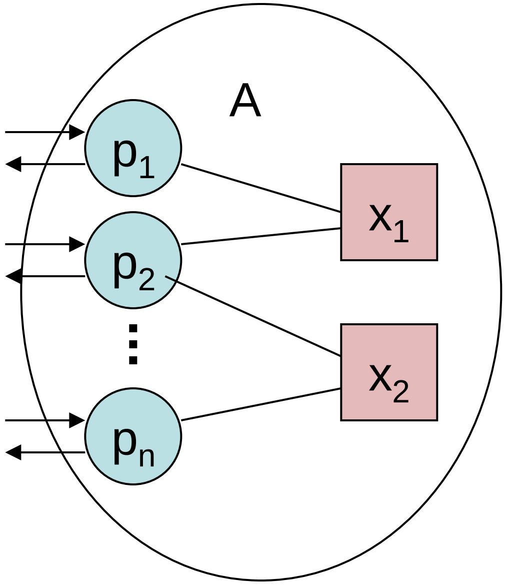
# 6.852: Distributed Algorithms
# Fall, 2015

Lecture 15

# Today's plan

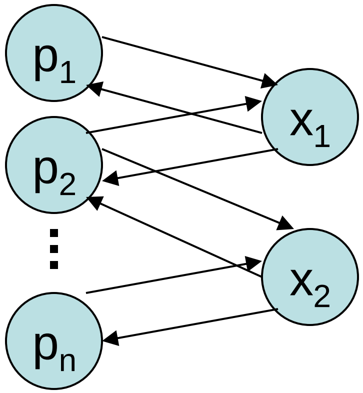
- Asynchronous shared-memory system model
- The Mutual Exclusion problem
- Dijkstra's algorithm
- Peterson's algorithms
- Lamport's Bakery algorithm
- Reading:  Chapter 9, Sections 10.1-10.5, 10.7
- Next:
  - More Mutual Exclusion algorithms
  - Lower bound on the number of shared variables
  - Resource allocation
- Reading:  Sections 10.6,10.8, Chapter 11 (skim)

# Asynchronous Shared-Memory Systems



# Asynchronous Shared-Memory Systems

- We've covered basics of non-fault-tolerant asynchronous network algorithms:
  - How to model them.
  - Basic asynchronous network protocols---broadcast, spanning trees, leader election,…
  - General methods for designing asynchronous network algorithms:
    - Synchronizers
    - Logical time
    - Global snapshots
- Now consider asynchronous shared-memory systems:
- Processes, interacting via shared objects, possibly subject to some access constraints.
- Shared objects have types, e.g.:
  - Read/write (weak)
  - Read-modify-write, compare-and-swap (strong)
  - Queues, stacks, others (in between)



# Asynch Shared-Memory systems

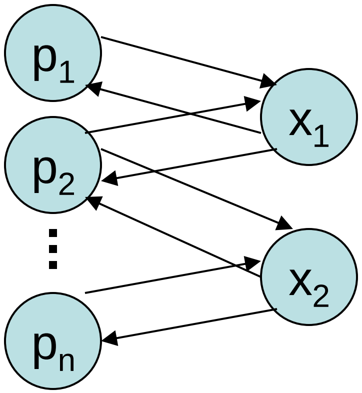
- Theory of ASM systems has much in common with theory of asynchronous networks:
  - Similar algorithms and impossibility results.
  - Even with failures.
  - Transformations from ASM model to asynch network model allow ASM algorithms to run in asynchronous networks.
    - “Distributed shared memory”.
- Historically, theory for ASM began first.
- Arose long ago, in study of early operating systems, in which several processes run on a single processor, sharing memory, with possibly-arbitrary interleavings of steps.
- Currently, ASM models are used to describe multiprocessor shared-memory systems, in which several processes run on separate processors and share memory.

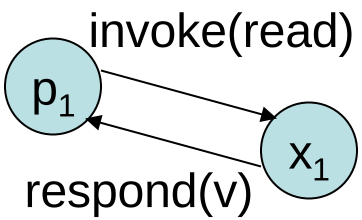
# Topics

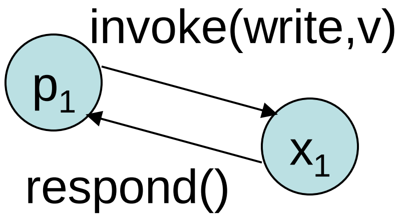
- Define the basic system model, without failures.
- Basic problems:
  - Mutual Exclusion.
  - Other resource-allocation problems (briefly).
- Then, introduce process failures into the model.
- More basic problems:
  - Distributed consensus
  - Implementing atomic objects:
    - Atomic snapshot objects
    - Atomic read/write registers
- Wait-free and fault-tolerant computability theory

# Basic ASM Model, Version 1

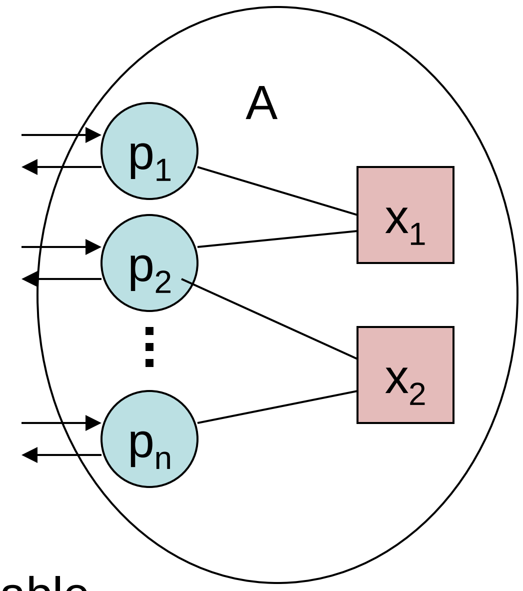
- Processes + objects, modeled as automata.
- Arrows:
  - Represent invocations and responses for operations on the objects.
  - Modeled as input and output actions.
- Fine-granularity model, can describe:
  - Delay between invocation and response.
  - Concurrent (overlapping) operations:
    - Object could reorder operations.
    - Could allow them to run concurrently, interfering with each other.
- We'll begin with a simpler, coarser model:
  - Object runs ops in invocation order, one at a time.
  - In fact, collapse each operation into a single step.
- Return to the finer model next week.



