Concurrent Programs

- reasoning about their execution
- proving correctness
- start by considering *execution sequences*
Execution Sequences

- consider the following instruction sequences executed by threads T0 and T1

<table>
<thead>
<tr>
<th>T0</th>
<th>T1</th>
</tr>
</thead>
<tbody>
<tr>
<td>n = n + 1</td>
<td>n = n + 1</td>
</tr>
<tr>
<td>n = n + 1</td>
<td>n = n + 1</td>
</tr>
<tr>
<td>n = n + 1</td>
<td>n = n + 1</td>
</tr>
<tr>
<td>n = n + 1</td>
<td>n = n + 1</td>
</tr>
</tbody>
</table>

- n is a shared global variable with initial value 0

- assume that each statement \([n = n + 1]\) is executed atomically

- n is effectively incremented by one thread at a time

- statement execution can be interleaved 20 different ways

- as each statement is atomic, n will always end up with the value 6, irrespective of how the execution of the statements are interleaved
Execution Sequences...

- one possible interleave

<table>
<thead>
<tr>
<th></th>
<th>T0</th>
<th>T1</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>n = n + 1</td>
<td>n = 1</td>
</tr>
<tr>
<td>n = n + 1</td>
<td></td>
<td>n = 2</td>
</tr>
<tr>
<td>n = n + 1</td>
<td></td>
<td>n = 3</td>
</tr>
<tr>
<td>n = n + 1</td>
<td></td>
<td>n = 4</td>
</tr>
<tr>
<td>n = n + 1</td>
<td></td>
<td>n = 5</td>
</tr>
<tr>
<td>n = n + 1</td>
<td></td>
<td>n = 6</td>
</tr>
</tbody>
</table>

- n = n + 1 is not normally executed atomically by a CPU
- CPU will read [load] n from shared memory into a CPU register, increment the register and then write [store] the register back to memory
- non atomic read-modify-write operation
Execution Sequences...

- \( n = n + 1 \) is split into two steps: \( t = n \) and \( n = t + 1 \)
- simulates a non atomic read-modify-write sequence
- each thread now has its own local variable \( t_0 \) and \( t_1 \)

<table>
<thead>
<tr>
<th>T0</th>
<th>T1</th>
</tr>
</thead>
<tbody>
<tr>
<td>( t_0 = n )</td>
<td>( t_1 = n )</td>
</tr>
<tr>
<td>( n = t + 1 )</td>
<td>( n = t_1 + 1 )</td>
</tr>
<tr>
<td>( t_0 = n )</td>
<td>( t_1 = n )</td>
</tr>
<tr>
<td>( n = t_0 + 1 )</td>
<td>( n = t_1 + 1 )</td>
</tr>
<tr>
<td>( t_0 = n )</td>
<td>( t_1 = n )</td>
</tr>
<tr>
<td>( n = t_0 + 1 )</td>
<td>( n = t_1 + 1 )</td>
</tr>
</tbody>
</table>

- statements can be interleaved 924 ways
- what are the resulting minimum and maximum values for \( n \)?
  - max \( n = ?? \) min \( n = ?? \)
  - max \( n = 6 \) min \( n = 2 \)
Execution Sequences...

• an execution sequence resulting in $n = 6$

<table>
<thead>
<tr>
<th>T0</th>
<th>T1</th>
</tr>
</thead>
<tbody>
<tr>
<td>$t_0 = n$</td>
<td>$t_0 = 0$</td>
</tr>
<tr>
<td>$n = t_0 + 1$</td>
<td>$n = 1$</td>
</tr>
<tr>
<td>$t_0 = n$</td>
<td>$t_0 = 1$</td>
</tr>
<tr>
<td>$n = t_0 + 1$</td>
<td>$n = 2$</td>
</tr>
<tr>
<td>$t_0 = n$</td>
<td>$t_0 = 2$</td>
</tr>
<tr>
<td>$n = t_0 + 1$</td>
<td>$n = 3$</td>
</tr>
<tr>
<td></td>
<td>$t_1 = n$</td>
</tr>
<tr>
<td></td>
<td>$n = t_1 + 1$</td>
</tr>
<tr>
<td></td>
<td>$t_1 = n$</td>
</tr>
<tr>
<td></td>
<td>$n = t_1 + 1$</td>
</tr>
<tr>
<td></td>
<td>$t_1 = n$</td>
</tr>
<tr>
<td></td>
<td>$n = t_1 + 1$</td>
</tr>
</tbody>
</table>
Execution Sequences...

