Concurrent Systems – The Big Picture

Concurrency: different processes trying not to run into each others’ way

Main problem of concurrency: sharing computational resources

Shared resource = crossing, bikers = processes, and a (data) race in progress, approaching a disaster.

Solutions to this must be carefully designed and verified, otherwise.
Concurrency: different processes trying not to run into each others' way

Main problem of concurrency: sharing computational resources

http://www.youtube.com/watch?v=JgMB6nEv7K0
http://www.youtube.com/watch?v=G8eqymwUFi8
Concurrent Systems – The Big Picture

Concurrency: different processes trying not to run into each others’ way

Main problem of concurrency: sharing computational resources

http://www.youtube.com/watch?v=JgMB6nEv7K0
http://www.youtube.com/watch?v=G8eqymwUFi8

Shared resource = crossing, bikers = processes,
Concurrent Systems – The Big Picture

Concurrency: different processes trying not to run into each others’ way

Main problem of concurrency: sharing computational resources

http://www.youtube.com/watch?v=JgMB6nEv7K0
http://www.youtube.com/watch?v=G8eqymwUFi8

Shared resource = crossing, bikers = processes, and a (data) race in progress, approaching a disaster.
Concurrent Systems – The Big Picture

Concurrency: different processes trying not to run into each others’ way

Main problem of concurrency: sharing computational resources

http://www.youtube.com/watch?v=JgMB6nEv7K0
http://www.youtube.com/watch?v=G8eqymwUFi8

Shared resource = crossing, bikers = processes, and a (data) race in progress, approaching a disaster.

Solutions to this must be carefully designed and verified, otherwise...
Focus of this Lecture

Aim of $\text{SPIN}$-style model checking methodology:

exhibit flaws in software systems
Focus of this Lecture

Aim of SPIN-style model checking methodology:

exhibit design flaws in concurrent and distributed software systems
Aim of \textit{SPIN}-style model checking methodology:

exhibit design flaws in \textit{concurrent} and \textit{distributed} software systems
Focus of this Lecture

Aim of SPIN-style model checking methodology:

exhibit design flaws in concurrent and distributed software systems

Focus of this lecture:

▶ Modeling and analyzing concurrent systems
Focus of this Lecture

Aim of SPIN-style model checking methodology:

exhibit design flaws in concurrent and distributed software systems

Focus of this lecture:

▶ Modeling and analyzing concurrent systems

Focus of next lecture:

▶ Modeling and analyzing distributed systems
Concurrent/Distributed systems difficult to get right

problems:

- hard to predict, hard to form faithful intuition

- enormous combinatorial explosion of possible behavior

- interleaving prone to unsafe operations

- counter measures prone to deadlocks

- limited control—from within applications—over 'external' factors:
  - scheduling strategies
  - relative speed of components
  - performance of communication mediums
  - reliability of communication mediums
Concurrent/Distributed systems difficult to get right

problems:
- hard to predict, hard to form faithful intuition
- enormous combinatorial explosion of possible behavior
Concurrent/Distributed systems difficult to get right

problems:
- hard to predict, hard to form faithful intuition
- enormous combinatorial explosion of possible behavior
- interleaving prone to unsafe operations
Concurrent/Distributed systems difficult to get right

problems:
- hard to predict, hard to form faithful intuition
- enormous combinatorial explosion of possible behavior
- interleaving prone to unsafe operations
- counter measures prone to deadlocks
Concurrent/Distributed systems difficult to get right

- hard to predict, hard to form faithful intuition
- enormous combinatorial explosion of possible behavior
- interleaving prone to unsafe operations
- counter measures prone to deadlocks
- limited control—from within applications—over ‘external’ factors:
Concurrent/Distributed systems difficult to get right

problems:

