Spin, Peterson and Bakery Locks

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

T0	T1
n = n + 1	n = n + 1
n = n + 1	n = n + 1
n = n + 1	n = n + 1

- 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

T0	T1	
	n = n + 1	n = 1
n = n + 1		n = 2
n = n + 1		n = 3
	n = n + 1	n = 4
	n = n + 1	n = 5
n = n + 1		n = 6

- 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 t0 and t1

T0	T1
t0 = n	t1 = n
n = t0 + 1	n = t1 + 1
t0 = n	t1 = n
n = t0 + 1	n = t1 + 1
t0 = n	t1 = n
n = t0 + 1	n = t1 + 1

- 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

T0		T1	
t0 = n	*t0 = 0*		
n = t0 + 1	*n = 1*		
t0 = n	*t0 = 1*		
n = t0 + 1	*n = 2*		
t0 = n	*t0 = 2*		
n = t0 + 1	*n = 3*		
		t1 = n	*t1 = 3*
		n = t1 + 1	*n = 4*
		t1 = n	*t1 = 4*
		n = t1 + 1	*n = 5*
		t1 = n	*t1 = 5*
		n = t1 + 1	*n = 6*

Execution Sequences…

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

T0		T1	
t0 = n	*t0 = 0*		
		t1 = n	*t1 = 0*
		n = t1 + 1	*n = 1*
		t1 = n	*t1 = 1*
		n = t1 + 1	*n = 2*
n = t0 + 1	*n = 1*		
		t1 = n	*t1 = 1*
t0 = n	*t0 = 1*		
n = t0 + 1	*n = 2*		
t0 = n	*t0 = 2*		
n = t0 + 1	*n = 3*		
		n = t1 + 1	*n = 2*

Execution Sequences…

- Check execution use Promela/Spin
- (_nr_pr == 1) waits until the two instances of p0 are *terminated* and then checks assert(n > 2)
- in verification mode, Spin will execute all possible interleaves and stop if assert(n > 2) is false [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]

```
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 >= 2) && (n <= 2*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

```
#define N 10

ìnt 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) [(run p0())]
- Starting p0 with pid 2
- 2: proc 0 (:init::1) count0.pml:21 (state 2) [(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) [((_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

T0 (P1)		T1 (P2)	
		t0 = n	*t0 = 0*
t1 = n	*t1 = 0*		
		n = t1 + 1	*n = 1*
		t1 = n	*t1 = 1*
		n = t1 + 1	*n = 2*
n = t0 + 1	*n = 1*		
		t1 = n	*t1 = 1*
t0 = n	*t0 = 1*		
n = t0 + 1	*n = 2*		
t0 = n	*t0 = 2*		
n = t0 + 1	*n = 3*		
		n = t1 + 1	*n = 2*

