Testing Concurrent Software
using Model Checking Techniques and POSIX-Threads

Dr. Wolfgang Koch

Friedrich Schiller University Jena

Faculty of Mathematics and Computer Science

Jena, Germany

wolfgang.koch@uni-jena.de

Motivation

2

Errare humanum est,

in errore perseverare stultum.

(To err is human, to persist in error is stupid.)

Contents

3

Motivation

4

The importance of **concurrent programming** is rapidly growing as multi-core processors replace older single core designs.

Today almost all PCs and Laptops have a **multi-core** (e.g. quad-core) processor

using SMP (symmetric multiprocessing) with shared memory and cache coherence

Concurrent software consists of **competing and cooperating** processes or threads.

Additional fault types exist in concurrent software compared to sequential software. Subtle **interactions among threads** and the timing of asynchronous events can result in concurrency errors that are hard to find, reproduce, and debug.

Motivation

5

Additional fault types exist in concurrent software:

Failures in sequential programs are **deterministic** – if a sequential program fails with a given set of inputs and initial state, it will fail every time.

Failures in concurrent programs, on the other hand, tend to be rare **probabilistic** events.

Unexpected interference among threads often results in "Heisenbugs" that are extremely difficult to reproduce and eliminate.

Software Test

6

Testing is the process of **executing a program** with the **intent of finding errors.**
(The art of software testing, Glenford J. Myers)

E. W. Dijkstra:
Program testing can be used to show the presence of bugs, but never to show their absence!
(The Humble Programmer, ACM Turing Lecture 1972)

This famous saying is formally correct, but completely misleading.

The fact is that NOTHING, not inspection, not formal proof, not testing, can give 100% certainty of no errors.
Yet all these techniques, at some cost, can in fact **reduce the errors** to whatever level you wish.

"You don't have to test anything unless you want it to work."

Software Test

7

Testing is the process of executing a program with the intent of finding errors.
(The art of software testing, Glenford J. Myers)

Testing can find faults

When they are removed, software quality and reliability is improved

- Build confidence
- Demonstrate conformance to requirements
- Assess the software quality

Companies spent 30 – 50% time and budget of their software development on testing

depending on the risks for the system
(loss of money, loss of market share, death or injury)

Software Test

8

Purpose of testing: build confidence

The testing paradox

Purpose of testing: to find faults

Finding faults destroys confidence

Purpose of testing: destroy confidence ???

The best way to build confidence
is to try to destroy it!

Defect - Error - Bug - Failure - Fault ?

9

No generally accepted set of testing definitions used world wide

(new) standard BS 7925-1 (Glossary of testing terms)

adopted by the ISEB / ISTQB

- Error: a human action that produces an incorrect result
- Fault: a manifestation of an error in software (a state)
  - also known as a defect or bug
  - if executed, a fault may cause a failure
- Failure: deviation of the software from its expected delivery or service (an event)
  - (found defect – debugging necessary)

Defect - Error - Bug - Failure - Fault ?

10

Standard BS 7925-1 (Glossary of terms in software testing)
developed by a working party of the BCS SIGIST,
adopted by the ISTQB

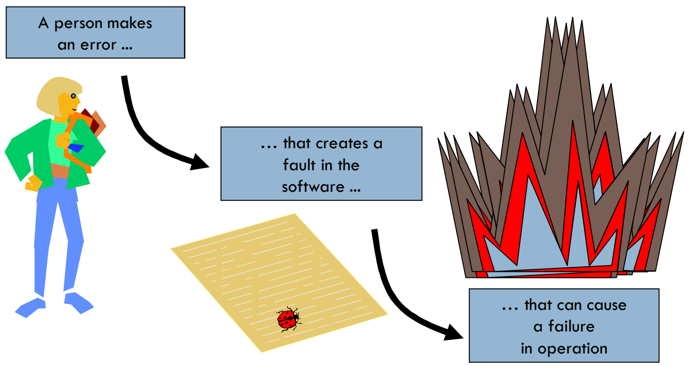
BCS - British Computer Society
BCS SIGiST - Specialist Group in Software Testing

ISTQB - International Software Testing Qualifications Board
has defined the "ISTQB® Certified Tester" scheme
that has become the world-wide leader in the certification
of competences in software testing.