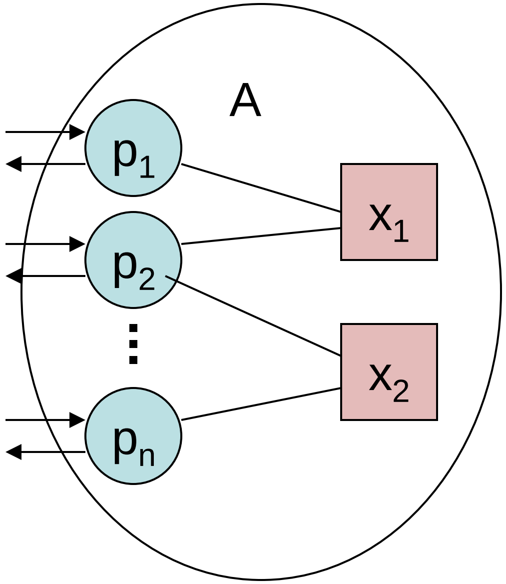



# Basic ASM Model, Version 2




- One big shared memory system automaton A.
- External actions at process "ports".
- Each process i has:
  - A set $states_i$ of states.
  - A subset $start_{i,}$ of start states.
- Each variable x has:
  - A set $values_x$ of values it can take on.
  - A subset $initial_x$ of initial values.
- Automaton A:
  - States: State for each process, a value for each variable.
  - Start: Start states, initial values.
  - Actions: Each action associated with one process, and some also with a single shared variable.
  - Input/output actions: At the external boundary.
  - Transitions: Correspond to local process steps and variable accesses.
    - Action enabling, which variable is accessed, depend only on process state.
    - Changes to variable and process state depend also on variable value.
    - Must respect the type of the variable.
  - Tasks: One or more per process (threads).

# Basic ASM Model




- Execution of A:
  - As specified by general definitions of executions, fair executions for I/O automata.
  - By fairness definition, each task gets infinitely many chances to take steps.
  - Model environment as a separate automaton, to express restrictions on environment behavior.
- Commonly-used variable types:
  - Read/write registers: Most basic object type.
    - Allows access using separate read and write operations.
  - Read-modify-write: Most powerful object type:
    - Atomically, read variable, do local computation, write to variable.
  - Compare-and-swap, fetch-and-add, queues, stacks, etc.
- Different computability and complexity results hold for different variable types.

# The Mutual Exclusion Problem

# The Mutual Exclusion Problem

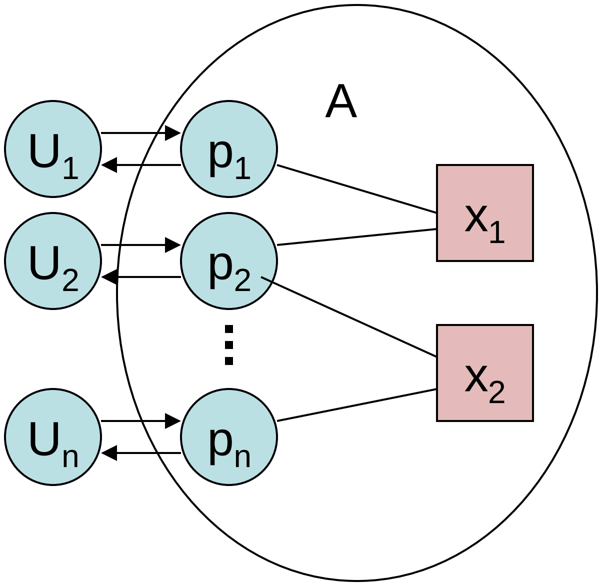
- Share one resource among n user processes, $U_1$, $U_2$,…,$U_n$.
  - E.g., printer, portion of a database.
- $U_i$ has four "regions".
  - Subsets of its states, described by portions of its code.
  - C critical; R remainder; $\underbrace{\text{T trying; E exit}}$

Protocols for obtaining and relinquishing the resource

- Cycle:

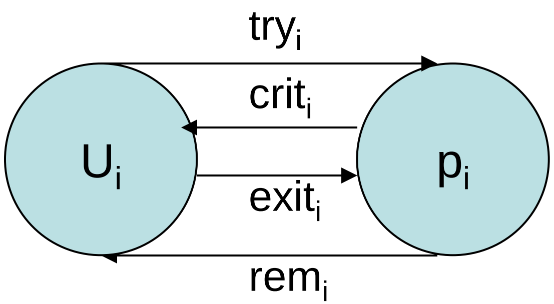
$$
\begin{tikzcd}
R \arrow[r] & T \arrow[r] & C \arrow[r] & E \arrow[lll, bend left]
\end{tikzcd}
$$

- Architecture:
  - $U_i$s and A are IOAs, compose.