- hard to predict, hard to form faithful intuition
- enormous combinatorial explosion of possible behavior
- interleaving prone to unsafe operations
- counter measures prone to deadlocks
- limited control—from within applications—over ‘external’ factors:
  - scheduling strategies
Concurrent/Distributed systems difficult to get right

problems:
  ▶ hard to predict, hard to form faithful intuition
  ▶ enormous combinatorial explosion of possible behavior
  ▶ interleaving prone to unsafe operations
  ▶ counter measures prone to deadlocks
  ▶ limited control—from within applications—over ‘external’ factors:
    ▶ scheduling strategies
    ▶ relative speed of components
Concurrent/Distributed systems difficult to get right

problems:

- hard to predict, hard to form faithful intuition
- enormous combinatorial explosion of possible behavior
- interleaving prone to unsafe operations
- counter measures prone to deadlocks
- limited control—from within applications—over ‘external’ factors:
  - scheduling strategies
  - relative speed of components
  - performance of communication mediums
Concurrent/Distributed systems difficult to get right

- hard to predict, hard to form faithful intuition
- enormous combinatorial explosion of possible behavior
- interleaving prone to unsafe operations
- counter measures prone to deadlocks
- limited control—from within applications—over ‘external’ factors:
  - scheduling strategies
  - relative speed of components
  - performance of communication mediums
  - reliability of communication mediums
We cannot exhaustively test concurrent/distributed systems

- lack of controllability
  ⇒ we miss failures in test phase
Testing Concurrent or Distributed System is Hard

We cannot exhaustively test concurrent/distributed systems

- lack of controllability ⇒ we miss failures in test phase
- lack of reproducability ⇒ even if failures appear in test phase, often impossible to analyze/debug defect
We cannot exhaustively test concurrent/distributed systems

- lack of controllability
  ⇒ we miss failures in test phase

- lack of reproducability
  ⇒ even if failures appear in test phase, often impossible to analyze/debug defect

- lack of time
  exhaustive testing exhausts the testers long before it exhausts behavior of the system...
Mission of Spin-style Model Checking

offer an efficient methodology to

- improve the design
- exhibit defects

of concurrent and distributed systems
Activities in **SPIN**-style Model Checking

1. model (critical aspects of) concurrent/distributed system with **Promela**
2. state crucial properties with assertions, temporal logic, ...
3. use **SPIN** to check all possible runs of the model
4. analyze result, possibly re-work 1. and 2.
Activities in **Spin**-style Model Checking

1. model (critical aspects of) concurrent/distributed system with **Promela**
2. state crucial properties with assertions, temporal logic, . . .
3. use **Spin** to check all possible runs of the model
4. analyze result, possibly re-work 1. and 2.

Separate concerns of model vs. property! Check the property you want the model to have, not the one it happens to have.
Main Challenges of Modeling

**expressiveness**

model must be expressive enough to ‘embrace’ defects the real system could have

**simplicity**

model must be simple enough to be ‘model checkable’, theoretically and practically
In the SPIN approach, the cornerstone of modeling concurrent/distributed systems are Promela processes.
There is always an initial process prior to all others often declared *implicitly* using ‘active’.

```c
init {
    printf("Hello \n\n")
}
```

If `init` is used to start other processes with `run` statement.
Initializing Processes

There is always an initial process prior to all others often declared *implicitly* using ‘active’.

Can be declared *explicitly* with key word ‘init’

```c
init {
    printf("Hello world\n")
}
```

If *explicit*, `init` is used to start other processes with `run` statement.
Processes can be started *explicitly* using `run`

```plaintext
proctype P() {
    byte local;
    ...
}

init {
    run P();
    run P()
}
```

Each `run` operator starts copy of process (with copy of local variables)
Starting Processes

Processes can be started *explicitly* using `run`

```plaintext
proctype P() {
    byte local;
    ...
}
```

```plaintext
init {
    run P();
    run P()
}
```

Each `run` operator starts copy of process (with copy of local variables)

`run P()` does *not* wait for `P` to finish
Starting Processes

Processes can be started *explicitly* using `run`

```c
proctype P() {
    byte local;
    ...
}
```

```c
init {
    run P();
    run P();
}
```

Each `run` operator starts copy of process (with copy of local variables)

`run P()` does *not* wait for `P` to finish

(Promela’s `run` corresponds to Java’s `start`, *not* to Java’s `run`)
Atomic Start of Multiple Processes