Hungarian Testing Board (HTB) - www.hstqb.org
Magyar Szoftvertesztelői Tanács Egyesület,
H-1117 Budapest, Neumann Janos u.1. Infopark "E"

Error - Fault - Failure

12



Source: ISTQB / ISEB Foundation Exam Practice

Sources of Errors in Concurrent Software

13

Additional Errors in Concurrent Software

- **Deadlock**, where task A can't continue until task B finished, but at the same time, task B can't continue until task A finishes.
- **Race condition**, where the computer does not perform tasks in the order the programmer intended.
- Concurrency errors in **critical sections**, mutual exclusions

Hard to reproduce
"Heisenbugs"

A neglect of the programmer:
One has to deal with the possible sources of **nondeterminism** in concurrent software.

Blocking Queue

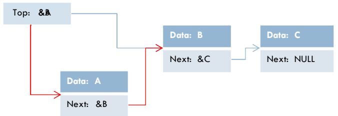
14

Example: “Blocking” LIFO Queue (a Stack):

```
shared Node * Top;
shared Lock lock;

void push(Node *node)
{
  Node *t;        // local pointer

  acquire(&lock);
   t = Top;
   node->Next = t;
   Top = node;
  release(&lock);
}
```



Wait-freedom, Lock-freedom

15

One disadvantage of locks:
If a thread holding a lock blocks, all waiting threads are blocked too, no one is making any progress

A wait-free operation is guaranteed to complete after a finite number of its own steps, regardless of the timing behavior of other operations.

A lock-free operation guarantees that after a finite number of its own steps, some operation (possibly in a different thread) completes (also called nonblocking).

wait-freedom is a stronger condition than lock-freedom
wait-freedom is hard to achieve (and only with a lot of overhead)

Our queue with locks is neither *wait-free* nor *lock-free*

Lock-free method

16

Disadvantages of locks → request for a lock-free method,

make changes on a copy, then set the copy into effect in a single atomic step - if the original has not changed

```
boolean try_push(Node *node)
{ boolean  res;
  Node     *t;         // local pointer

  t = Top;             // local copy
  node->Next = t;      // still private node
  // Top = node;       // global - Danger!
  atomic( if (Top==t) {Top=node; res=true;}
          else res=false;     // try again
        )
  return res;
}
```

Lock-free method, CAS

17

. . . set the copy into effect in a single atomic step
if the original has not changed

```
atomic( if (Top==t) {Top=node; res=true;}
        else res=false;    // Top not changed
      )
```

We need an **atomic primitive** that accomplishes this task
(TSB and XCHG are not strong enough)

IBM introduced `CompareAndSwap (CAS)` in the IBM 370

```
res = CAS(&mem, expected, new);    // boolean CAS
```

(Intel 1989: `CMPXCHG` – Compare and Exchange)

Lock-free methods

19

```
void push(Node *node)
{ Node *t;
  while(true){
    t = top;
    node->Next = t;
    if (CAS(&top,t,node)) break;
  }
}
```

is lock-free:

if CAS succeeds, our thread completes the push-operation
if CAS fails, it failed because another thread has changed `top`
so the CAS of that other thread succeeded
the other tread has completed its (push-) operation

Lock-free pop-operation

20

```
Node * pop(void)
{
  Node *t, *next;
  while(true){
    t = top;
    if (t == NULL) break;     // empty stack
    next = t->Next;
    if (CAS(&top,t,next)) break;    // lock-free
  }
  return t;
}
```

There might be a problem: we use a pointer to a node (`t,t->Next`), but that node may be freed meanwhile by another thread (in systems without garbage collection) – problem of **data persistence**.

In addition (more serious): the ABA-problem

Stress Test

21

In practice, people almost always identify concurrency testing with **stress testing**, which evaluates the behavior of a concurrent system under heavy load for a long time.

While stress testing does indirectly **increase the variety of thread schedules**, such testing is far from sufficient.