- an execution sequence resulting in \( n = 2 \) (how many such sequences exist)?

<table>
<thead>
<tr>
<th>( T0 )</th>
<th>( T1 )</th>
</tr>
</thead>
<tbody>
<tr>
<td>( t0 = n )</td>
<td>( t0 = 0 )</td>
</tr>
<tr>
<td>( t1 = n )</td>
<td>( t1 = 0 )</td>
</tr>
<tr>
<td>( n = t1 + 1 )</td>
<td>( n = 1 )</td>
</tr>
<tr>
<td>( n = t1 + 1 )</td>
<td>( n = 2 )</td>
</tr>
<tr>
<td>( n = t0 + 1 )</td>
<td>( n = 1 )</td>
</tr>
<tr>
<td>( t0 = n )</td>
<td>( t0 = 1 )</td>
</tr>
<tr>
<td>( n = t0 + 1 )</td>
<td>( n = 2 )</td>
</tr>
<tr>
<td>( t0 = n )</td>
<td>( t0 = 2 )</td>
</tr>
<tr>
<td>( n = t0 + 1 )</td>
<td>( n = 3 )</td>
</tr>
<tr>
<td>( n = t1 + 1 )</td>
<td>( n = 2 )</td>
</tr>
</tbody>
</table>
Execution Sequences...

- use Promela/Spin to check results

- 
  \(_{nr\_pr == 1}\) waits until the two instances of p0 are terminated and then checks \assert(n > 2)\n
- in verification mode, Spin will execute all possible interleaves and stop if \assert(n > 2)\ is false [ie. will stop if n = 2, 1, ...]

- this sequence can then be replayed for analysis

- change to \assert(n > 1)\ and use verification mode to confirm that the resulting value of n is always greater than 1

- DEMONSTRATE ispin.tcl [relatively easy to install on Windows and Ubuntu; provides a basic user i/f to spin]

```c
int n = 0;

proctype p0() {
    int t;
    t = n;
    n = t + 1; // n = n + 1
    t = n;
    n = t + 1; // n = n + 1
    t = n;
    n = t + 1; // n = n + 1
}

init {
    run p0();
    run p0();
    (_nr_pr == 1);
    assert(n > 2)
}
```

Promela source code
Execution Sequences...

- modify to use a *for* loop [*constructed from a do statement*] to increment \( n \) from 0 to \( N \)

- each *process* executes the read-modify-sequence \( N \) times

- can confirm \( (n \geq 2) \land (n \leq 2 \times N) \)

- if \( N \) large, verification may not complete [*typically runs out of memory*]

- need to increase memory allocated to Spin or use an alternative mode which uses less memory [*eg. compresses state data*], but is more compute intensive

- can also change number of processes [*add run \( p() \)]

- minimum result for \( n \) is the number of processes

---

```promela
#define N 10

int n = 0;

proctype p() {
    int t;
    int i = 0;
    do
        :: (i >= N) ->
            break;
        :: else ->
            t = n;
            n = t + 1;
            i++
    od
}

init {
    run p();
    run p();
    (_nr_pr == 1);
    assert (n > 1)
}
```

*Promela source code*
### Execution Sequences...