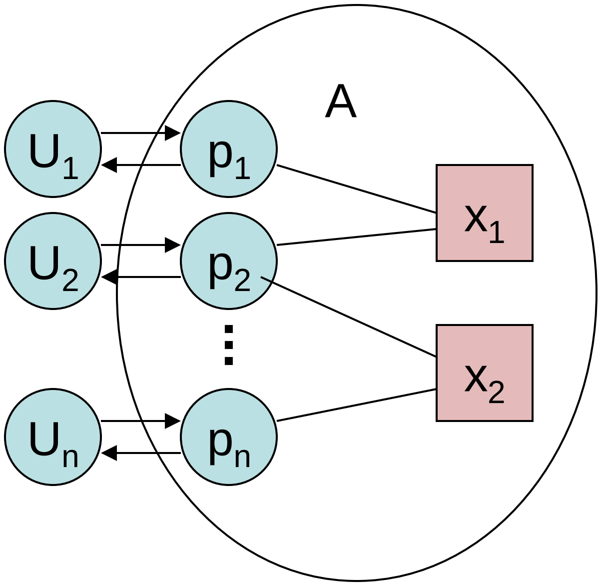


# The Mutual Exclusion Problem

- Actions at user interface:
  - Connect $U_i$ to $P_i$
  - $p_i$ is $U_i$'s "agent"

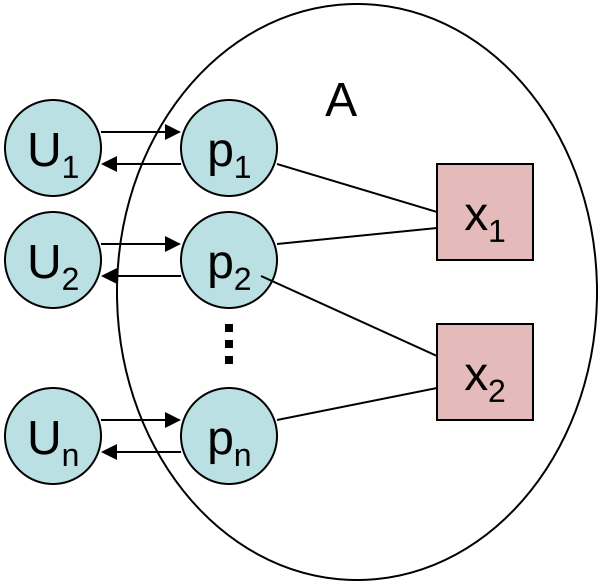



- Correctness conditions:
  - Well-formedness (Safety):
    - System also obeys cyclic discipline.
    - E.g., doesn't grant resource when it wasn't requested.
  - Mutual exclusion (Safety):
    - System never grants to > 1 user simultaneously.
    - Trace safety property.
    - Or, there's no reachable system state in which >1 user is in C at once.
  - Progress (Liveness):
    - From any point in a fair execution:
      - If some user is in T and no user is in C then at some later point, some user enters C.
      - If some user is in E then at some later point, some user enters R.



# The Mutual Exclusion Problem

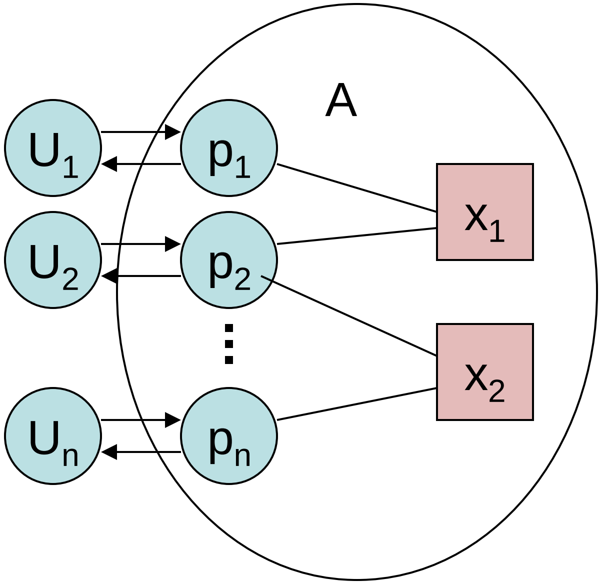
- Well-formedness (Safety):
  - System obeys cyclic discipline.
- Mutual exclusion (Safety):
  - System never grants to > 1 user.
- Progress (Liveness):
  - From any point in a fair execution:
    - If some user is in T and no user is in C then at some later point, some user enters C.
    - If some user is in E then at some later point, some user enters R.




- Conditions constrain only system automaton A, not users.
  - System determines if/when users enter C and R.
  - Users determine if/when users enter T and E.
  - We don't state any requirements on the users, except that users respect well-formedness.

# The Mutual Exclusion Problem

- Well-formedness (Safety):
- Mutual exclusion (Safety):
- Progress (Liveness):
  - From any point in a fair execution:
    - If some user is in T and no user is in C then at some later point, some user enters C.
    - If some user is in E then at some later point, some user enters R.




- Fairness assumption:
  - Progress condition requires fairness assumption (all process tasks continue to get turns to take steps).
  - Needed to guarantee that some user enters C or R.
  - In general, in the asynchronous model, liveness properties require fairness assumptions.
  - Contrast:  Well-formedness and mutual exclusion are safety properties, don’t depend on fairness.

# One more assumption…

- No permanently active processes.
  - Locally-controlled actions enabled only when user is in T or E.
  - No always-awake, dedicated processes.
  - Motivation:
    - Multiprocessor settings, where users can run processes when needed, but are otherwise not involved in the protocol.
    - Avoid “wasting a processor”.

# Dijkstra's Mutual Exclusion Algorithm

[Dijkstra 65]

# Mutual Exclusion algorithm

- Based on Dekker's 2-process solution.
- Pseudocode, p. 265-266
  - Written in traditional sequential style, then translated into more detailed state/transition description.
- Shared variables: Read/write registers.
  - turn, in {1,2,…,n}, multi-writer, multi-reader (mWmR), initially any
  - for each process i:
    - flag(i), in {0,1,2}, single-writer, multi-reader (1WmR), initially 0
    - Written by i, read by everyone.
- Process i's Stage 1:
  - Set flag := 1, test to see if turn = i.
  - If not, and turn's current owner is seen to be inactive, then set turn := i.
  - Otherwise go back to to testing…
  - When you see turn = i, move to Stage 2.