By convention, run operators enclosed in atomic block

\begin{verbatim}
proctype P() {
    byte local;
    ...
}

init {
    atomic {
        run P();
        run P()
    }
}
\end{verbatim}
Atomic Start of Multiple Processes

By convention, run operators enclosed in atomic block

```plaintext
proctype P() {
    byte local;
    ...
}

init {
    atomic {
        run P();
        run P()
    }
}
```

Effect: processes only start executing once all are created
Atomic Start of Multiple Processes

By convention, run operators enclosed in atomic block

```java
proctype P() {
    byte local;
    ...
}
```

```java
init {
    atomic {
        run P();
        run P()
    }
}
```

Effect: processes only start executing once all are created

(More on atomic later)
Joining Processes

Following trick allows ‘joining’: waiting for all processes to finish

byte result;

proctype P() {
    ...
}

init {
    atomic {
        run P();
        run P()
    }
    (_nr_pr == 1); /*blocks until join*/
    printf("result␣=%d", result)
}
Following trick allows ‘joining’: waiting for all processes to finish

```c
byte result;

proctype P() {
    ...
}

init {
    atomic {
        run P();
        run P()
    } (_nr_pr == 1); /*blocks until join*/
    printf("result = \%d", result)
}
```

_nr_pr built-in variable holding number of running processes
_nr_pr == 1 only ‘this’ process (init) is (still) running
Processes may have formal parameters, instantiated by `run`:

```java
proctype P(byte id; byte incr) {
  ...
}

init {
  run P(7, 10);
  run P(8, 15)
}
```
Active (Sets of) Processes

`init` can be made **implicit** by using the `active` modifier:

```plaintext
active proctype P() {
    ...
}
```

Implicit `init` will run one copy of `P`
Active (Sets of) Processes

init can be made implicit by using the active modifier:

```c
active proctype P() {
    ...
}
```

Implicit init will run one copy of P

```c
active [n] proctype P() {
    ...
}
```

Implicit init will run n copies of P
Local and Global Data

Variables declared outside of the processes are **global** to all processes.

Variables declared inside a process are **local** to that processes.

```plaintext
byte n;

proc type P(byte id; byte incr) {
    byte t;
    ...
}
```

n is **global**

t is **local**
Pragmatics of modeling with global data:

**Shared memory** of concurrent systems often modeled by global variables of numeric (or array) type.

**Status of shared resources** (printer, traffic light, ...) often modeled by global variables of Boolean or enumeration type (bool/mtype).

**Communication mediums** of distributed systems often modeled by global variables of channel type (*chan*). (next lecture)
byte n = 0;

active proctype P() {
    n = 1;
    printf("Process P, n = %d\n", n)
}

active proctype Q() {
    n = 2;
    printf("Process Q, n = %d\n", n)
}

How many outputs possible?
Interference on Global Data

byte n = 0;

active proctype P() {
    n = 1;
    printf("Process P, n = %d\n", n)
}

active proctype Q() {
    n = 2;
    printf("Process Q, n = %d\n", n)
}
byte n = 0;

active proctype P() {
    n = 1;
    printf("Process P, n = %d\n", n)
}

active proctype Q() {
    n = 2;
    printf("Process Q, n = %d\n", n)
}

How many outputs possible?
Interference on Global Data

```c
byte n = 0;

active proctype P() {
  n = 1;
  printf("Process P, n = %d\n", n)
}

active proctype Q() {
  n = 2;
  printf("Process Q, n = %d\n", n)
}

How many outputs possible?
```

Different processes can interfere on global data
Examples

1. interleave0.pml
   SPIN simulation, SPINSPIDER automata + transition system

2. interleave1.pml
   SPIN simulation, adding assertion, fine-grained execution model, model checking

3. interleave5.pml
   SPIN simulation, SPIN model checking, trail inspection
Promela has no synchronization primitives, like semaphores, locks, or monitors.
Promela has no synchronization primitives, like semaphores, locks, or monitors.