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*

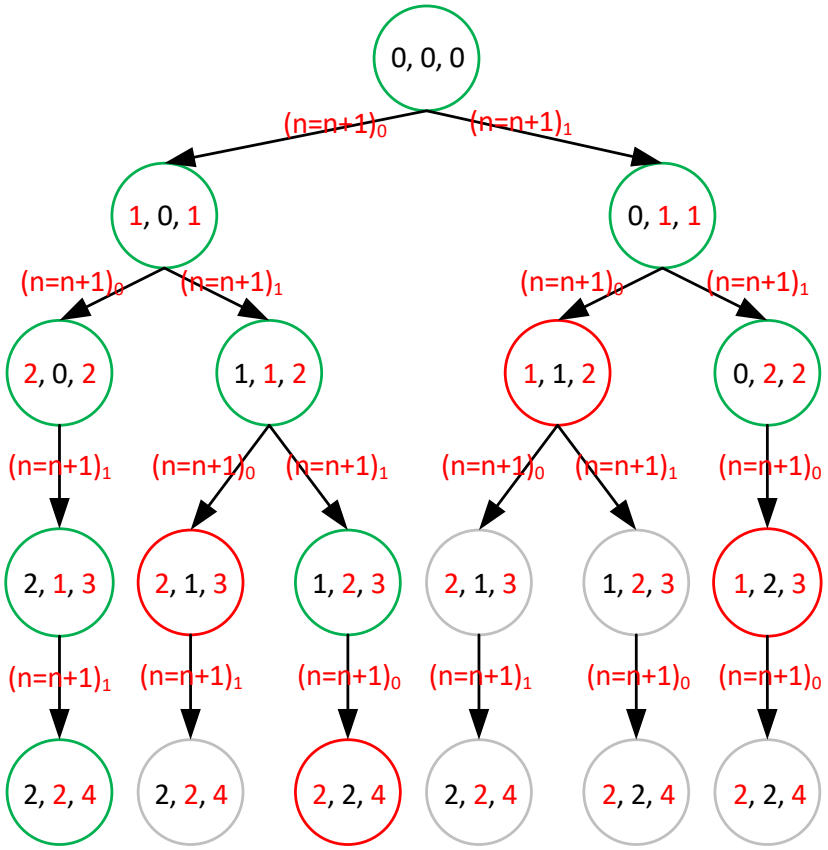
start 2 instances of p0

```
int n = 0;

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

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



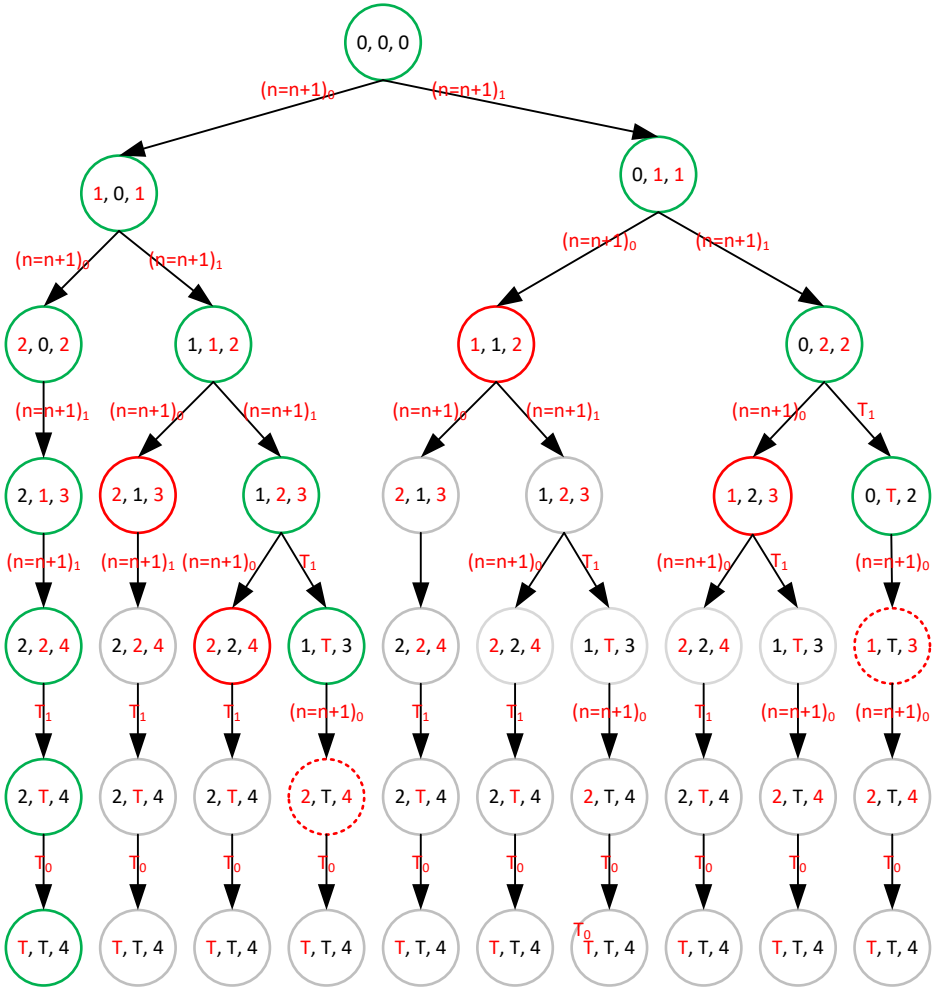
state transition diagram

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

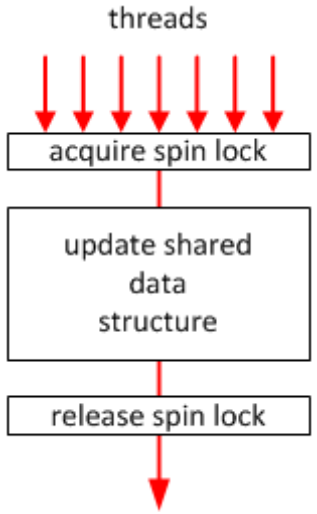
Spin *states stored* and *states matched*

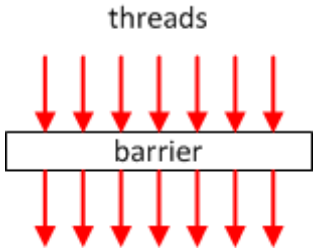
- 13 stored states [in green]
- 6 matched states without partial order reduction [in red]
- 4 matched states with partial order reduction
- red dotted states can be skipped [partial order reduction]
- $(n = n + 1)_0$; T_1 results in the same state change as T_1 ; $(n = n + 1)_0$ as statements/steps independent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

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

what happens if the variable last removed?

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

check using Spin

- if both threads execute "flags[id] = 1; last = id" and then enter while statement *last* will be used to determine which thread gets lock

- is there any reason why this might NOT work?

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
- does code match what the hardware does?