# Dijkstra's algorithm

- Stage 2:
  - Set flag(i) := 2.
  - Check (one at a time, any order) that no other process has flag = 2.
  - If check completes successfully, go to C.
  - If not, go back to the beginning of Stage 1.
- Exit protocol:
  - Set flag(i) := 0.
- Problem with the sequential code style:
  - Unclear what constitutes an atomic step.
    - E.g., need three separate steps to test turn, test flag(turn), and set turn.
  - Must rewrite to make this clear:
    - E.g., precondition/effect code (p. 268-269)
    - E.g., sequential-style code with explicit reads and writes, one per line.

# Dijkstra's algorithm, pre/eff code

- One transition definition for each kind of atomic step.
- Explicit program counter, pc.
- Transitions:
  - $\text{set-flag-1}_i$: Sets flag to 1 and prepares to test turn.
  - $\text{test-turn}_i$: Tests turn, and either moves to Stage 2 or prepares to test the current owner's flag.
  - $\text{test-flag(j)}_i$: Tests j's flag, and either goes on to set turn or goes back to test turn again.
  - …
  - $\text{set-flag-2}_i$: Sets flag to 2 and initializes set S, preparing to check all other processes' flags.
  - $\text{check(j)}_i$: If flag(j) = 2, go back to beginning.
  - …

- S keeps track of which processes have been successfully checked in Stage 2.

# Precondition/effect code

## Shared variables:

turn $\in$ {1,…,n}, initially arbitrary

for every i:

flag(i) $\in$ {0,1, 2}, initially 0

## Actions of process i:

Input: $try_i$, $exit_i$

Output: $crit_i$, $rem_i$

Internal: $set\text{-}flag\text{-}1_i$, $test\text{-}turn_i$, $test\text{-}flag(j)_i$, $set\text{-}turn_i$, $set\text{-}flag\text{-}2_i$, $check(j)_i$, $reset_i$

# Precondition/effect code, Dijkstra process i

**$try_i$:**

Eff: pc := set-flag-1

**$set\text{-}flag\text{-}1_i$ :**

Pre: pc = set-flag-1

Eff: flag(i) := 1

pc := test-turn

**$test\text{-}turn_i$ :**

Pre: pc = test-turn

Eff: if turn = i then pc := set-flag-2

else pc := test-flag(turn)

**$test\text{-}flag(j)_i$**

Pre: pc = test-flag(j)

Eff: if flag(j) = 0 then pc := set-turn

else pc := test-turn

**$set\text{-}turn_i$ :**

Pre: pc = set-turn

Eff: turn := i

pc := set-flag-2

**$set\text{-}flag\text{-}2_i$ :**

Pre: pc = set-flag-2

Eff: flag(i) := 2

S := {i}

pc := check

# More code, Dijkstra process i

$check(j)_i$ :

Pre: pc = check

$j \notin S$

Eff: if flag(j) = 2 then

S := $\varnothing$

pc := set-flag-1

else

S := S $\cup$ {j}

if |S| = n then pc := leave-try

$crit_i$ :

Pre: pc = leave-try

Eff: pc := crit

$exit_i$

Eff: pc := reset

$reset_i$ :

Pre: pc = reset

Eff: flag(i) := 0

S := $\varnothing$

pc := leave-exit

$rem_i$ :

Pre: pc = leave-exit

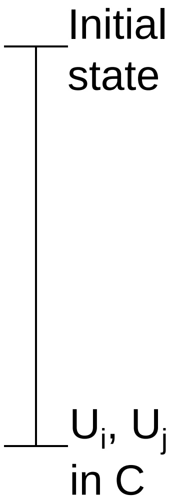
Eff: pc := rem

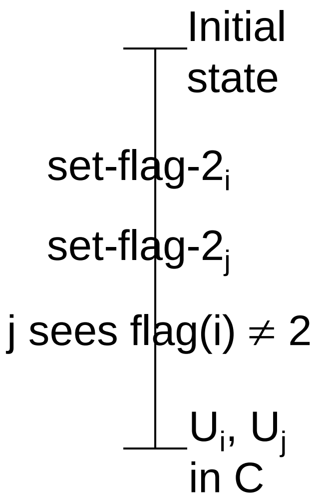
# Note on code style

- Explicit pc makes atomicity clear, but may look awkward.
- pc is often needed in invariants.
- Alternative idea:
  - Use sequential style, with explicit reads or writes (or other operations), one per line.
  - Need line numbers:
    - Play same role as pc.
    - Used in invariants: "If process i is at line 7 then…"

# Correctness

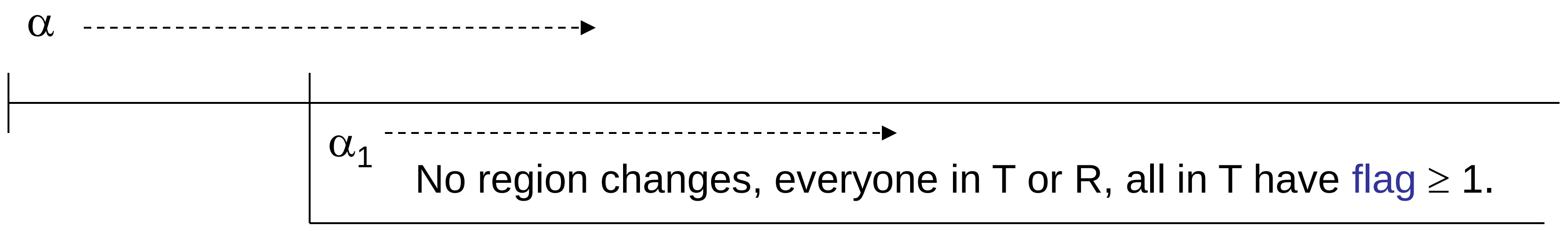
- Well-formedness: Obvious.
- Mutual exclusion:
  - Argue using event order in executions, instead of invariants as usual.
  - By contradiction: Assume $U_i$, $U_j$ find themselves in C at the same time.
  - Both must set-flag-2 before entering C; consider the last time they do this.
  - WLOG, suppose set-flag-$2_i$ comes first.
  - Then flag(i) = 2 from that point onward (until the assumed point when they are both in C).
  - However, j must see flag(i) $\neq$ 2, in order to enter C.
  - Impossible.