- using statement merging
- Starting p0 with pid 1
  - 1: proc 0 (:init::1) count0.pml:20 (state 1) \[\text{[run p0()]}\]
- Starting p0 with pid 2
  - 2: proc 0 (:init::1) count0.pml:21 (state 2) \[\text{[run p0()]}\]
  - 3: proc 2 (p0:1) count0.pml:11 (state 1) \[t = n\]
  - 4: proc 1 (p0:1) count0.pml:11 (state 1) \[t = n\]
  - 5: proc 2 (p0:1) count0.pml:12 (state 2) \[n = (t+1)\]
  - 6: proc 2 (p0:1) count0.pml:13 (state 3) \[t = n\]
  - 7: proc 2 (p0:1) count0.pml:14 (state 4) \[n = (t+1)\]
  - 8: proc 1 (p0:1) count0.pml:12 (state 2) \[n = (t+1)\]
  - 9: proc 2 (p0:1) count0.pml:15 (state 5) \[t = n\]
  - 10: proc 1 (p0:1) count0.pml:13 (state 3) \[t = n\]
  - 11: proc 1 (p0:1) count0.pml:14 (state 4) \[n = (t+1)\]
  - 12: proc 1 (p0:1) count0.pml:15 (state 5) \[t = n\]
  - 13: proc 1 (p0:1) count0.pml:16 (state 6) \[n = (t+1)\]
  - 14: proc 2 (p0:1) count0.pml:16 (state 6) \[n = (t+1)\]
  - 15: proc 2 terminates
  - 16: proc 1 terminates
  - 17: proc 0 (:init::1) count0.pml:22 (state 3) \[(_\text{nr}_pr==1)\]
- spin: count0.pml:23, Error: assertion violated
- spin: text of failed assertion: assert((n>2))
- \#processes: 1
- 18: proc 0 (:init::1) count0.pml:23 (state 4)
- 3 processes created
- Exit-Status 0

<table>
<thead>
<tr>
<th>T0 (P1)</th>
<th>T1 (P2)</th>
</tr>
</thead>
<tbody>
<tr>
<td>t0 = n</td>
<td>t0 = 0</td>
</tr>
<tr>
<td>t1 = n</td>
<td>t1 = 0</td>
</tr>
<tr>
<td>n = t1 + 1</td>
<td>n = 1</td>
</tr>
<tr>
<td>t1 = n</td>
<td>t1 = 1</td>
</tr>
<tr>
<td>n = t1 + 1</td>
<td>n = 2</td>
</tr>
<tr>
<td>n = t0 + 1</td>
<td>n = 1</td>
</tr>
<tr>
<td>t1 = n</td>
<td>t1 = 1</td>
</tr>
<tr>
<td>t0 = n</td>
<td>t0 = 1</td>
</tr>
<tr>
<td>n = t0 + 1</td>
<td>n = 2</td>
</tr>
<tr>
<td>t0 = n</td>
<td>t0 = 2</td>
</tr>
<tr>
<td>n = t1 + 1</td>
<td>n = 2</td>
</tr>
</tbody>
</table>

First two steps swapped compared with slide 6.
Spin states stored and states matched

- consider a simple example with two processes [process 0 and 1] each with two statements
- number of interleaves 6
- draw a state transition diagram where each state represented by a triple \((PC_0, PC_1, n)\)
- arcs represent executed statements

eg. \((n = n + 1)_p\) statement/step executed by process \(p\)

```c
int n = 0;

active[2] proctype p0()
{
    n = n + 1; // PC = 0
    n = n + 1; // PC = 1
    n = n + 1; // PC = 2 (process ended)
}

start 2 instances of p0
```
Spin states stored and states matched...

- assume a depth first search
- new states coloured green
- matched states, which have already been visited, coloured red
- remaining states coloured grey
- 9 states stored [green], 4 states matched [red]
- **NOT** the same as counts reported by Spin
  - 13, 4 with partial order reduction
  - 13, 6 without partial order reduction
Spin states stored and states matched...

