Verifying Concurrent Programs
Advanced Critical Section Solutions

Ch 4.1-3, App B [BenA 06]
Ch 5 (no proofs) [BenA 06]

Propositional Calculus
Invariants
Temporal Logic
Automatic Verification
Bakery Algorithm & Variants
Propositional Calculus

- Atomic propositions
  - A, B, C, …
  - True (T) or False (F)
- Operators
  - not
  - disjunction, or
  - conjunction, and
  - implication
  - equivalence

Boolean algebra

<table>
<thead>
<tr>
<th></th>
<th>(v(A_1))</th>
<th>(v(A_2))</th>
<th>(v(A))</th>
</tr>
</thead>
<tbody>
<tr>
<td>(\neg A_1)</td>
<td>T</td>
<td></td>
<td>F</td>
</tr>
<tr>
<td>(\neg A_1)</td>
<td>F</td>
<td></td>
<td>T</td>
</tr>
<tr>
<td>(A_1 \lor A_2)</td>
<td>F</td>
<td>F</td>
<td>F</td>
</tr>
<tr>
<td>(A_1 \land A_2)</td>
<td>T</td>
<td>T</td>
<td>T</td>
</tr>
<tr>
<td>(A_1 \rightarrow A_2)</td>
<td>T</td>
<td>F</td>
<td>F</td>
</tr>
<tr>
<td>(A_1 \leftrightarrow A_2)</td>
<td>(v(A_1) = v(A_2))</td>
<td></td>
<td>T</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

(propositiolaskenta, propositiologiikka totuusarvoilla laskeminen)

atominen propositio, tilapropositio

4.11.2008
Copyright Teemu Kerola 2008
Propositional Calculus

• Implication
  – Premise or antecedent
  – Conclusion or consequent

• Formula
  – Atomic proposition
  – Atomic propositions or formulaes combined with operators

• Assignment $v(f)$ of formula $f$
  – Assigned values (T or F) for each atomic proposition in formula
  – Interpretation $v(f)$ of formula $f$ computed with operator rules
  – Formula $f$ is $\text{true}$ if $v(f) = T$, $\text{false}$ if $v(f) = F$
Propositional Calculus

• Formula
  – Implication
    • Premise or antecedent
    • Conclusion or consequent
  – Formula $f$ is true/false if it’s interpretation $v(f)$ is true/false
    • Given assignment values for each argument
  – Formula is **valid** if it is tautology
    • Always true for all interpretations (all atomic propos. values)
  – Formula is **satisfiable** if true in some interpretation
  – Formula is **falsifiable** if sometimes false
  – Formula is **unsatisfiable** if always false

\[(A_1 \land A_2 \land \cdots \land A_n) \rightarrow B\]
Methods for Proving Formulae Valid

• Induction proof $F(n)$ for all $n=1, 2, 3, \ldots$
  - $F(1)$
  - $F(n) \rightarrow F(n+1)$

• Dual approach: $f$ is valid $\leftrightarrow \neg f$ is unsatisfiable
  - Find one interpretation that makes $\neg f$ true
  - Go through (automatically) all interpretations of $\neg f$
  - If such interpretation found, $\neg f$ is satisfiable, i.e., $f$ is not valid
  - O/w $f$ is valid

• Proof by contradiction
  - Assume: $f$ is not valid
  - Deduce contradiction with propositional calculus
    $\neg X \land X$
Methods for Proving Formulaes Valid

• Deductive proof
  - Deduce formula from axioms and existing valid formulaes
  - Start from the “beginning”

• Material implication
  - Formula is in the form “p → q”
  - Can show that “¬(p → q)” can not be (or can not become): \( v(p) = T \) and \( v(q) = F \)
    • if \( v(p) = v(q) = T \) and \( v(q) \) becomes \( F \), then \( v(p) \) will not stay \( T \)
    • if \( v(p) = v(q) = F \) and \( v(p) \) becomes \( T \),
Correctness of Programs

- **Program P is partially correct**
  - If P halts, then it gives the correct answer

- **Program P is totally correct**
  - P halts and it gives the correct answer
  - Often very difficult to prove (“halting problem” is difficult)

- Program P can have
  - preconditions $A(x_1, x_2, \ldots)$ for input values $(x_1, x_2, \ldots)$
  - postconditions $B(y_1, y_2, \ldots)$ for output values $(y_1, y_2, \ldots)$

- Partial and total correctness with respect to $A(\ldots)$ and $B(\ldots)$

More? Se courses on specification and verification
Verification of Concurrent Programs

- State diagrams can be very large
  - Can do them automatically
- Making conclusions on state diagrams is difficult
  - Mutex, no deadlock, no starvation?
  - Can do automatically with temporal logic based on propositional calculus
- Model checker programs (not covered in this course!)