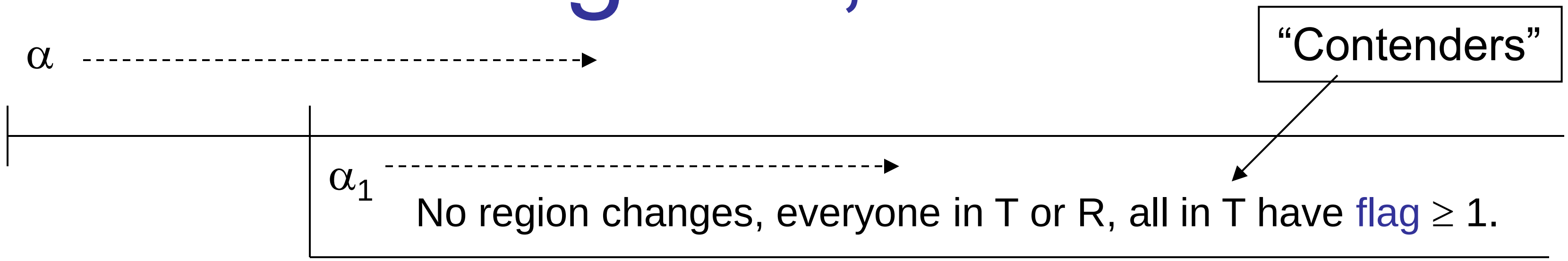



# Progress

- Interesting case: Trying region.
- Proof by contradiction:
  - Suppose $\alpha$ is a fair execution, reaches a point where some process is in T, no process is in C, and thereafter, no process ever enters C.
  - Now start removing complications…
  - Eventually, all region changes stop and all in T keep their flags $\geq 1$.
  - Then it must be that everyone is in T and R, and all in T have flag $\geq 1$.



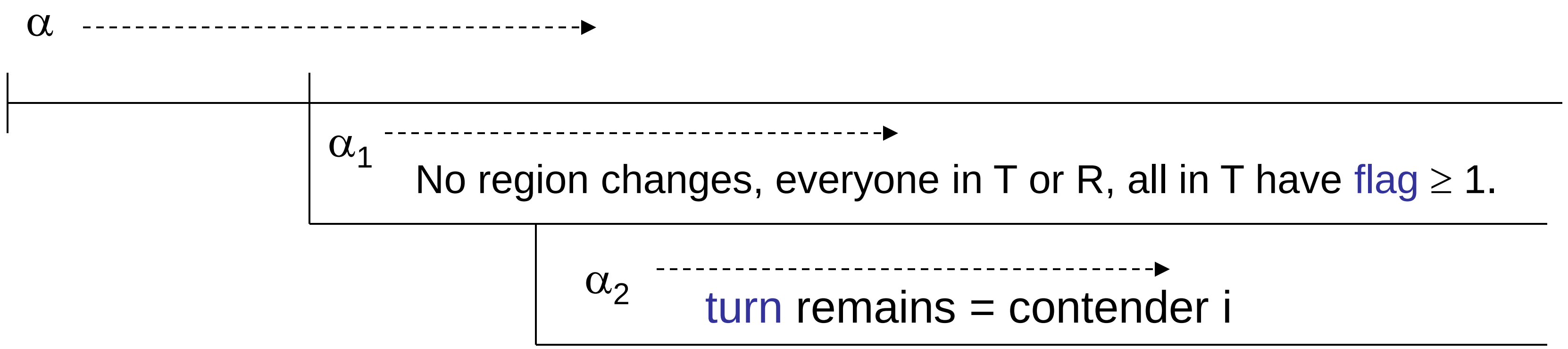
# Progress, cont'd




- Then whenever turn is reset in $\alpha_1$, it must be set to a contender's index.
- Claim: In $\alpha_1$, turn eventually acquires a contender's index.
- Proof:
  - Suppose not---it stays non-contender forever.
  - Consider any contender i.
  - If it ever reaches test-turn, then it will set turn := i, since it sees a non-contending process, yielding a contradiction.
  - Why must process i reach test-turn?
    - Either that, or it succeeds in reaching C.
    - But we have assumed no one reaches C.

# Progress, cont'd

- In $\alpha_1$, once turn = contender's index, it is thereafter always = some contender's index.
  - Because contenders are the only processes that can change turn.
- May change several times.
- Eventually, turn stops changing (because tests come out negative), stabilizes to some contender index, say i.




- Thereafter, all contenders $j \neq i$ wind up looping in Stage 1.
  - If j reaches Stage 2, it returns to Stage 1, since it doesn't go to C.
  - But then j's tests always fail, so j stays in Stage 1.
- But then nothing stops process i from entering C.

# Mutual exclusion, Proof 2

- Use invariants.
- Must show they hold after any number of steps.
- Main goal invariant: $| \{i : pc_i = crit \} | \leq 1$.
- To prove by induction, need more:
  1. If $pc_i$ = crit (or leave-try or reset) then $|S_i| = n$.
  2. There do not exist i, j, $i \neq j$, with i in $S_j$ and j in $S_i$.
- 1 and 2 easily imply mutual exclusion.
- Proof of 1: Easy induction
- Proof of 2:
  - Needs some easy auxiliary invariants saying what S-values go with what flag values and what pc values.
  - Key step: When j gets added to $S_i$, by $check(j)_i$ event.
    - Then must have $flag(j) \neq 2$.
    - But then $S_j = \emptyset$ (by auxiliary invariant), so $i \notin S_j$, can't break invariant.