- Spin uses an extra state/step to terminate a process [after last statement has been executed]
- why?
- processes are created in source code order [apart from init, if present, which is always process 0]
- terminated in reverse order [process 1 must be terminated before process 0]
- use T for the PC of instruction used to terminate process
- processes numbered 0 and 1 as per previous example
- modified state transition diagram to match Spin
Spin states stored and states matched

- 13 stored states [in green]
- 6 matched states without partial order reduction [in red]
- 4 matched states with partial order reduction
- Red dotted states can be skipped [partial order reduction]
- \((n = n + 1)_0; T_1\) results in the same state change as \(T_1; (n = n + 1)_0\) as statements/steps independent of each other
Synchronisation

- **spin lock**: ensures that only one thread can access a particular shared data structure at a time [*serialise access*]

- **barrier**: ensures that no thread advances beyond a particular point in a computation until ALL have arrived at the barrier - used typically to separate program phases

- synchronization constructs divided into two classes
  - **blocking**: de-schedule waiting thread and schedule another thread to run
  - **busy-wait**: threads repeatedly test a shared variable to determine when they can proceed

- busy-wait preferred when scheduling overhead exceeds expected wait time
Spin Lock Implementations without Atomic Instructions

- Peterson algorithm for **TWO** threads [also google Dekker’s algorithm]

```c
int flag[2]; // initially 0
int last;

void acquire(int id) { // id is the thread ID [0 or 1]
    int j = 1 - id; // 0 -> 1 and 1 ->0
    flag[id] = 1; // want lock
    last = id; // other thread has priority
    while (flag[j] && last == id); // NB last == id
}

void release(int id) { // release lock
    flag[id] = 0;
}
```

- thread sets its *flag* indicating it wants lock and then sets *last* to indicate other thread can have lock if there is a conflict

- wait while other thread has lock AND other thread has priority

**what happens if the variable last removed?**

**what happens if the statement flag[id] = 1 removed?**

**check using Spin**
Peterson Lock

- Promela code for Peterson lock
- Two active processes
- _pid is the process number [0 or 1 in this case]
- Although processes never end, state will eventually be repeated

```promela
// Peterson lock

bool flag[2]; // 0 initially
byte last; // 0 initially

active[2] proctype P() {
    byte i = _pid; // process #
    byte j = 1 - i; // other pid

    again:
        flag[i] = 1;
        last = i;

        (flag[j] == 0 || last == j); // wait until true

        flag[i] = 0; // release lock

        goto again
}
```

**Promela code**
Peterson Lock...

- desirable properties
  - safety  
    "nothing bad ever happens"
    mutual exclusion not violated
  - deadlock free  
    "in every state of every computation, if processes are trying to enter the critical section one will eventually succeed"
  - liveness  
    "something good eventually happens"
    processes continually enter critical section
  - starvation free  
    "if in every state of every computation, if a process tries to enter its critical section it will eventually succeed"

- will use Spin to test for these properties
Peterson Lock...

- base line
- verify using safety mode
- NO errors found
- max depth reached 13
Peterson Lock...

- max depth = 13
- in this case, Spin finds ALL infinite runs (depth first search)
- think of an infinite run as a sequence of states that eventually repeat
- a possible infinite run with depth 13 [there are many others]
- with this example, Spin will eventually reaches a previously visited state
Peterson Lock...

• safety check for mutual exclusion

• first approach

• declare global variable $ncs$ and add following code to critical section

  ```
  ncs++;
  assert(ncs == 1);
  ncs--;
  ```

• run model and verify assertion NOT violated

• comment out line containing "flag[i] = 1;" to force a mutual exclusion error

• verify assert(ncs == 1) violated

• replay trail to find cause of error
Peterson Lock...

- extra code
- NO errors
Peterson Lock...

- // flag[i] = 1;
- assertion violated
Peterson Lock...

- replay trail to find error
- both processes can pass through statement 20 simultaneously
- critical section entered by process 0 in step 20 and ALSO by process 1 in step 24
- error results from ncs++ in step 20 and 24
Peterson Lock...

• second approach for mutual exclusion checking

• check against an LTL claim [linear temporal logic]

  \[ \text{ltl claim } \{ \text{always (ncs } \leq 1) \} \]

• think of LTL checking as a game played between the model and Spin

• Promela claim \textit{process} generated from the inverse of the LTL claim

• claim \textit{process} executed before each step of model to check claim is true in every model state

• Spin wins if assertion error in claim OR acceptance cycle in claim OR claim \textit{process} terminates [meaning claim is FALSE]

\begin{verbatim}
never claim { // !(always (p))
T0_init:
  do
    :: atomic { !(p) -> assert(p)}
    :: (1) -> goto T0_init
  od;
accept_all:
  skip
}
\end{verbatim}

Promela never claim process (very stylised)
Peterson Lock...