Spin
STeP
mallin tarkastin
Atomic propositions

- **Boolean variables**
  - Consider them as atomic propositions
  - *Proposition* \( \text{wantp} \) is true, iff variable \( \text{wantp} \) is true in given state

- **Integer variables**
  - Comparison result is an atomic proposition
  - Example: proposition “\( \text{turn} \neq 2 \)” is true, iff variable \( \text{turn} \) value is not 2 in given state

- **Control pointers**
  - Comparison to given value is an atomic proposition
  - Example: proposition \( p1 \) is true, iff control pointer for \( P \) is \( p1 \) in given state

Idea: system state described with propositional logic
Formulaes

- **Formula**: $p_1 \land q_1 \land \neg \text{wantp} \land \neg \text{wantq}$
  - True only in the starting state

- **Formula**: $p_4 \land q_4$
  - True only if mutex is broken
  - Mutex condition can be defined: $\neg(p_4 \land q_4)$
    - Must be true in all possible states in all possible computations
    - Invariant

Algorithm 3.8: Third attempt

<table>
<thead>
<tr>
<th>p</th>
<th>q</th>
</tr>
</thead>
<tbody>
<tr>
<td>loop forever</td>
<td>loop forever</td>
</tr>
<tr>
<td>$p_1$: non-critical section</td>
<td>$q_1$: non-critical section</td>
</tr>
<tr>
<td>$p_2$: wantp $\leftarrow$ true</td>
<td>$q_2$: wantq $\leftarrow$ true</td>
</tr>
<tr>
<td>$p_3$: await wantq = false</td>
<td>$q_3$: await wantp = false</td>
</tr>
<tr>
<td>$p_4$: critical section</td>
<td>$q_4$: critical section</td>
</tr>
<tr>
<td>$p_5$: wantp $\leftarrow$ false</td>
<td>$q_5$: wantq $\leftarrow$ false</td>
</tr>
</tbody>
</table>

4.11.2008

Copyright Teemu Kerola 2008
Mutex Proof

- **Invariant** \(\neg (p4 \land q4)\)
  - If this is proven correct (true in all states), then mutex is proven
- **Inductive proof**
  - True for *initial state*
  - Assuming true for *current state*, prove that it still applies in *next state*
    - Consider only statements that affect propositions in invariant

<table>
<thead>
<tr>
<th>(p)</th>
<th>(q)</th>
</tr>
</thead>
<tbody>
<tr>
<td>non-critical section</td>
<td>non-critical section</td>
</tr>
<tr>
<td>wantp (\leftarrow) true</td>
<td>wantq (\leftarrow) true</td>
</tr>
<tr>
<td>await wantq = false</td>
<td>await wantp = false</td>
</tr>
<tr>
<td>critical section</td>
<td>critical section</td>
</tr>
<tr>
<td>wantp (\leftarrow) false</td>
<td>wantq (\leftarrow) false</td>
</tr>
</tbody>
</table>
Mutex Proof

• Invariant \( \neg(p4 \land q4) \)
  - Can not prove directly (yet) - too difficult

• Need proven Lemma 4.3
  - Lemma 4.1: \( p3..5 \rightarrow wantp \) is invariant
  - Lemma 4.2: \( wantp \rightarrow p3..5 \) is invariant
  - Lemma 4.3: \( p3..5 \leftrightarrow wantp \) and \( q3..5 \leftrightarrow wantq \) are invariants

• Can now prove original invariant \( \neg(p4 \land q4) \)
  - Inductive proof with Lemma 4.3
  - Details on next slide

