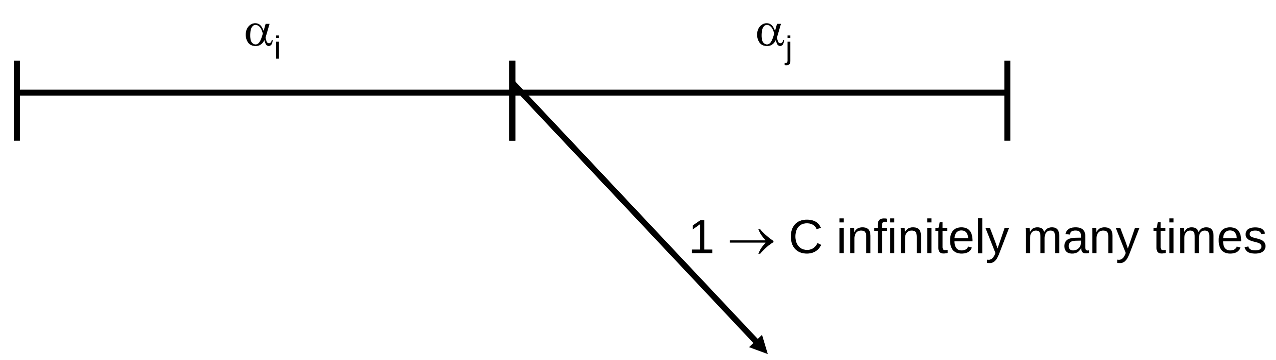
  - Run process 2 until it accesses the variable, α_2:



  - Continue by running each of 3, 4,…,n, obtaining α_3, α_4, …, α_n.

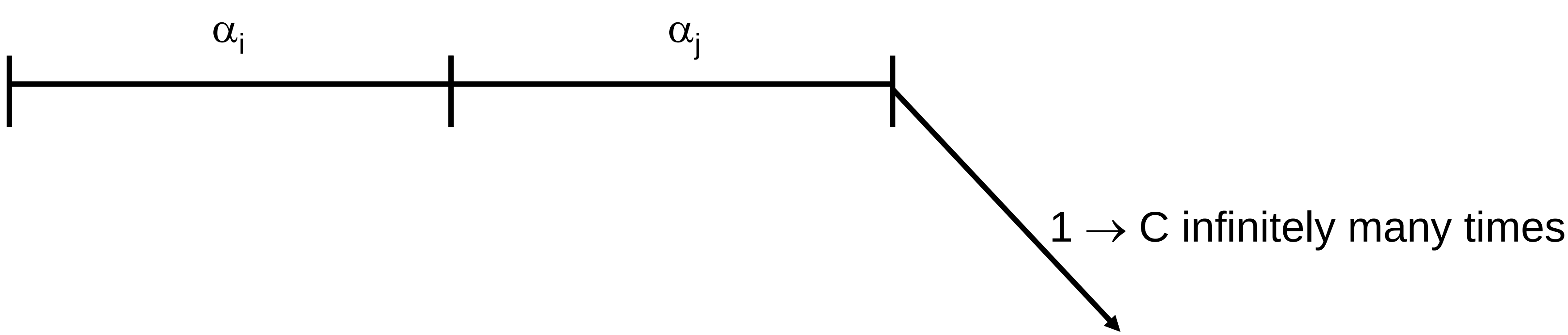
Lower bound on variable size for mutual exclusion + bounded bypass

- Theorem: In any mutual exclusion algorithm guaranteeing bounded bypass, using a single RMW shared variable, the variable must be able to take on at least n distinct values.
- Proof, cont'd:
  - Since the variable takes on $< n$ values, there must be two processes, i and j, $i < j$, for which α_i and α_j leave the variable with the same value v.
  - Now extend α_i so that 1,…,i exit, then 1 reenters repeatedly, $\rightarrow$ C infinitely many times.
    - Possible since progress is required in a fair execution.



Lower bound on variable size for mutual exclusion + bounded bypass

- Theorem: In any mutual exclusion algorithm guaranteeing bounded bypass, using a single RMW shared variable, the variable must be able to take on at least n distinct values.
- Proof, cont'd:
  - Now apply the same steps after α_j.
  - Result is an execution in which process 1 $\rightarrow$ C infinitely many times, while process j remains in T.
  - Violates bounded bypass.



  - Note: The extension of α_j isn't a fair execution; this is OK since fairness isn't required to violate bounded bypass.

Mutual exclusion + lockout-freedom

- Can solve with $O(n)$ values.
  - Actually, can achieve $n/2 + k$, small constant k.
- Lower bound of $\Omega(\sqrt{n})$ values.
  - Actually, about $\sqrt{n}$.
  - Uses a more complicated version of the construction for the bounded bypass lower bound.

Next time:

- More practical mutual exclusion algorithms
- Reading:
  - Herlihy, Shavit book, Chapter 7
  - Mellor-Crummey and Scott paper (Dijkstra prize winner)
  - (Optional) Magnussen, Landin, Hagersten paper
- Generalized resource allocation and exclusion problems
- Reading:
  - Distributed Algorithms, Chapter 11

6.852J / 18.437J Distributed Algorithms
Fall 2009