- claim
- NO errors
- NB: select **acceptance cycles** and use **claim** buttons selected
Peterson Lock...

- // flag[i] = 1;
- claim
- claim assertion violated
- replay trail to find error
Peterson Lock...

- steps 34 and 42 show how \textit{ncs} gets a value of 2
Peterson Lock...

- click on a never claim (43) step to view Promela code for claim
Peterson Lock...

- third approach for mutual exclusion checking
- LTL claim using statement labels
- separate into two processes P and Q
- add labels $csp$ and $cpq$ to mark critical section in P and Q respectively

$$\text{ltl claim } \{ \text{always } ! (P@csp \&\& Q@csq) \}$$

- $P@csp$ means statement labeled $csp$ in P
Peterson Lock...

- two separate processes
- labels
- `//flag[i] = 1;`
- claim
- NB: *acceptance cycles* and use *claim* buttons selected
- claim assertion violated
Peterson Lock...

- check for deadlock
- Spin selects an executable statement on each step – if there is no executable statement we have deadlock
- force deadlock by removing variable last
- verify in safety mode
- reports pan:1: invalid end state (at depth 8)
- replay trail to find error
Peterson Lock...

- variable last removed
- safety mode
- invalid end state [at depth 4]
Peterson Lock...

- replay trail to find error
- confirms both \textit{processes} deadlocked at flag[j] == 0 (18)
Peterson Lock...

- check for liveness
- separate processes with csp and csq incremented and decremented in respective critical sections (e.g., csp++; csp--)
- show that csp is true infinitely often

\[ \text{ltl claim } \{ \text{always eventually csp} \} \]

\[ \text{csp = 0} \quad \text{csp = 1} \quad \text{...} \quad \text{csp = 0} \quad \text{...} \]

- claim also true with following infinite run

\[ \text{csp = 0} \quad \text{csp = 0} \quad \text{csp = 1} \quad \text{...} \]

- but this will not happen because step csp++ will not be executed again until after csp—
- if not convinced, could use claim

\[ \text{ltl claim } \{ \text{always (eventually csp } \&\& \text{ eventually !csp) } \} \]
Peterson Lock...

- claim
- fails due to an acceptance cycle being found
Peterson Lock...

- replay trail to find error
- verifier is ONLY selecting statements from process 1
- strong fairness
Peterson Lock...

• need to verify with the “enforce weak fairness constraint”

• a computation is weakly fair if and only if the following conditions holds:

  *if a statement is always executable, then it is eventually executed as part of the computation*

• alternatively could try following claim for liveness

  \[ \text{ltl claim} \{ \text{always eventually csp} \text{ || eventually csq} \} \]

• check for starvation freedom

  \[ \text{ltl claim} \{ \text{always eventually csp} \text{ && eventually csq} \} \]

  [need to select “enforce weak fairness constraint”]
Peterson Lock

- claim
- *enforce weak fairness constraint SELECTED*
- NO errors
Peterson Lock...

- an interesting property of the Peterson lock is that if a *process* sets its flag, it will *eventually* enter critical section

- LTL claim

\[
\text{ltl claim \{ always ((flag[0] -> eventually P@csp) \&\& (flag[1] -> eventually Q@csq)) \}}
\]

- implication

- need to add labels csp and csq to mark start of respective critical sections

- **NO** need to *enforce weak fairness constraint*
Peterson Lock...