<table>
<thead>
<tr>
<th>( )</th>
<th>loop forever</th>
<th>non-critical section</th>
</tr>
</thead>
<tbody>
<tr>
<td>( p1 )</td>
<td>wantp ( \leftarrow ) true</td>
<td></td>
</tr>
<tr>
<td>( p2 )</td>
<td>await wantq = false</td>
<td></td>
</tr>
<tr>
<td>( p3 )</td>
<td>critical section</td>
<td></td>
</tr>
<tr>
<td>( p4 )</td>
<td>wantp ( \leftarrow ) false</td>
<td></td>
</tr>
<tr>
<td>( p5 )</td>
<td>non-critical section</td>
<td></td>
</tr>
<tr>
<td>( q1 )</td>
<td>wantq ( \leftarrow ) true</td>
<td></td>
</tr>
<tr>
<td>( q2 )</td>
<td>await wantp = false</td>
<td></td>
</tr>
<tr>
<td>( q3 )</td>
<td>critical section</td>
<td></td>
</tr>
<tr>
<td>( q4 )</td>
<td>wantq ( \leftarrow ) false</td>
<td></td>
</tr>
</tbody>
</table>

Algorithm 3.8: Third attempt

boolean wantp \( \leftarrow \) false, wantq \( \leftarrow \) false

lemma, apulause
Mutex Proof

• **Lemma 4.3:** $p_{3..5} \iff \text{want}_p$ and $q_{3..5} \iff \text{want}_q$ invariants

• **Theorem 4.4:** $\neg(p_4 \land q_4)$ is invariant
  - Prove $(p_4 \land q_4)$ inductively false in every state
  - Initial state: trivial
  - Only states $\{p_3, \ldots\}$ need to be considered
    - $p_4$ may become true only here, i.e., state $\{p_4, q?, \ldots\}$
    - States $\{\ldots, q_3, \ldots\}$ similar, symmetrical
  - Can execute $\{p_3, \ldots\}$ only if $\text{want}_q=false$ (i.e., $\neg \text{want}_q$)
  - Because $\text{want}_q=false$, $q_4$ is also false (Lemma 4.3)
  - Next state can not be $\{p_4, q_4, \ldots\}$, i.e., $(p_4 \land q_4)$ is false

---

**Algorithm 3.8: Third attempt**

<table>
<thead>
<tr>
<th>$p$</th>
<th>$q$</th>
</tr>
</thead>
<tbody>
<tr>
<td>loop forever</td>
<td>loop forever</td>
</tr>
<tr>
<td>$p_1$: non-critical section</td>
<td>$q_1$: non-critical section</td>
</tr>
<tr>
<td>$p_2$: want$_p \leftarrow$ true</td>
<td>$q_2$: want$q \leftarrow$ true</td>
</tr>
<tr>
<td>$p_3$: await want$q = \text{false}$</td>
<td>$q_3$: await want$\text{p} = \text{false}$</td>
</tr>
<tr>
<td>$p_4$: critical section</td>
<td>$q_4$: critical section</td>
</tr>
<tr>
<td>$p_5$: want$\text{p} \leftarrow$ false</td>
<td>$q_5$: want$\text{q} \leftarrow$ false</td>
</tr>
</tbody>
</table>
Temporal Logic

- Propositional logic with **extra temporal operators**

- **Computation**
  - Infinite sequence of states: \( \{ s_0, s_1, s_2, \ldots \} \)

- **Temporal operators**
  - Value (T or F) of given predicate does not necessarily depend only on current state
    - It may depend on also on (some or all) future states
  - **Always or box (□) operator**
    - □A true in state \( s_i \) if A true in all \( s_j, j \geq i \)
    - E.g., mutex must always be true
  - **Eventually or diamond (◊) operator**
    - ◊A true in state \( s_i \) if A true in some \( s_j, j \geq i \)
    - E.g., no starvation means that something eventually will become true

\[ (p_2 \lor p_4) \land \neg (p_4 \land q_4) \]

\( \square (p_2 \rightarrow \Diamond p_4) \)
Other Temporal Logic Operators

• True in next state (O) operator
  - $\text{Op } p$ true in state $s_i$, if $p$ is true in the state $s_{i+1}$

• Until eventually (U) operator
  - $p U q$ true in state $s_i$, if $p$ is true in every state in future until eventually $q$ becomes true

• Not used (needed) in this course...