Stress testing does not cover enough different thread schedules and, as a result, yields unpredictable results.
A bug may surface one week, when stress testing happens to cover a low-probability schedule, and then disappear for months.

"**Heisenbugs**" that rarely surface and are hard to reproduce

Stress Test Scheme

22

In practice, people almost always identify concurrency testing with **stress testing**, which evaluates the behavior of a concurrent system under load for a long time.

```
while(true){
  Setup_Test();
  RunTestScenario();
  err = CheckErrors();
  Shutdown_Test();
  if(err) break;      // Error, Timeout, etc.
}
```

Stress Test Example

23

My Test-Example: We have a queue with 4 nodes and then concurrently pop 3 nodes and push one additional node. (OK – it's not really heavy load but it works – and we need it this way later with model checking)

`Setup_Test():`
Create a queue with 4 nodes

`RunTestScenario():`
Start 4 threads: 3 ThreadPop, 1 ThreadPush
Wait for the ending of all threads (`pthread_join();`)

`Shutdown_Test();`
Delete remaining queue, free nodes

Stress Test Example

24

Concurrently pop 3 nodes and push one additional node:

```
ThreadPush(Params)
{
  Node *node = new(Node);
  node->Data = Params->Value;
  push(node);
}

ThreadPop(Params)
{
  Node *node = pop();
  if (node) Params->Value = node->Data;    //store Data
  delete(node);
}
```

Stress Test Example

25

Concurrently pop 3 nodes and push one additional node:

```
ThreadPush(Params)
{ new(Node); … push(node); }

ThreadPop(Params)
{ node = pop(); … delete(node); }
```

Running this test for a long time showed no **failures!**

Most of the time short blocks (threads) will run to completion without preemption.
This limits the likelihood that race conditions will be disclosed.

Enhancement: **Insertion of random delays**

Stress Test with Delays

26

Running the test for a long time showed no failures.

Enhancement: Insertion of random delays in push and pop:

```
t=top;
…
if (do-test) Sleep(wait_rand);
if (CAS(&top,t,next)) break;
```

`wait_rand:` small numbers – milliseconds

20% - no Sleep()
20% - Sleep(0)
30% - Sleep(1)
20% - Sleep(3)
10% - Sleep(9)

The tool ConTest (IBM) does something like this automatically for Java applications

Stress Test with Delays

27

Enhancement: `if (do-test) Sleep(wait_rand);`

Now in most cases I got a failure within the first 50 ... 150 passes.

But what went wrong? (I.e. I found a failure, not the defect!)

Adding `printf` (attention - this may cause the failures to disappear):

T1: 9 T2: x T3: 1 T4: 3 T4: 1 - Error!

Analysis:

t0: pop1.read - sleep 9, pop2.read - no sleep - pop2.CAS+
(+ **free** Node), pop3.read - sleep 1, push4.read (+ **new**) - sleep 3

t1: pop3.CAS+

t3: push4.CAS–, push4.read again - sleep 1

t4: push4.CAS+ ABA-prone

t9: pop1.CAS+ ABA occurred !

ABA-problem

28

Is ABA really a problem ? (the value has not changed)

Yes, it can – of cause – the data structure may have changed.

Imagine, we have a stack:

top --> A --> B --> C --> /

thread1 - pop():

```
t = top;              // top  = &A
next = t->Next;       // next = &B
            // thread2: A=pop, B=pop, push A
            // top --> A --> C --> /
if (CAS(&top,t,next)) break;    // succeeds !
            // top --> B --> ??    -- Error !!
```

Stress Test, ABA-prevention

29

ABA-prevention

we don't call new() and delete() within the threads,
but use a pool of Nodes – each thread has it's own Node

```
ThreadPush(Params)
{
  Node *node = pool[Params->Nr];   //new(Node);
  node->Data = Params->Value;
  push(node);
}
```

Now the test runs without failure for an arbitrary long time!