Instead, Promela inhibits concept of statement executability.
Promela has *no synchronization primitives*, like semaphores, locks, or monitors.

Instead, Promela inhibits concept of statement **executability**.

Executability addresses many issues in the interplay of processes.
**Synchronization on Global Data**

Promela has *no synchronization primitives*, like semaphores, locks, or monitors.

Instead, Promela inhibits concept of statement executability.

Executability addresses many issues in the interplay of processes.

Most known synchronization primitives (e.g. test & set, compare & swap, semaphores) can be modeled using executability and atomicity.
executeability

Each statement has the notion of executability. However, only statements are considered executable.

<table>
<thead>
<tr>
<th>statement type</th>
<th>executable</th>
</tr>
</thead>
<tbody>
<tr>
<td>assignment</td>
<td>always</td>
</tr>
<tr>
<td>assertion</td>
<td>always</td>
</tr>
<tr>
<td>print statement</td>
<td>always</td>
</tr>
<tr>
<td>expression statement</td>
<td>if value not 0, false</td>
</tr>
<tr>
<td>send/receive statement</td>
<td>(next lecture)</td>
</tr>
</tbody>
</table>
Each statement has the notion of executability.

<table>
<thead>
<tr>
<th>statement type</th>
<th>executable</th>
</tr>
</thead>
<tbody>
<tr>
<td>assignment</td>
<td>always</td>
</tr>
<tr>
<td>assertion</td>
<td>always</td>
</tr>
<tr>
<td>print statement</td>
<td>always</td>
</tr>
<tr>
<td>expression statement</td>
<td>iff value not 0 = false</td>
</tr>
<tr>
<td>send/receive statement</td>
<td>(next lecture)</td>
</tr>
</tbody>
</table>

**Definition (Expression Statement)**

An **expression statement** is a statement only consisting of an expression.
Executability

Each statement has the notion of executability.

Executability of basic statements:

<table>
<thead>
<tr>
<th>statement type</th>
<th>executable</th>
</tr>
</thead>
<tbody>
<tr>
<td>assignment</td>
<td>always</td>
</tr>
<tr>
<td>assertion</td>
<td>always</td>
</tr>
<tr>
<td>print statement</td>
<td>always</td>
</tr>
<tr>
<td>expression statement</td>
<td>iff value not 0/true</td>
</tr>
<tr>
<td>send/receive statement</td>
<td>(next lecture)</td>
</tr>
</tbody>
</table>

**Definition (Expression Statement)**

An expression statement is a statement only consisting of an expression.
Each statement has the notion of executability.

<table>
<thead>
<tr>
<th>statement type</th>
<th>executable</th>
</tr>
</thead>
<tbody>
<tr>
<td>assignment</td>
<td>always</td>
</tr>
<tr>
<td>assertion</td>
<td>always</td>
</tr>
<tr>
<td>print statement</td>
<td>always</td>
</tr>
<tr>
<td>expression statement</td>
<td>iff value not 0/false</td>
</tr>
<tr>
<td>send/receive statement</td>
<td>(next lecture)</td>
</tr>
</tbody>
</table>

Definition (Expression Statement)
An expression statement is a statement only consisting of an expression.
Executability (Cont’d)

Executability of **compound statements**:

- If a compound statement is executable if and only if at least one of its alternatives is executable.
- An alternative is executable if and only if its guard (the first statement) is executable.

(Recall: in alternatives, "->" is syntactic sugar for ";")

[1] alternative = list of statements
Executability (Cont’d)

Executability of **compound statements**:

if resp. do statement is executable
iff
any of its alternatives\(^1\) is executable

\(^1\)alternative = list of statements
Executability (Cont’d)

Executability of compound statements:

if resp. do statement is executable
iff
any of its alternatives\(^1\) is executable

An alternative is executable
iff
its guard (the first statement) is executable

\(^1\)alternative = list of statements
Executability of compound statements:

if resp. do statement is executable
iff
any of its alternatives\(^1\) is executable

An alternative is executable
iff
its guard (the first statement) is executable
(Recall: in alternatives, “\(\rightarrow\)” syntactic sugar for “;”)

\(^1\)alternative = list of statements
Executability of **compound statements:**

\[
\text{if resp. do statement is executable iff any of its alternatives}^1 \text{ is executable}
\]

An alternative is executable iff its guard (the first statement) is executable

(Recall: in alternatives, “\(\rightarrow\)” syntactic sugar for “;”)

Revisit end.pml

---

1alternative = list of statements
Executability and Blocking

**Definition (Blocking)**

A statement blocks iff it is *not* executable.