```
//
// 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"*
    eg. thread1 one tries get lock A then B and thread2 B then A
  - liveness/livelock — *"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 show how Spin can be used to test for these properties

Peterson Lock…

- safety check for mutual exclusion
- first approach
- declare global variable *ncs* and add following code to critical section

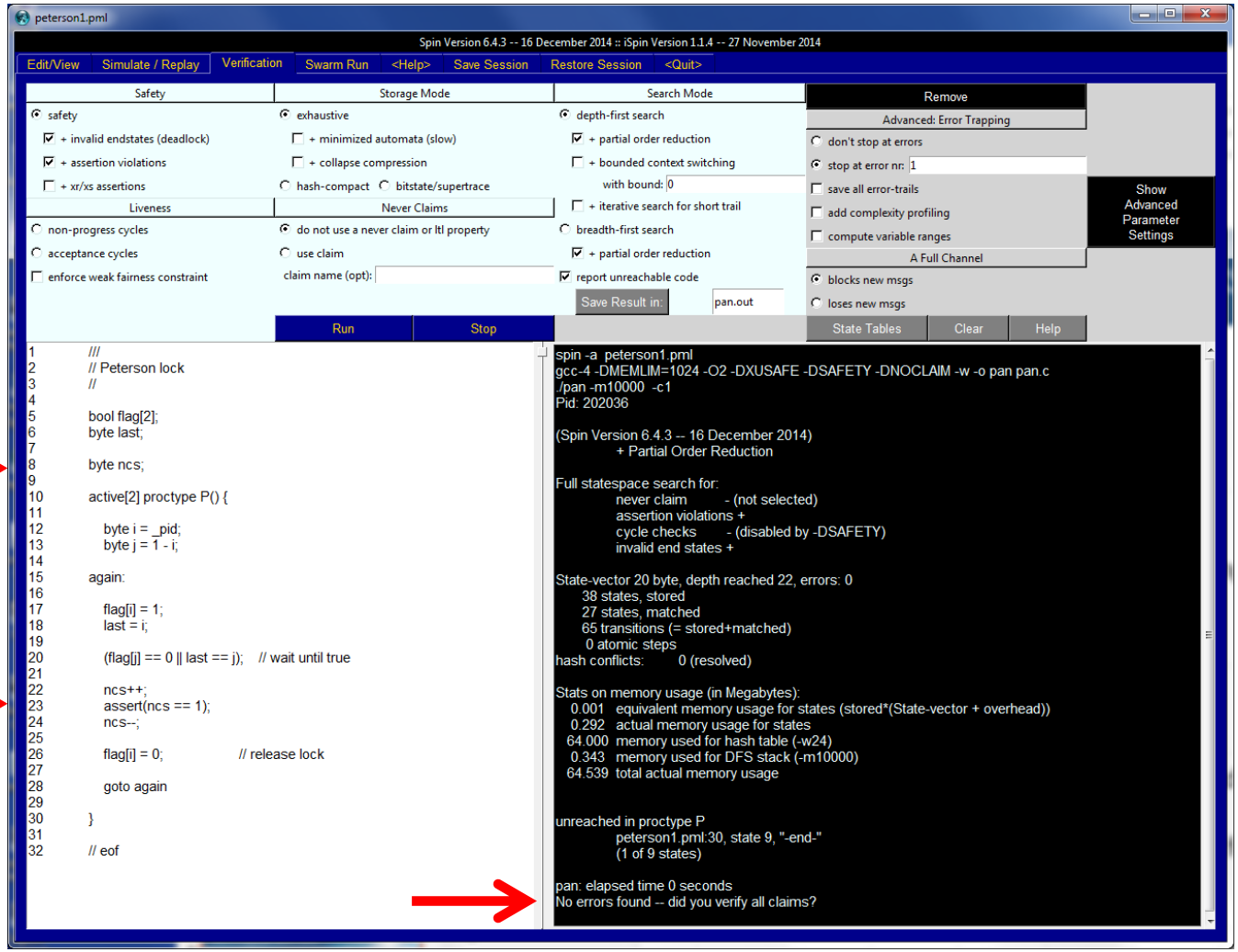
```
ncs++;
assert(ncs == 1);
ncs--;
```

- run model and verify assertion NOT violated
- comment out line containing "flag[i] = 1;" to force a mutual exclusion error
- assert(ncs == 1) violated
- replay trail to find cause of error