(There is no serious ABA in systems with garbage collection
and on RISC machines with LL/SC instead of CAS)

Linearization

30

Correctness Proof – **Safety**: guaranteeing that nothing bad happens

The safety aspects of concurrent data structures are complicated by the need to argue about the **many possible interleavings** of methods called by different threads.
It is infinitely easier and more intuitive for us humans to specify how abstract data structures **behave in a sequential setting**, where there are no interleavings.
Thus, the standard approach to arguing the safety properties of a concurrent data structure is to specify the structure's properties sequentially, and find a way **to map its concurrent executions** to these "correct" sequential ones.
(serializability, linearizability)

Model Checking Techniques

31

Different approach:

use of **model checking techniques**

to **systematically generate all interleavings** of a given scenario

A model checker essentially captures the nondeterminism of a system and then systematically enumerates all possible choices.

For a multithreaded process, this approach is tantamount to running the system under a **demonic scheduler**.

I first learned about this technique from a paper by Madanlal Musuvathi et. all.

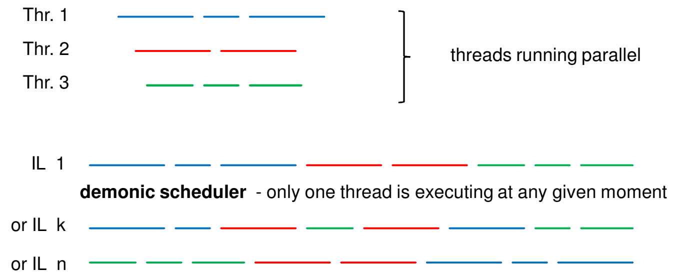
CHESS: A Systematic Testing Tool for Concurrent Software Technical Report Microsoft Research

Model Checking Techniques

32

A model checker systematically generates all possible interleavings

(example: 3 threads with 2 or 3 'atomic' parts)



Model Checking Techniques

33

Another way to tell the same story:

The model checker abstracts a program as a **nondeterministic state transition system** in which each transition is executed by a task (a thread).

Given a state and task enabled in it, executing the task results in a unique new state.

Nondeterminism arises because in each state more than one task may be enabled and any one of them may be scheduled.

Starting from the initial state, an execution is obtained by iteratively picking an enabled task and executing it for one step.

Given the task abstraction and knowledge of the set of tasks enabled in a state, all such execution can be systematically generated in a straightforward manner.

Model Checking Techniques

34

A model checker systematically generates all possible interleavings.

Our stress test scheme is still valid:

```
while(true){              // for all interleavings
  Setup_Test();           // same testcase in every pass
  RunTestScenario();      // different interleaving
  err = CheckErrors();
  Shutdown_Test();
}
```

The model checker guarantees that every execution of RunTestScenario generates a new interleaving and that each such interleaving can be replayed (→ easy **debugging**).

Model Checking Techniques

36

Three key challenges in making model checking applicable:

1. Existing model checkers requires the programmer to do a huge amount of work just to get started. The "**Perturbation Problem**"
2. Concurrency is enabled via rich and complex **concurrency API**.

   We **wrap** the concurrency APIs to **capture and control** the nondeterminism, without changing the underlying OS or reimplementing the synchronization primitives of the API.

   The only perturbation here is **a thin wrapper layer** between the program under test and the concurrency API.
3. The classic problem of **state-space explosion**. The number of thread interleavings even for small systems can be astronomically large.

Wrapper Layer

37

The model checker controls the scheduling of tasks (threads) by **instrumenting** all functions in the concurrency API that create tasks and access synchronization objects.

```
(pthread_create(), CAS(), ReadGlobal(),
 pthread_mutex_lock(), …)
```

The idea is that when the instrumented function is executed, either prior the execution of function, or at its point of return, or both, a block of code in the model checker is able to gain control. It can obtain access to the function arguments, execute its own logic, and even decide whether or not the instrumented function will run at all, and with what argument values, and what it shall return.

Wrapper Layer

38

Instrumenting functions in the concurrency API – we write **wrappers** (a thin wrapper layer between the program under test and the concurrency API)

```
int Wrapper_CAS(void *mem, void exp, void new)
{
  int re;          // boolean

  if (do_test) MC_sched();
  re = Orig_CAS(mem, exp, new);
  if (do_test) MC_CAS_Result(mem,re);
    // to tackle the state-space problem

  return re;
}
```

Wrapper Layer

39

How can we apply the **wrapper layer** to the test program?