More? See courses on specification and verification.
Some Laws of Temporal Logic

- **deMorgan**
  \[ \neg(A \land B) \iff (\neg A \lor \neg B) \]
  \[ \neg(A \lor B) \iff (\neg A \land \neg B) \]

- **Distributive Laws**
  \[ \Box(A \land B) \iff (\Box A \land \Box B) \]
  \[ \Diamond(A \lor B) \iff (\Diamond A \lor \Diamond B) \]

- **Duality**
  - Not always is equivalent to eventually not
    \[ \neg \Box A \iff \Diamond \neg A \]
  - Not eventually is equivalent to always not
    \[ \neg \Diamond A \iff \Box \neg A \]
Sequence

- **Eventually always**
  - Will come true and then stays true forever

- **Always eventually**
  - Always will become true some times in future (again)
More Complex Proofs

- State diagrams become easily too large for manual analysis
- Use model checkers
  - Spin for Promela programs (algorithms)
  - Java PathFinder for Java programs
- More details?
  - Course
    - An Introduction to Specification and Verification
Advanced Critical Section Solutions

Ch 5 [BenA 06] (no proofs)

Bakery Algorithm

Bakery for N processes

Fast for N processes
**Bakery Algorithm**

- **Environment**
  - Shared memory, atomic read/write
  - No HW support needed
  - Short exclusive access code segments
    - Wait in busy loop (no process switch)

- **Goal**
  - Mutex and Customers served in request order
  - Independent (distributed) decision making

- **Solution idea**
  - Get queue number, service requests in ascending order

- **Possible problems**
  - Shared, distributed queuing machine, will it work?
  - Get same queue number as someone else? Problem?
  - Some number skipped? Problem or not?
  - Will numbers grow indefinitely (overflow)?

*(Leslie Lamport)*

Very strong requirement!
Bakery Algorithm (2 processes)

**Algorithm 5.1: Bakery algorithm (two processes)**

<table>
<thead>
<tr>
<th>p</th>
<th>q</th>
</tr>
</thead>
<tbody>
<tr>
<td>integer np ← 0, nq ← 0</td>
<td>loop forever</td>
</tr>
<tr>
<td>p1:</td>
<td>non-critical section</td>
</tr>
<tr>
<td>p2: np ← nq + 1</td>
<td>q1: non-critical section</td>
</tr>
<tr>
<td>p3:</td>
<td>await nq = 0 or np ≤ nq</td>
</tr>
<tr>
<td>p4:</td>
<td>critical section</td>
</tr>
<tr>
<td>p5:</td>
<td>np ← 0</td>
</tr>
<tr>
<td>q in non-critical section</td>
<td>q4: critical section</td>
</tr>
<tr>
<td></td>
<td>q in q3 or q4</td>
</tr>
</tbody>
</table>

- Can enter CS, if ticket (np or nq) is “smaller” than that of the other process
- Priority: if equal tickets, both compete, but P wins
  - Fixed priority not so good, but acceptable (rare occurrence)
Correctness Proof for 2-process Bakery Algorithm

• Mutex?
• No deadlock?
• No starvation?
• No counter overflow?

• What else, if any?
  - Temporal logic

Alg. 5.1

Spesifioinnin ja verifiioinnin perusteet
(Slides Conc.Progr. 2006)
(for those who really like temporal logic…)

4.11.2008
Bakery for n Processes

<table>
<thead>
<tr>
<th>Algorithm 5.2: Bakery algorithm (N processes)</th>
</tr>
</thead>
<tbody>
<tr>
<td>integer array[1..n] number ← [0,..,0]</td>
</tr>
</tbody>
</table>

| p1: non-critical section                |
| p2: number[i] ← 1 + max(number)         |
| p3: for all other processes j           |
| p4: await (number[j] = 0) or (number[i] ≪ number[j]) |
| p5: critical section                    |
| p6: number[i] ← 0                       |

- No **write** competition to shared variables
  - Load/store assumed atomic
- Ticket numbers increase continuously while critical section is taken – danger?
- All other processes polled
  - Not so good!