- labels
- claim
- NO need to enforce weak fairness constraint
- NO errors
Bakery Lock

• Leslie Lamport CACM Aug 1974 [2013 A. M. Turing Award Winner]

• algorithm works with N threads

• think of a baker’s shop

• customers enter door and obtain a unique ticket number from a ticket dispenser [tickets issued in ascending order]

• customers then served in ticket order

• the problem is how to obtain a unique ticket without using any atomic instructions [straightforward with a modern CPU if the right atomic instruction is available]

• often called a ticket lock

• let’s examine a C/C++ version of the code from the original paper
Bakery Lock

```c
int number[MAXTHREAD];  // thread IDs 0 to MAXTHREAD-1
int choosing[MAXTHREAD];

void acquire(int pid) {  // pid is thread ID
    choosing[pid] = 1;
    int max = 0;
    for (int i = 0; i < MAXTHREAD; i++) {  // find maximum ticket
        if (number[i] > max)
            max = number[i];
    }
    number[pid] = max + 1;  // our ticket number is maximum ticket found + 1
    choosing[pid] = 0;
    for (int j = 0; j < MAXTHREAD; j++) {  // wait until our turn i.e. have lowest ticket
        while (choosing[j]);  // wait while thread j choosing
        while (number[j] && ((number[j] < number[pid]) || ((number[j] == number[pid]) && (j < pid))));
    }
}

void release(int pid) {
    number[pid] = 0;  // release lock
}
```
Bakery Lock

• how does the algorithm work?

• consider 3 threads numbered 0, 1 and 2
• imagine thread2 holds lock and number[] = [0, 0, 2]

• if thread0 and thread1 *concurrently* execute the code to get a ticket what, ticket values can be returned?
• NB: number[] can be changed by other threads while a thread is obtaining its ticket

• 3, 4 or 4, 3 or 3, 3 or 1, 2 or 2, 1 or 1, 1 ??

• since threads can be issued with the same ticket number, threadID is used as a differentiator [thread with lower threadID given priority]

• when thread releases lock it sets number[threadID] = 0

• what is the maximum ticket value? can algorithm handle ticket value wrap around?
• why is the *while* (*choosing[jj]*) loop needed?
• what happens if thread goes to sleep *choosing* or holding lock?
Bakery Lock...

- the necessity for variable `choosing` may not be obvious

- there is no 'lock' around lines 5 to 10 where the maximum ticket is calculated

- suppose choosing was removed and two processes computed the same maximum ticket

- if the higher-priority process was pre-empted before setting its number[i], the low-priority process will see that the other process has a number of zero, and enter the critical section; later, the high-priority process may also enter the critical section resulting in two processes entering the critical section at the same time

- the Bakery Algorithm uses the `choosing` variable to make the assignment on line 10 appear atomic

- process `pid` will never see a number equal to zero for a process `j` that is going to pick the same number as `pid`
Using Spin to check the Bakery Lock algorithm

- following statement forms a key part of the Bakery Lock algorithm
  
  ```plaintext
  while (number[j] && ((number[j] < number[pid]) || ((number[j] == number[pid]) && (j < pid))));
  ```

- at first sight, this statement accesses number[j] three times, number[pid] twice, j four times and pid twice

- must make sure that possible interleaved accesses to these variables by the multiple processes at runtime are correctly modelled [think individual memory accesses]

  _pid, j and number[pid] are essentially local to the process, NO problem
  number[j] can be changed asynchronously by other processes
  assume that compiler would generate code that only makes one access to number[j]
  by transforming statement into

  ```plaintext
  while ((v = number[j]) && ((v < number[pid]) || ((v == number[pid]) && (j < pid))));
  ```
  where v is a local variable

- statement can be used AS IS, but what would happen if number[j] was read 3 times?
Learning Outcomes

• you are now able to:
  
  ▪ show how the different execution interleaves of a concurrent algorithm can lead to different results

  ▪ explain the desirable properties of a concurrent algorithms (safety, deadlock free, liveness and starvation free)

  ▪ write simple Spin programs to test the properties of concurrent algorithms

  ▪ use LTL expressions to test for properties of concurrent algorithms

  ▪ analyse the operation and properties of the Peterson lock

  ▪ analyse the operation and properties of the Bakery lock

  ▪ use Spin to test the properties of the Black-White Bakery Algorithm