SWEN-220
Mathematical Models of Software

Concurrency in SPIN
Interleaving
Topics

• Interleaving
• Process Interference
• Race Conditions
• Atomicity
Concurrency

• Concurrency can come in many flavors
  – **Concurrent Programming:**
    • Programming for multi-process / multi-core systems
    • Programming for multi-threaded applications
  – **Distributed programming:** programming for multi-processor systems
  – Execution of statements is non-deterministic

• **Interleaving:** Arbitrary selection of statements for execution from the possible computations of individual processes (threads of execution).
Interleaving

byte n=0;     // global – shared mutable resource (SMR)

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

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

• What output do you expect?
Interleaving

byte n=0;  // Global – Shared Mutable Resource (SMR)

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

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

<table>
<thead>
<tr>
<th>Execution Order</th>
<th>Run 1</th>
<th>Run 2</th>
<th>Run 3</th>
<th>Run 4</th>
<th>Run 5</th>
<th>Run 6</th>
</tr>
</thead>
<tbody>
<tr>
<td>time 1</td>
<td>P()</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>n = 1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>time 2</td>
<td>P()</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>“In P, n = 1”</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>time 3</td>
<td>Q()</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>n = 2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>time 4</td>
<td>Q()</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>“In Q, n = 2”</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
## Interleaving

byte n=0;  // Global – Shared Mutable Resource (SMR)

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

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

table

<table>
<thead>
<tr>
<th>Execution Order</th>
<th>Run 1</th>
<th>Run 2</th>
<th>Run 3</th>
<th>Run 4</th>
<th>Run 5</th>
<th>Run 6</th>
</tr>
</thead>
<tbody>
<tr>
<td>time 1</td>
<td>n = 1</td>
<td>n=1</td>
<td>n=1</td>
<td>n=2</td>
<td>n=2</td>
<td>n=2</td>
</tr>
<tr>
<td>time 2</td>
<td>“In P, n = 1”</td>
<td>n=2</td>
<td>n=2</td>
<td>“In Q, n=2”</td>
<td>n=1</td>
<td>n=1</td>
</tr>
<tr>
<td>time 3</td>
<td>n = 2</td>
<td>“In P, n=2”</td>
<td>“In Q, n=2”</td>
<td>n=1</td>
<td>“In Q, n=1”</td>
<td>“In P, n=1”</td>
</tr>
</tbody>
</table>
Interleaving

One "atomic" Instruction

Processes

P()

Q()

One Execution Cycle

Time
Process Interference

• Interleaving can cause interference between processes, leading to bizarre errors

• Interference is more likely
  – when more than one process modifies the same variable (shared mutable resource – SMR)
  – The variable is operated on by more than one statement within each process
Race Conditions

• Processes running concurrently may access *shared variables* (Shared Mutable Resources – SMR) variables in a “haphazard” way based on *interleaving*, leaving the application in an unexpected (often unsafe) state.

• The timing of execution may vary from one run to the next, resulting in a process “*racing*” ahead of the execution of other processes. (see: Heisenbug)

“If the odds are a million to one against something occurring, chances are fifty-fifty that it will”

Two Finite State Machines (FSM) P, R

P & R processes both run “forever” transitioning from one state to another: (i.e. changing the value of i) .

When run independently, neither process can transition to an “unsafe” state:
P : i=0, i=1, i=2, i=3
R : i=0, i=2, i=4

However, when run concurrently, both processes are competing for the variable i (Shared Mutable Resource).
Note the execution of these guards in processes R & P occurred in an "orderly fashion".

Race condition - evaluation of guard conditions & execution of statements happen in separate execution steps.
Atomicity

• Promela statements are atomic

• Statements are executed in their entirety
  – e.g., n = n + 1

• Note: it is possible for interleaving to occur between the evaluation in the guard and the execution of the statement after the guard
Atomicity with a Guard

int a=1, b=4, c=2;

active proctype P()
{
  if :: atomic{
    a != 0 -> c = b / a  // insures the guard & statement are executed consecutively
  } :: else -> c=b
  fi;
  printf("a=%d\n", a)
}

active proctype Q()
{
  a=0;
}
Using atomic

- Run statements should be enclosed in an atomic sequence
- atomic makes sure that processes are instantiated before any of them begins execution