- when equality, give priority to smaller number[x]
- in non-critical section?
- in q3..q6?
Bakery for n Processes

• Mutex OK?
  – Yes, because of priorities at competition time

• Deadlock OK?
  – Yes, because of priorities at competition time

• Starvation OK?
  – Yes, because
    • Your (i) turn will come eventually
    • Others (j) will progress and leave CS
    • Next time their number[j] will be bigger than yours

• Overflow
  – Not good. Numbers grow unbounded if some process always in CS
    • Must have other information/methods to guarantee that this does not happen.

 e.q., max 100 processes, CS less than 0.01% of executed code ??
• Concurrent read & write may result to bad read
  – Correct behaviour in p7 even if number[j] value read wrong!
• Assuming that await is in busy loop

Performance Problems with Bakery Algorithm

- Problem
  - Lots of overhead work, if many concurrent processes
  - Check status for all possibly competing other processes
    - Other processes (not in CS) slow down the one process trying to get into CS – not good
  - Most of the time wasted work
    - Usually not much competition for CS

- How to do it better?
  - Check competition in fixed time
  - In a way not dependent on the number of possible competitors
  - Suffer overhead only when competition occurs
• Assume atomic read/write
• 2 shared variables, both read/written by P and Q
• Block at gate1, if contention
  – Last one to get there waits
• Access to CS, if success in writing own id to both gates

**Algorithm 5.4: Fast algorithm for two processes (outline)**

<table>
<thead>
<tr>
<th>p</th>
<th>q</th>
</tr>
</thead>
<tbody>
<tr>
<td>loop forever</td>
<td>loop forever</td>
</tr>
<tr>
<td>non-critical section</td>
<td>non-critical section</td>
</tr>
<tr>
<td>p1: gate1 ← p</td>
<td>q1: gate1 ← q</td>
</tr>
<tr>
<td>p2: if gate2 ≠ 0 goto p1</td>
<td>q2: if gate2 ≠ 0 goto q1</td>
</tr>
<tr>
<td>p3: gate2 ← p</td>
<td>q3: gate2 ← q</td>
</tr>
<tr>
<td>p4: if gate1 ≠ p</td>
<td>q4: if gate1 ≠ q</td>
</tr>
<tr>
<td></td>
<td>if gate2 ≠ q goto q1</td>
</tr>
<tr>
<td>p5: if gate2 ≠ p goto p1</td>
<td>critical section</td>
</tr>
<tr>
<td>p6: gate2 ← 0</td>
<td>q6: gate2 ← 0</td>
</tr>
</tbody>
</table>
Algorithm 5.4: Fast algorithm for two processes (outline)

<table>
<thead>
<tr>
<th>p</th>
<th>q</th>
</tr>
</thead>
<tbody>
<tr>
<td>loop forever</td>
<td>loop forever</td>
</tr>
<tr>
<td></td>
<td>non-critical section</td>
</tr>
<tr>
<td>p1: gate1 ← p</td>
<td>q1: gate1 ← q</td>
</tr>
<tr>
<td>p2: if gate2 ≠ 0 goto p1</td>
<td>q2: if gate2 ≠ 0 goto q1</td>
</tr>
<tr>
<td>p3: gate2 ← p</td>
<td>q3: gate2 ← q</td>
</tr>
<tr>
<td>p4: if gate1 ≠ p</td>
<td>q4: if gate1 ≠ q</td>
</tr>
<tr>
<td>p5: if gate2 ≠ p goto p1</td>
<td>q5: if gate2 ≠ q goto q1</td>
</tr>
<tr>
<td></td>
<td>critical section</td>
</tr>
<tr>
<td>p6: gate2 ← 0</td>
<td>q6: gate2 ← 0</td>
</tr>
</tbody>
</table>

- No contention for P, if P alone (i.e., gate2 = 0)
- Little overhead in entry
- 2 assignments and 2 comparisons
**Algorithm 5.4: Fast algorithm for two processes (outline)**

<table>
<thead>
<tr>
<th>p</th>
<th>q</th>
</tr>
</thead>
<tbody>
<tr>
<td>loop forever</td>
<td>loop forever</td>
</tr>
<tr>
<td>non-critical section</td>
<td>non-critical section</td>
</tr>
<tr>
<td>p1: gate1 ← p</td>
<td>q1: gate1 ← q</td>
</tr>
<tr>
<td>p2: <strong>if gate2 ≠ 0 goto p1</strong></td>
<td>q2: if gate2 ≠ 0 goto q1</td>
</tr>
<tr>
<td>p3: gate2 ← p</td>
<td>q3: gate2 ← q</td>
</tr>
<tr>
<td>p4: if gate1 ≠ p goto p1</td>
<td>q4: if gate1 ≠ q</td>
</tr>
<tr>
<td>p5:</td>
<td>q5: if gate2 ≠ q goto q1</td>
</tr>
<tr>
<td>critical section</td>
<td>critical section</td>
</tr>
<tr>
<td>p6: gate2 ← 0</td>
<td>q6: gate2 ← 0</td>
</tr>
</tbody>
</table>