# Running Time

- Upper bound on time from when some process is in T until some process is in C.
- Assume upper bound of l on successive turns for each process task (here, all steps of each process are in one task).
- Time upper bound for [Dijkstra]:  O(l n).
- Proof:  LTTR (see p. 275)

# Mutual Exclusion Algorithms with Fairness Guarantees

[Peterson]

# Adding fairness guarantees

- Dijkstra's algorithm does not guarantee fairness in granting the resource to different users.
- This might not be important in practice, if contention is rare.
- Other theoretical algorithms add fairness guarantees.
- [Peterson]: a suite of algorithms guaranteeing lockout-freedom.
- Lockout-freedom: In any (low-level) fair execution:
  - If all users always return the resource then any user that enters T eventually enters C.
  - Any user that enters E eventually enters R.

# Peterson 2-process algorithm

- Shared variables:
  - turn, in {0,1}, 2W2R read/write register, initially arbitrary.
  - for each process i in {0,1}:
    - flag(i), in {0,1}, 1W1R register, initially 0
    - Written by i, read by 1-i.

- Process i's trying protocol:
  - Sets flag(i) := 1, sets turn := i.
  - Waits for either flag(1-i) = 0 or turn $\neq$ i.

    | Other process not active. | Other process has the turn variable. |
    | --- | --- |

  - Toggles between the two tests.

- Exit protocol:
  - Sets flag(i) := 0

# Precondition/effect code

## Shared variables:

turn $\in$ {0,1}, initially arbitrary

for every i $\in$ {0,1}:

  flag(i) $\in$ {0,1}, initially 0

## Actions of process i:

Input: $try_i$, $exit_i$

Output: $crit_i$, $rem_i$

Internal: $set\text{-}flag_i$, $set\text{-}turn_i$, $check\text{-}flag_i$, $check\text{-}turn_i$, $reset_i$

# Precondition/effect code, Peterson 2P, process i

**$try_i$**

Eff: pc := set-flag

**$set\text{-}flag_i$**

Pre: pc = set-flag

Eff: flag(i) := 1

pc := set-turn

**$set\text{-}turn_i$**

Pre: pc = set-turn

Eff: turn := i

pc := check-flag

**$check\text{-}flag_i$**

Pre: pc = check-flag

Eff: if flag(1-i) = 0 then pc := leave-try

else pc := check-turn

**$check\text{-}turn_i$**

Pre: pc = check-turn

Eff: if turn $\neq$ i then pc := leave-try

else pc := check-flag

# More code, Peterson 2P, process i

$crit_i$

Pre: pc = leave-try

Eff: pc := crit

$exit_i$

Eff: pc := reset

$reset_i$

Pre: pc = reset

Eff: flag(i) := 0

pc := leave-exit

$rem_i$

Pre: pc = leave-exit

Eff: pc := rem

# Correctness: Mutual exclusion

- Key invariant:
  - If $pc_i \in$ {leave-try, crit, reset} (essentially in C), and $pc_{1-i} \in$ {check-flag, check-turn, leave-try, crit, reset} (engaged in the competition or has won the competition), then $turn \neq i$.
- That is:
  - If i has won and 1-i is currently competing then turn is set favorably for i---which means it is set to 1-i.
- Implies mutual exclusion: If both are in C then turn must be set both ways, contradiction.
- Proof of invariant: All cases of inductive step are easy.
  - E.g.: a successful $check\text{-}turn_i$, causing i to advance to leave-try.
  - This explicitly checks that $turn \neq i$, as needed.

# Correctness: Progress

- By contradiction:
  - Suppose someone is in T, and no one is ever thereafter in C.
  - Then the execution eventually stabilizes so no new region changes occur.
  - After stabilization:
    - If exactly one process is in T, then it sees the other's flag = 0 and enters C.
    - If both processes are in T, then turn is set favorably to one of them, and it enters C.

# Correctness: Lockout-freedom

- Argue that neither process can enter C three times while the other stays in T, after setting its flag := 1.
- Bounded bypass.
- Proof: By contradiction.
  - Suppose process i is in T and has set flag := 1, and subsequently process (1-i) enters C three times.
  - In each of the second and third times through T, process (1-i) sets turn := 1-i but later sees turn = i.
  - That means process i must set turn := i at least twice during that time.
  - But process i sets turn := i only once during its one execution of T.
  - Contradiction.
- Bounded bypass + progress imply lockout-freedom.

# Time complexity

- Time from when any particular process i enters T until it enters C: c + O(l), where:
  - c is an upper bound on the time any user remains in the critical section, and
  - l is an upper bound on local process step time.
- Detailed proof: See book, p. 283.
- Rough idea:
  - Either process i either enters immediately, or has to wait for (1-i).
  - But in that case, it only has to wait for one critical-section time, since if (1-i) reenters, it will set turn favorably for i.

# Peterson n-process algorithms

- Extend 2-process algorithm for lockout-free mutual exclusion to an n-process algorithm, in two ways:
  - Using linear sequence of competitions, or
  - Using binary tree of competitions.

# Sequence of competitions

- Competitions 1,2,…,n-1.
- Competition k has one loser, up to n-k winners.
- Thus, only one can win in competition n-1, implying mutual exclusion.
- Shared vars:
  - For each competition k in {1,2,…,n-1}:
    - turn(k) in {1,2,…n}, mWmR register, written and read by all, initially arbitrary.
  - For i in {1,2,…n}:
    - flag(i) in {0,1,2,…,n-1}, 1WmR register, written by i and read by all, initially 0.
- Process i trying protocol:
  - For each level k:
    - Set flag(i) := k, indicating that i is competing at level k.
    - Set turn(k) := i.
    - Wait for either turn(k) $\neq$ i, or everyone else's flag < k (check flags one at a time).
- Exit protocol:
  - Set flag(i) := 0

# Correctness: Mutual exclusion

- Definition: Process i is a winner at level k if either:
  - $level_i > k$, or
  - $level_i = k = n-1$ and $pc_i \in$ {leave-try, crit, reset}.
- Definition: Process i is a competitor at level k if either:
  - Process i is a winner at level k, or
  - $level_i = k$ and $pc_i \in$ {check-flag, check-turn}.