SPIN, PETERSON AND BAKERY LOCKS

Peterson Lock…

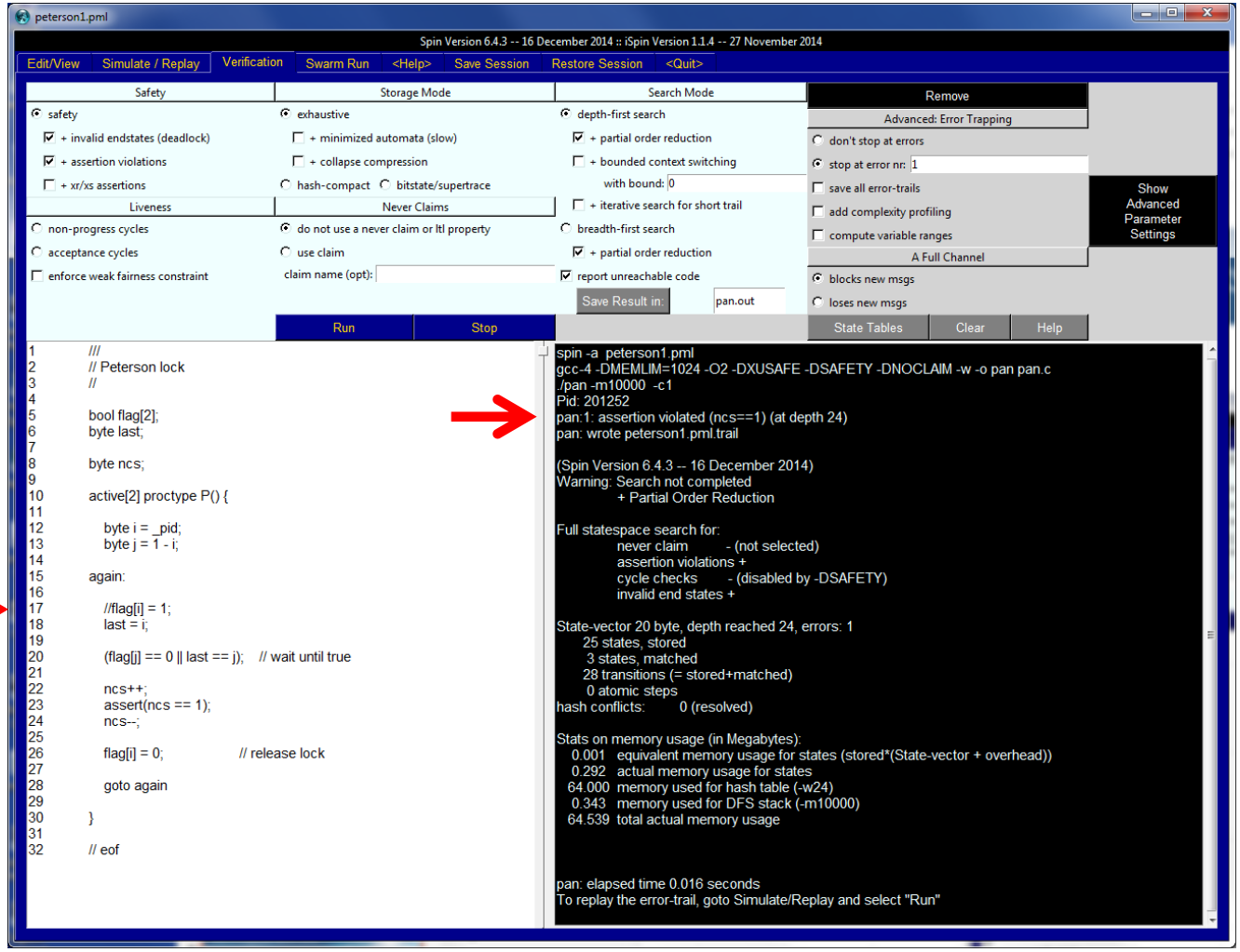
- extra code
- NO errors



Spin, Peterson and Bakery Locks

Peterson Lock…

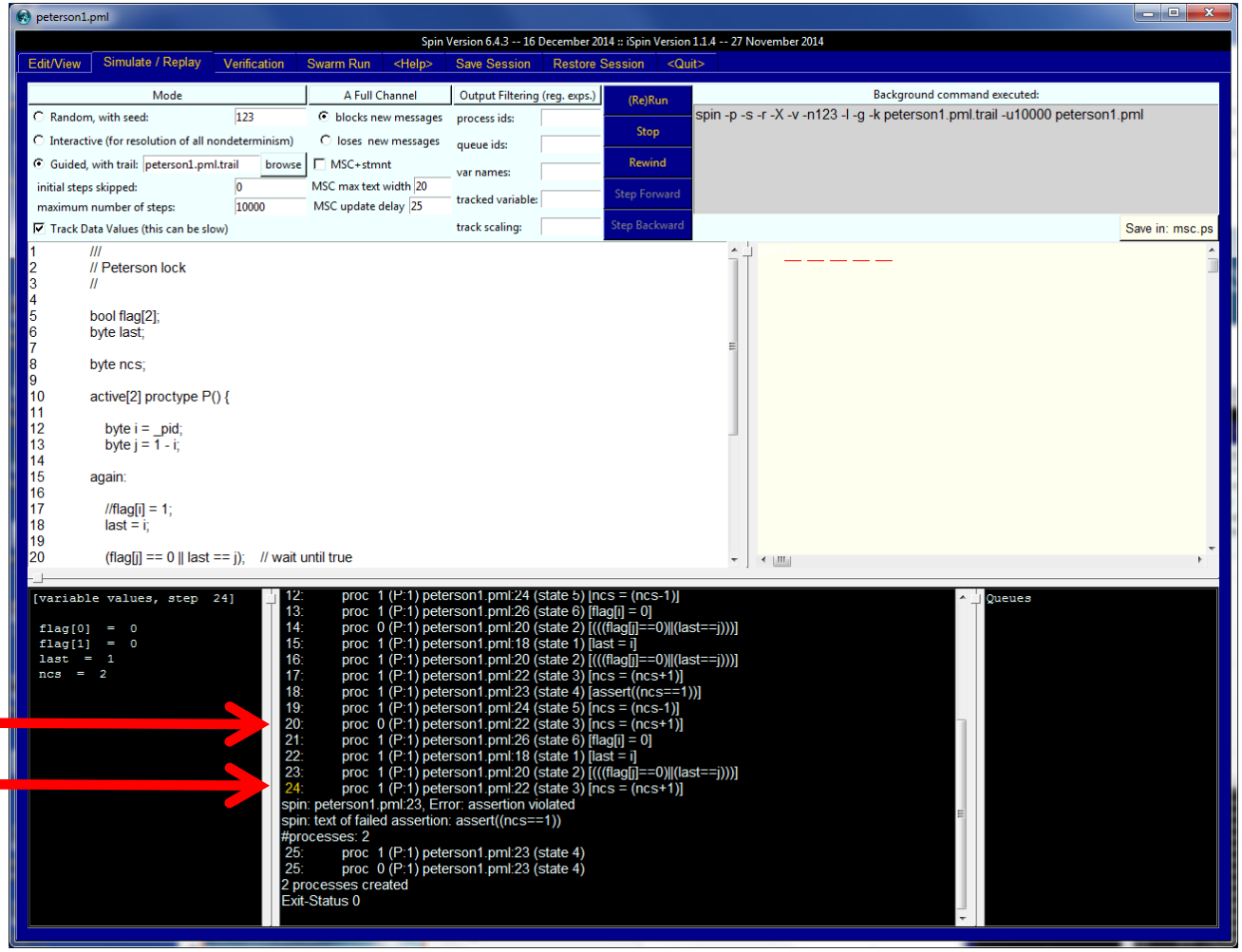
- // flag[i] = 1;
- assertion violated



Spin, Peterson and Bakery Locks

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 - C/C++ code that works

```
volatile int flag[2];              // NB volatile
volatile int victim;               // NB volatile

inline void acquire(int me) {   // variable names
changed
    int j = 1 - me;
    flag[me] = 1;
    victim = me;
    _mm_mfence();                  // NB synchronisation
    while (flag[j] && victim == me);
}

inline void release(int me) {
    flag[me] = 0;
}

inline void init() {
    flag[0] = flag[1] = 0;
}
```

variable
names
changed from
previous examples

Peterson Lock - C/C++ testing framework

```
UINT64 cnt = 0;                    // shared global counter
UINT64 t = getWallClockMS();       // get time

WORKER worker(void _pid) {
    UINT64 lcnt = 0;               // increment local counter
    while (1) {                    //
        lcnt++;                    // increment
        aquire(_pid);              // get lock
        cnt++;                     // non atomic increment
        release(_pid)              // release lock
        if (getWallClockMS() - t > NSECS*1000)
            break;
    }
}
```

- create two threads to execute worker function concurrently
- threads run for NSECS concurrently
- when ALL threads have finished, check that the sum of the local counters == cnt
- if NOT (cnt may be less than the sum of the local counters), the lock is unsafe
- demonstration

Volatile

- flag and victim must be declared volatile
- description of volatile from Visual Studio documentation

  objects that are declared as volatile are not used in certain optimizations because their values can change at any time. The system always reads the current value of a volatile object when it is requested, even if a previous instruction asked for a value from the same object. Also, the value of the object is written immediately on assignment.