- **Q pass gate2 (q3), when P tries to get in**
- P blocks at p2, until Q releases gate2
- Q will advance even if P gets to p1 before q4 executed
Algorithm 5.4: Fast algorithm for two processes (outline)

integer gate1 ← 0, gate2 ← 0

<table>
<thead>
<tr>
<th>p</th>
<th>q</th>
</tr>
</thead>
<tbody>
<tr>
<td>loop forever</td>
<td>loop forever</td>
</tr>
<tr>
<td>non-critical section</td>
<td>non-critical section</td>
</tr>
<tr>
<td>p1: gate1 ← p</td>
<td>q1: gate1 ← q</td>
</tr>
<tr>
<td>p2: if gate2 ≠ 0 goto p1</td>
<td>q2: if gate2 ≠ 0 goto q1</td>
</tr>
<tr>
<td>p3: gate2 ← p</td>
<td>q3: gate2 ← q</td>
</tr>
<tr>
<td>p4: if gate1 ≠ p</td>
<td>q4: if gate1 ≠ q</td>
</tr>
<tr>
<td>p5: if gate2 ≠ p goto p1</td>
<td>q5: if gate2 ≠ q goto q1</td>
</tr>
<tr>
<td>critical section</td>
<td>critical section</td>
</tr>
<tr>
<td>p6: gate2 ← 0</td>
<td>q6: gate2 ← 0</td>
</tr>
</tbody>
</table>

- Q arrives at the same time with P
  - Competition on who wrote to gate1 and gate2 last
  - P & P: P advances, Q blocks at q5
  - P & Q: P advances, Q advances, i.e., no mutex (ouch!)
<table>
<thead>
<tr>
<th></th>
<th>( p )</th>
<th>( q )</th>
</tr>
</thead>
<tbody>
<tr>
<td>p1</td>
<td>( \text{gate}_1 \leftarrow p )</td>
<td>( \text{gate}_1 \leftarrow q )</td>
</tr>
<tr>
<td></td>
<td>( \text{want}_p \leftarrow \text{true} )</td>
<td>( \text{want}_q \leftarrow \text{true} )</td>
</tr>
<tr>
<td>p2</td>
<td>if ( \text{gate}_2 \neq 0 ) ( \text{want}_p \leftarrow \text{false} )</td>
<td>if ( \text{gate}_2 \neq 0 ) ( \text{want}_q \leftarrow \text{false} )</td>
</tr>
<tr>
<td></td>
<td>goto p1</td>
<td>goto q1</td>
</tr>
<tr>
<td>p3</td>
<td>( \text{gate}_2 \leftarrow p )</td>
<td>( \text{gate}_2 \leftarrow q )</td>
</tr>
<tr>
<td>p4</td>
<td>if ( \text{gate}_1 \neq p ) ( \text{want}_p \leftarrow \text{false} )</td>
<td>if ( \text{gate}_1 \neq q ) ( \text{want}_q \leftarrow \text{false} )</td>
</tr>
<tr>
<td></td>
<td>await ( \text{want}_q = \text{false} ) ( \text{critical section} )</td>
<td>await ( \text{want}_p = \text{false} ) ( \text{critical section} )</td>
</tr>
<tr>
<td>p6</td>
<td>( \text{gate}_2 \leftarrow 0 )</td>
<td>( \text{gate}_2 \leftarrow 0 )</td>
</tr>
<tr>
<td></td>
<td>( \text{want}_p \leftarrow \text{false} )</td>
<td>( \text{want}_q \leftarrow \text{false} )</td>
</tr>
</tbody>
</table>

P last at gate1
Q last at gate 2

Q blocks here
Fast N Process Baker

• Expand Alg. 5.6
  – Still with just 2 gates

P: \( \text{await wantq=false} \)  \quad \rightarrow \quad \text{Pi: For all other } j \text{ \ await want}[j]=\text{false} \\

• Still fast, even with “for all other”
  – Fast when no contention (\( \text{gate2} = 0 \))
    • Entry: 3 assignments, 2 if’s
  – Awaits done only when contention
    • \( p4: \text{ if } \text{gate1} \neq i \)