- Invariant 1: If process i is a winner at level k, and process $j \neq i$ is a competitor at level k, then $turn(k) \neq i$.
- Proof: By induction, similar to 2-process case.
  - Complication: More steps to consider.
  - Now have many flags, checked in many steps.
  - Need auxiliary invariants saying something about what is true in the middle of checking a set of flags.

# Correctness: Mutual exclusion

- Invariant 2: For any k, $1 \leq k \leq n-1$, there are at most n-k winners at level k.

- Proof: By induction, on level number, for a particular reachable state (not induction on number of steps).
  - Basis: k = 1:
    - Suppose false, for contradiction.
    - Then all n processes are winners at level 1.
    - Then Invariant 1 implies that turn(1) is unequal to all indices, contradiction.
  - Inductive step: Assume for k, $1 \leq k \leq n-2$, show for k+1.
    - …

# Correctness: Mutual exclusion

- Invariant 2: For any k, $1 \leq k \leq n-1$, there are at most n - k winners at level k.
- Inductive step: Assume for k, $1 \leq k \leq n-2$, show for k+1.
  - Suppose false, for contradiction.
  - Then more than n – (k + 1) processes, that is, at least n – k processes, are winners at level k + 1: $|\ Win_{k+1}\ | \geq n - k$.
  - Every level k+1 winner is also a level k winner: $Win_{k+1} \subseteq Win_k$.
  - By inductive hypothesis, $|\ Win_k\ | \leq n-k$.
  - So $Win_{k+1} = Win_k$, and $|\ Win_{k+1}\ | = |\ Win_k\ | = n - k$.
  - Q: What is the value of turn(k+1) ?
    - Can't be the index of any process in $Win_{k+1}$, by Invariant 1.
    - Must be the index of some competitor at level k+1 (Invariant, LTTR).
    - But every competitor at level k+1 is a winner at level k, so is in $Win_k$.
    - Contradiction, since $Win_{k+1} = Win_k$.

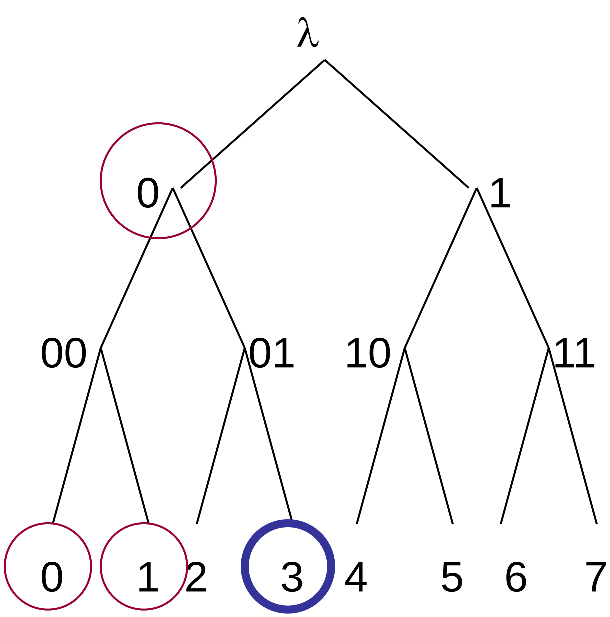
# Progress, Lockout-freedom

- Lockout-freedom proof idea:
  - Let k be the highest level at which some process, say i, gets stuck.
  - Then turn(k) must remain = i.
  - That means no one else ever reenters the competition at level k.
  - Eventually, winners from level k will finish, since k is the highest level at which anyone gets stuck.
  - Then all other flags will be < k, so i advances.

- Alternatively, prove lockout-freedom by showing a time bound for each process, from $\rightarrow$T until $\rightarrow$C. (See book)
  - Define T(0) = maximum time from when a process $\rightarrow$T until $\rightarrow$C.
  - Define T(k), $1 \leq k \leq n-1$ = max time from when a process wins at level k until $\rightarrow$C.
  - $T(n-1) \leq l$.
  - $T(k) \leq 2\ T(k+1) + c + (3n+2)\ l$, by detailed analysis.
  - Solve recurrences, get exponential bound, good enough for showing lockout-freedom.

# Peterson Tournament Algorithm

- Assume $n = 2^h$.
- Processes = leaves of binary tree of height h.
- Competitions = internal nodes, labeled by binary strings.
- Each process engages in log n competitions, following path up to root.
- Each process i has:
  - A unique competition x at each level k.
  - A unique role in x (0 = left, 1 = right).
  - A set of potential opponents in x.

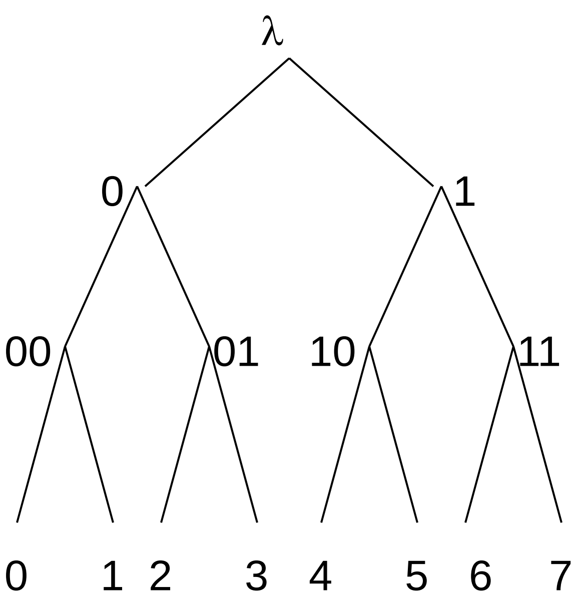


# Peterson Tournament Algorithm

- Shared variables:
  - For each process i, flag(i) in {0,…,h}, indicating level, initially 0
  - For each competition x, turn(x), a Boolean, initially arbitrary.