A process blocks iff its location counter points to a blocking statement.

For each step of execution, the scheduler nondeterministically chooses a process to execute.
Executability and Blocking

Definition (Blocking)

A statement blocks iff it is not executable.
A process blocks iff its location counter points to a blocking statement.

For each step of execution, the scheduler nondeterministically chooses a process to execute among the non-blocking processes.
Executability and Blocking

**Definition (Blocking)**

A statement blocks iff it is *not* executable.

A process blocks iff its location counter points to a blocking statement.

For each step of execution, the scheduler nondeterministically chooses a process to execute among the non-blocking processes.

Executability, resp. blocking are the key to PROMELA-style modeling of solutions to synchronization problems.
Atomicity

limit the possibility of sequences being interrupted by other processes

**weakly atomic sequence**

- can *only* be interrupted if a statement blocks

**strongly atomic sequence**

- cannot be interrupted at all
Atomicity

limit the possibility of sequences being interrupted by other processes

**weakly atomic sequence**

- can *only* be interrupted if a statement blocks
- defined in Promela by `atomic{list_of_statements}`

**strongly atomic sequence**

- cannot be interrupted at all
- defined in Promela by `d_step{list_of_statements}`
atomic resp. d-step statement is executable
iff
  guard (i.e., the first inner statement) is executable
Deterministic Sequences

d_{step}:
- strongly atomic
- deterministic (like a single step)
Deterministic Sequences

d_step:

- strongly atomic
- deterministic (like a single step)

choices resolved in fixed way (always take the first possible option)
⇒ avoid choices in d_step
Deterministic Sequences

\( d_{\text{step}}: \)
- strongly atomic
- deterministic (like a single step)
- choices resolved in fixed way (always take the first possible option)  
  \( \Rightarrow \) avoid choices in \( d_{\text{step}} \)
- it is an error if any statement within \( d_{\text{step}} \), other than the first one (called ‘guard’), blocks

\[
\begin{align*}
\text{d\_step} & \{ \\
\text{stmt1; } & \leftarrow \text{guard} \\
\text{stmt2; } & \\
\text{stmt3} & \\
\} \\
\end{align*}
\]

If \( \text{stmt1} \) blocks, \( d_{\text{step}} \) is **not entered**, and blocks as a whole.
It is an error if \( \text{stmt2} \) or \( \text{stmt3} \) block.
(Weakly) Atomic Sequences

atomic:
  ▶ weakly atomic
  ▶ can be non-deterministic

atomic {
  stmt1; ← guard
  stmt2;
  stmt3
}

If guard blocks, atomic is not entered, and blocks as a whole.

Once atomic is entered, control is kept until a statement blocks, and only in this case passed to another process.
Prohibit Interference by Atomicity

apply atomic or d_step to interference examples
The Critical Section Problem

Archetypal problem of concurrent systems

Given a number of looping processes, each containing a critical section, design an algorithm such that:

- Mutual Exclusion: At most one process is executing its critical section at any time.
- Absence of Deadlock: If some processes are trying to enter their critical sections, then one of them must eventually succeed.
- Absence of (individual) Starvation: If any process tries to enter its critical section, then that process must eventually succeed.
The Critical Section Problem

Archetypal problem of concurrent systems

Given a number of looping processes, each containing a critical section, design an algorithm such that:

**Mutual Exclusion** At most one process is executing its critical section at any time.
The Critical Section Problem

Archetypal problem of concurrent systems

Given a number of looping processes, each containing a critical section, design an algorithm such that:

**Mutual Exclusion** At most one process is executing its critical section at any time.

**Absence of Deadlock** If some processes are trying to enter their critical sections, then one of them must eventually succeed.
The Critical Section Problem

Archetypal problem of concurrent systems

Given a number of looping processes, each containing a critical section, design an algorithm such that:

**Mutual Exclusion** At most one process is executing its critical section at any time.

**Absence of Deadlock** If some processes are trying to enter their critical sections, then one of them must eventually succeed.