- to declare object pointed to by a pointer as volatile use:

```
volatile int *p;            // what p points to is volatile
```

- to declare the pointer itself volatile use:

```
int * volatile p;           // contents of p is volatile
```

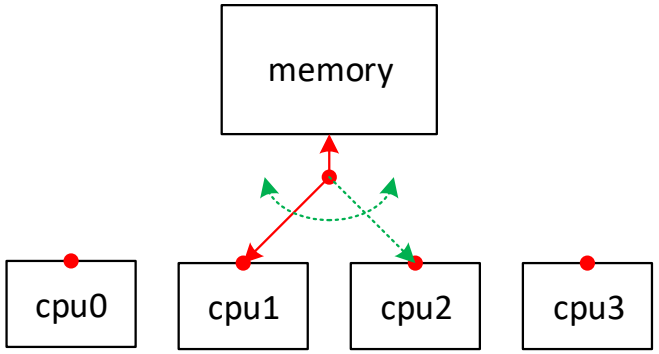
- both

```
volatile int* volatile p;   // p and what p points to are both volatile
```

Sequential Consistency

- programming model that can be used and understood by programmers

- **definition:** *a multiprocessor system is sequentially consistent if the result of any execution is the same as if the operations of all the processors were executed in some sequential order, and the operations of each individual processor appear in the order specified by its program* [Leslie Lamport 1979]



- an individual processor's memory accesses are made in program order
- accesses made by the different processors are interleaved arbitrarily AND memory accesses are seen in the same order by ALL processors

CPU Memory Ordering

- program order maybe relaxed to gain performance
- X -> Y means than X must complete before a later Y
- sequential consistency requires maintaining all 4 orderings

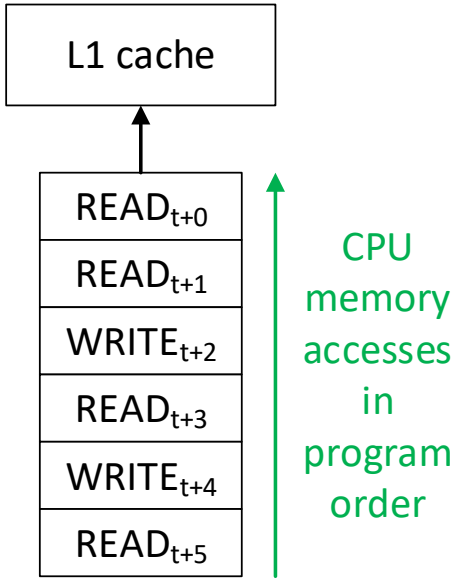
  R_t -> W_{t+n}, R_t -> R_{t+n}, W_t -> R_{t+n} and W_t -> W_{t+n}

- relaxing W_t -> R_{t+n} is known as ***processor ordering*** *or total store ordering* [reads can move ahead of writes]

- relaxing W_t -> W_{t+n} known as *partial store ordering*

- relaxing R_t -> W_{t+n} or R_t -> R_{t+n} gives variations known as *weak ordering*, *release consistency*, …

- with relaxed CPU memory ordering, sequential consistency is normally enforced at synchronisation points using serialising instructions