- Process i's trying protocol: For each level k:
  - Set flag(i) := k.
  - Set turn(x) := b, where:
    - x is i's level k competition,
    - b is i's "role", 0 or 1
  - Wait for either:
    - turn(x) = opposite role, or
    - all flags of potential opponents in x are < k.

- Exit protocol:
  - Set flag(i) := 0.



# Correctness

- Mutual exclusion:
  - Similar to before.
  - Key invariant: At most one process from any particular subtree rooted at level k is currently a winner at level k.

- Time bound (from $\rightarrow$T until $\rightarrow$C): $(n-1)\, c + O(n^2\, l)$
  - Implies progress, lockout-freedom.
  - Define: $T(0)$ = max time from $\rightarrow$T until $\rightarrow$C.
  - $T(k)$, $1 \leq k \leq \log n$ = max time from winning at level k until $\rightarrow$C.
  - $T(\log n) \leq l$.
  - $T(k) \leq 2\, T(k+1) + c + (2^{k+1} + 2^k + 7)\, l$ (see book).
    - Roughly: Might need to wait for a competitor to reach C, then finish C, then for yourself to reach C.
  - Solve recurrences.

# Bounded Bypass?

- Peterson's Tournament algorithm has a low time bound from $\rightarrow$T until $\rightarrow$C:

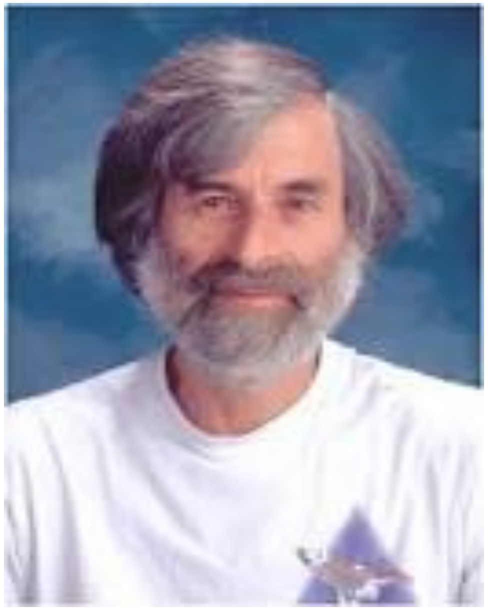
  $$(n - 1)\, c + O(n^2\, l)$$

- Implies lockout-freedom, progress.

- Q: Does it satisfy bounded bypass?
- No! There's no upper bound on the number of times one process could bypass another in the trying region. E.g.:
  - Process 0 enters, starts competing at level 1, then pauses.
  - Process 7 enters, quickly works its way to the top, enters C, leaves C.
  - Process 7 enters again…repeats any number of times.
  - All while process 0 is paused.

- No contradiction between small time bound and unbounded bypass.
  - Because of the way we're modeling timing of asynchronous executions, using upper bound assumptions.
  - When processes go at very different speeds, we say that the slow processes are going at normal speed, faster processes are going very fast.

# Lamport's Bakery Algorithm



# Lamport's Bakery Algorithm

- Like taking tickets for service in a bakery.
- Nice features:
  - Uses only single-writer, multi-reader registers.
  - Extends to even weaker ("safe") 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 of 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 > 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 in a 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 ≠ i} number(j)
choosing(i) := 0
for j ≠ 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

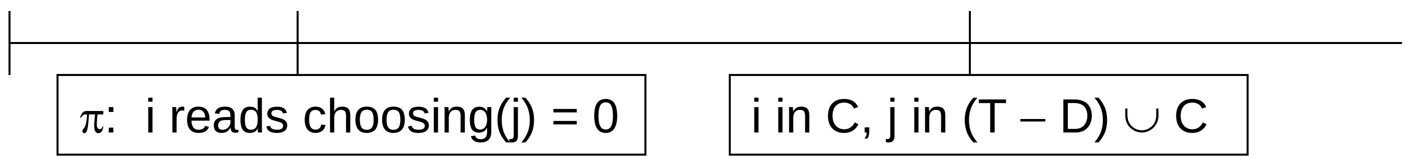
- Key invariant: If process i is in C, and process $j \neq i$ is in $\underbrace{(T - D) \cup C}_{\text{Trying region after doorway, or critical region}}$,

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

- Proof:
  - Could prove by induction.
  - Instead, give argument based on events in executions.

# 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 $\pi$.




  - Case 1: j sets choosing(j) := 1 (starts choosing) after $\pi$.
    - Then number(i) is set before j starts choosing.
    - So j sees the "correct" number(i) and chooses something bigger.
    - That suffices.
  - Case 2: j sets choosing(j) := 0 (finishes choosing) before $\pi$.
    - Then when i reads number(j) in its second waitfor loop, it gets the "correct" number(j).
    - Since i decides to enter C, it must see (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 $(number(i),i) < (number(j),j)$.
- 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.
  - Suppose for contradiction that i never reaches C.
  - Eventually it finishes the doorway.
  - Thereafter, any newly-entering process picks a bigger number.
  - Progress implies that some 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 some 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.

# Next time…

- More Mutual Exclusion algorithms:
  - Burns' algorithm
- Number of registers needed for mutual exclusion.
- Reading:
  - Sections 10.6,10.8
- Generalized resource allocation and exclusion problems
- Reading:
  - Chapter 11