**Absence of (individual) Starvation** If any process tries to enter its critical section, then that process must eventually succeed.
Critical Section Pattern

For demonstration and simplicity:
Noncritical and critical sections only `printf` statements

```
active proctype P() {
    do :: printf("P non-critical actions\n");
    /* begin critical section */
    printf("P uses shared resources\n")
    /* end critical section */

    od
}

active proctype Q() {
    do :: printf("Q non-critical actions\n");
    /* begin critical section */
    printf("Q uses shared resources\n")
    /* end critical section */

    od
}
```
No Mutual Exclusion Yet

More infrastructure to achieve ME.

Adding two Boolean flags:

```c
bool P_in_CS = false;
bool Q_in_CS = false;
```

```c
active proctype P() {
    do :
        printf("P\nnon-critical actions\n");
        P_in_CS = true;
    /* begin critical section */
    printf("P\nuses \shared \resources\n");
    /* end critical section */
    P_in_CS = false
    od
}
```

```c
active proctype Q() {
    ...correspondingly...
}
```
Show Mutual Exclusion VIOLATION with \textbf{SPIN}

adding assertions

\begin{verbatim}
bool P_in_CS = false;
bool Q_in_CS = false;
\end{verbatim}

\begin{verbatim}
active proctype P() {
  do ::
    printf("P non-critical actions\n");
    P_in_CS = true;
    /* begin critical section */
    printf("P uses shared resources\n");
    assert(!Q_in_CS);
    /* end critical section */
    P_in_CS = false
  od
}
\end{verbatim}

\begin{verbatim}
active proctype Q() {
  .......assert(!P_in_CS); .......
}
\end{verbatim}
bool P_in_CS = false;
bool Q_in_CS = false;

active proctype P() {
    do :: printf("P\nnon-critical\nactions\n");
    P_in_CS = true;
    do :: !Q_in_CS -> break
        :: else -> skip
    od;
    /* begin critical section */
    printf("P\nuses\nshared\nresources\n");
    assert(!Q_in_CS);
    /* end critical section */
    P_in_CS = false
}

active proctype Q() { ...correspondingly... }
Mutual Exclusion by Blocking

Instead of Busy Waiting, process should

1. yield control,

2. continue to run only when exclusion properties becomes true again.
Mutual Exclusion by Blocking

Instead of Busy Waiting, process should

1. yield control,
2. continue to run only when exclusion properties becomes true again.

We can use expression statement $!Q_{\text{in\_CS}}$, to let process $P$ block where it should not proceed!
active proctype P() {
    do :: printf("P\non-critical\nactions\n");
    P_in_CS = true;
    !Q_in_CS;
    /* begin critical section */
    printf("P\nuses\nshared\nresources\n");
    assert(!Q_in_CS);
    /* end critical section */
    P_in_CS = false
    od
}

active proctype Q() {
    ...correspondingly...
}
Verify Mutual Exclusion of this

Verify with \texttt{SPIN}
Verify Mutual Exclusion of this

Verify with \texttt{SPIN}

\texttt{SPIN} error (invalid end state) \implies \text{deadlock}
Verify Mutual Exclusion of this

Verify with SPIN

SPIN error (invalid end state) ⇒ deadlock

can make pan ignore the deadlock: ./pan -E
Verify Mutual Exclusion of this

Verify with **SPIN**

**SPIN** error *(invalid end state)* → deadlock

can make `pan` ignore the deadlock: `./pan -E`

**SPIN** still reports assertion violation(!)
In this example:

- mutual exclusion (ME) cannot be shown by Spin
In this example:

- mutual exclusion (ME) cannot be shown by Spin
- P/Q_in_CS sufficient for achieving ME
In this example:

- mutual exclusion (ME) cannot be shown by $\text{SPIN}$
- $P/Q\_in\_CS$ sufficient for *achieving* ME
- $P/Q\_in\_CS$ *not* sufficient for *proving* ME
Proving Mutual Exclusion

In this example:

- mutual exclusion (ME) cannot be shown by Spin
- $P/Q_{in\_CS}$ sufficient for *achieving* ME
- $P/Q_{in\_CS}$ *not* sufficient for *proving* ME

Need more infrastructure.

**Ghost variables**: variables for verification, not for modeling
int critical = 0;

active proctype P() {
    do ::
        printf("P\nnon-critical\nactions\n");
        P_in_CS = true;
        !Q_in_CS;
/* begin critical section */
critical++;
        printf("P\nuses\nshared\nresources\n");
        assert (critical < 2);
        critical--;
/* end critical section */
        P_in_CS = false
    od
}

active proctype Q() {
    ...correspondingly...
}
Verify Mutual Exclusion of this

\( \text{Spin} \ (./\text{pan} \ -E) \) shows no assertion is violated

\[ \Rightarrow \text{mutual exclusion is verified} \]
Verify Mutual Exclusion of this

\texttt{SPIN (./pan  -E)} shows no assertion is violated
\Rightarrow mutual exclusion is verified

Still \texttt{SPIN (without -E) reports (invalid end state)}
\Rightarrow deadlock
Deadlock Hunting

Invalid End State:

- A process does not finish at its end
- OK if it is not crucial to continue – see end.pml
- If it is crucial to continue:

  Real deadlock
Invalid End State:

- A process does not finish at its end
- OK if it is not crucial to continue – see end.pml
- If it is crucial to continue:
  Real deadlock

Find Deadlock with Spin:

- Verify to produce a failing run trail
- Simulate to see how the processes get to the interlock
- Fix the code (not using the end labels nor –E option)
Atomicity against Deadlocks

solution:

checking and setting the flag in one atomic step
solution:

checking and setting the flag in one atomic step

(demonstrate that in csGhost.pml)
Atomicity against Deadlocks

solution:

(checking and setting the flag in one atomic step)

( demonstrate that in csGhost.pml)

```plaintext
atomic {
  !Q_in_CS;
  P_in_CS = true
}
```
Variations of Critical Section Problem

- Verification artifacts:
  - ghost variables ('verification only' variables)
  - temporal logic (later in the course)
  - Max $n$ processes allowed in critical section
    - modeling possibilities include:
      - counters instead of booleans
      - semaphores (see demo)
    - More fine grained exclusion conditions, e.g.
      - several critical sections (Leidestraat in Amsterdam)
      - writers exclude each other and readers
      - readers exclude writers, but not other readers
    - FIFO queue semaphores, for fairly choosing processes to enter
Variations of Critical Section Problem

- Verification artifacts:
  - ghost variables (‘verification only’ variables)

- Max $n$ processes allowed in critical section

Modeling possibilities include:

- Counters instead of booleans
- Semaphores (see demo)

- More fine-grained exclusion conditions, e.g.
  - Several critical sections (Leidestraat in Amsterdam)
  - Writers exclude each other and readers
  - Readers exclude writers, but not other readers

- FIFO queue semaphores, for fairly choosing processes to enter

... and many more
Variations of Critical Section Problem

▶ Verification artifacts:
  ▶ ghost variables (‘verification only’ variables)
  ▶ temporal logic (later in the course)

Max $n$ processes allowed in critical section modeling possibilities include:

▶ counters instead of booleans
▶ semaphores (see demo)

▶ More fine grained exclusion conditions, e.g.
  ▶ several critical sections (Leidestraat in Amsterdam)
  ▶ writers exclude each other and readers
  ▶ readers exclude writers, but not other readers
  ▶ FIFO queue semaphores, for fairly choosing processes to enter