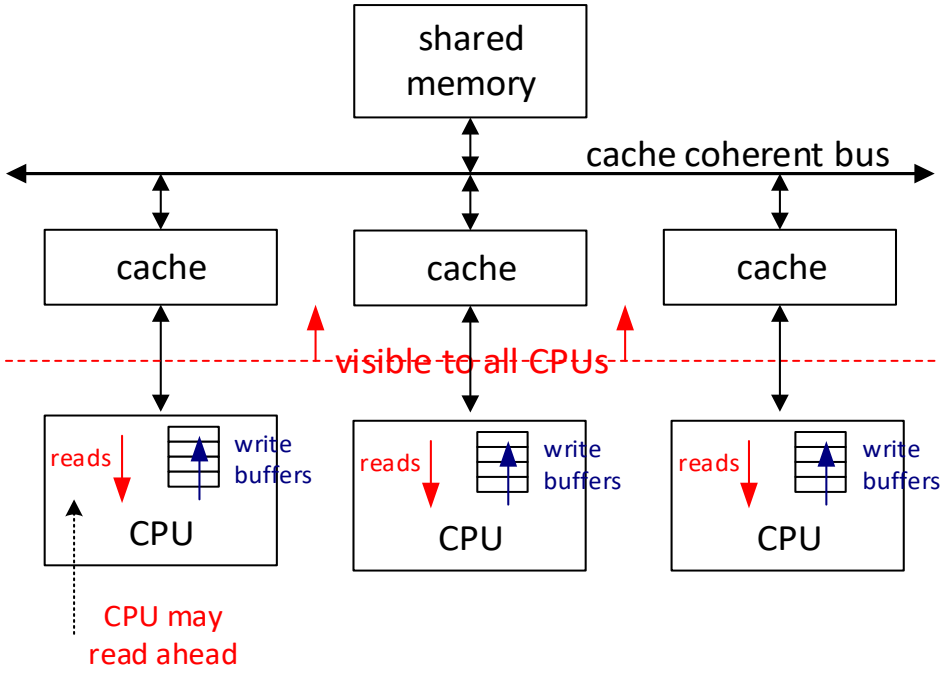


Intel IA32/x64 Memory Ordering

- IA32/x64 uses the processor ordering memory model [relax W_t -> R_{t+n}]
- see section 8.2 on Memory Ordering in Intel® 64 and IA-32 Architectures Software Developer's Manual Volume 3A: System Programming Guide, Part 1 and Intel® 64 Architecture Memory Ordering White Paper
- explicit *fence* instructions are used to enforce memory ordering and to flush the CPU write buffer so the writes are visible to other CPUs
  - LFENCE load fence — doesn't read ahead until instruction *executed*
  - SFENCE store fence — flushes all writes from write buffer to L1 cache before *executing* instruction
  - MFENCE memory fence — flushes all writes from write buffer to L1 cache before executing instruction AND … doesn't read ahead until instruction *executed*

Serializing Instructions

- from a hardware perspective...
- CPU has an internal write buffer which is used to buffer writes to the memory hierarchy [for improved performance]
- data in write buffer not visible to other CPUs until written to L1 cache [written by CPU asynchronously]
- SFENCE and MFENCE *wait* until write buffer flushed to cache
- MFENCE enforces sequential consistency (while data in write buffer, CPUs can read different values for the same memory address)



writes to L1 cache seen "instantaneously" by ALL processors

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

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

3   void acquire(int pid) {                         // pid is thread ID
4         choosing[pid] = 1;
5         int max = 0;
6         for (int i = 0; i < MAXTHREAD; i++) {     // find maximum ticket
7               if (number[i] > max)
8                     max = number[i];
9         }
10        number[pid] = max + 1;                    // our ticket number is maximum ticket found + 1
11        choosing[pid] = 0;
12        for (int j = 0; j < MAXTHREAD; j++) {     // wait until our turn i.e. have lowest ticket
13              while (choosing[j]);                // wait while thread j choosing
14              while (number[j] && ((number[j] < number[pid]) || ((number[j] == number[pid]) && (j < pid))));
15        }
16  }

17  void release(int pid) {
18        number[pid] = 0;                          // release lock
19  }
```

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[j])* 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 *higher-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

```
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 the compiler would generate code that only makes one access to number[j] by transforming statement into

```
while ((v = number[j]) && ((v < number[pid]) || ((v == number[pid]) && (j < pid))));
```

  where v is a local variable

- THUS original statement can be used AS IS, but what would happen if compiler generated code such that number[j] was read 3 times?

Tutorial 1

- Bakery or Black and White lock
- prove lock has the following desirable properties: safety, deadlock free, liveness and starvation free
- will discuss how to test for liveness and starvation freedom later
- need to get Spin working on laptop/desktop
- the Bakery Lock state space is unbounded as there is an interleave where the allocated ticket number keeps increasing
  - need to bound state space eg. 3 processes (N) each getting lock 3 times (CNT)
  - takes a couple of seconds to prove safety property if N = 3 and CNT = 3
- the Black and White Lock state space is bounded as max ticket is 2N
  - takes a couple of seconds to prove safety property if N = 3 and a couple of hundred seconds if N = 4