... and many more
Variations of Critical Section Problem

- Verification artifacts:
  - ghost variables (‘verification only’ variables)
  - temporal logic (later in the course)
- Max $n$ processes allowed in critical section modeling possibilities include:
Variations of Critical Section Problem

- Verification artifacts:
  - ghost variables (‘verification only’ variables)
  - temporal logic (later in the course)
- Max $n$ processes allowed in critical section modeling possibilities include:
  - counters instead of booleans
Variations of Critical Section Problem

- Verification artifacts:
  - ghost variables (‘verification only’ variables)
  - temporal logic (later in the course)

- Max $n$ processes allowed in critical section
  modeling possibilities include:
  - counters instead of booleans
  - semaphores (see demo)
Variations of Critical Section Problem

- Verification artifacts:
  - ghost variables (‘verification only’ variables)
  - temporal logic (later in the course)
- Max $n$ processes allowed in critical section
  modeling possibilities include:
  - counters instead of booleans
  - semaphores (see demo)
- More fine grained exclusion conditions, e.g.
Variations of Critical Section Problem

- Verification artifacts:
  - ghost variables (‘verification only’ variables)
  - temporal logic (later in the course)

- Max $n$ processes allowed in critical section modeling possibilities include:
  - counters instead of booleans
  - semaphores (see demo)

- More fine grained exclusion conditions, e.g.
  - several critical sections (Leidestraat in Amsterdam)
Variations of Critical Section Problem

- Verification artifacts:
  - ghost variables (‘verification only’ variables)
  - temporal logic (later in the course)
- Max $n$ processes allowed in critical section modeling possibilities include:
  - counters instead of booleans
  - semaphores (see demo)
- More fine grained exclusion conditions, e.g.
  - several critical sections (Leidestraat in Amsterdam)
  - writers exclude each other and readers
    readers exclude writers, but not other readers
Variations of Critical Section Problem

- Verification artifacts:
  - ghost variables (‘verification only’ variables)
  - temporal logic (later in the course)

- Max $n$ processes allowed in critical section
  modeling possibilities include:
  - counters instead of booleans
  - semaphores (see demo)

- More fine grained exclusion conditions, e.g.
  - several critical sections (Leidestraat in Amsterdam)
  - writers exclude each other and readers
    readers exclude writers, but not other readers
  - FIFO queue semaphores, for fairly choosing processes to enter
Variations of Critical Section Problem

- Verification artifacts:
  - ghost variables (‘verification only’ variables)
  - temporal logic (later in the course)
- Max $n$ processes allowed in critical section modeling possibilities include:
  - counters instead of booleans
  - semaphores (see demo)
- More fine grained exclusion conditions, e.g.
  - several critical sections (Leidestraat in Amsterdam)
  - writers exclude each other and readers
    readers exclude writers, but not other readers
  - FIFO queue semaphores, for fairly choosing processes to enter
- ... and many more
Why Not Critical Section in Single Atomic Block?

▶ Does not carry over to variations (see previous slide).
▶ atomic only weakly atomic!
▶ step excludes any nondeterminism!
▶ Most important: this misses the point.
   We verify effectiveness of atomic, not of the modeled protection solution!
   Using atomic and step too heavily, for too large blocks, can result in well-behaved models, while modeling the wrong system.
Why Not Critical Section in Single Atomic Block?

- Does not carry over to variations (see previous slide).
- atomic only weakly atomic!
- d_step excludes any nondeterminism!
- Does not carry over to variations (see previous slide).
- atomic only weakly atomic!
- \texttt{d\_step} excludes any nondeterminism!
- Most important: this misses the point. We verify effectiveness of atomic, not of the modeled protection solution!
Why Not Critical Section in Single Atomic Block?

- Does not carry over to variations (see previous slide).
- \texttt{atomic} only weakly atomic!
- \texttt{d\_step} excludes any nondeterminism!
- Most important: this misses the point. We verify effectiveness of \texttt{atomic}, not of the modeled protection solution!

Using \texttt{atomic} and \texttt{d\_step} too heavily, for too large blocks, can result in well-behaved models, while modeling the wrong system.