If we have access to the code of the test program:

```
#include "MC_Wrapper.h"
```

and link the model-checking module to the program.

If we don't have access to the code -

we can use **DLL Injection**

changing the addresses of the API routines in the **Import Address Table (IAT)** of the executable (.exe) file (I gave a lecture on API Hooking and DLL Injection in 2009)

Wrapper Layer

40

```
#include "MC_Wrapper.h"

int Orig_CAS(void * mem, void exp, void new)
{
  return CAS(mem, exp, new);
}

#define CAS(m,o,n)    Wrapper_CAS(m,o,n)

int Wrapper_CAS(void * mem, void exp, void new)
{
  int re;          // boolean
  if (do_test) MC_sched();
  re = Orig_CAS(mem, exp, new);
  return re;
}
```

Wrapper Layer

41

Given the knowledge of the set of tasks enabled in a state …

The model checker must know about active threads –
it needs additional scheduling points at the beginning
and at the end of each thread.

So we don't start (create) the original thread, but a Thread-Wrapper
that brackets the call to the original thread's function
by calls to the model checker (bookkeeping + MC_sched())

```
int Wrapper_pthread_create(&thr, 0, function, arg)
{
  tid = MC_NewThread();         // MC tread-ID, No.
  Closure c = <function, arg, tid>;
  return Real_pthread_create(&thr,0,ThreadWrapper,c);
}
```

Wrapper Layer

42

```
int Wrapper_pthread_create(&thr, 0, function, arg)
{
  tid = MC_NewThread();
  Closure c = <function, arg, tid>;
  return Real_pthread_create(&thr,0,ThreadWrapper,c);
}
```

A Thread-Wrapper that brackets the call to the original thread's function
by calls to the model checker

```
void * ThreadWrapper(Closure c)
{
  MC_ThrBegin(c.tid);   // Bookkeeping + MC_sched();
  retVal = c.function(c.arg);
  MC_ThrEnd(c.tid);     // -> MC_sched();
  return retVal;
}
```

MC Scheduler

43

The model checking approach is tantamount
to running the system under a demonic scheduler –

only one thread is executing at any given moment

```
void MC_sched(void)
{
  int  old,new;
  old = thr_old;          // global variable
  new = find_new();
  if (new < 0 ) return;   // end of test
  if (new==old) return;   // simply go on
  thr_old = new;
  . . .      // suspend old, resume new thread
}
```

MC Scheduler

44

```
void MC_sched(void)
{
  old = thr_old;  new = find_new();
  // suspend old, resume new thread
  ResumeThread(new);     // first
  if(tcbs[old].id)
    SuspendThread(old);
}
```

The MC-scheduler is running in the **context** of the **active** (the old) thread. So we cannot simply suspend the old thread and thereafter resume the new one. The scheduler would stop itself (and the old thread) immediately – the program would hang.

– we have to reverse things.

MC Scheduler

45

```
old = thr_old;  new = find_new();
ResumeThread(new);       // first
SuspendThread(old);
```

We cannot simply suspend the old thread and thereafter resume the new one – we have to reverse things.

But there might be a new problem: now (for a short time) new thread and old thread are running at the same time.

What happens, if the new thread schedules the old thread again, before the old thread reached it's suspend – will it then hang? (It gets the wake-up call before it starts sleeping.)

(Using Microsoft Threads we can **detect** this situation, to avoid hanging.)

MC Scheduler

46

How are things getting started?

– only one thread is executing at any given moment

```
void MC_ThrBegin(int tid)    // in ThreadWrapper()
{
  int nst = AtomicIncrement( &nstart );  //global
            // number_of_started_threads
  pthread_t id_own = pthread_self();
  tcbs[tid].id = id_own;  tcbs[tid].enabled = 1;
  if (nst < nthreads)  SuspendThread(tid);
  else{ thr_old = tid; MC_sched(); }
}
```

MC Scheduler

47

How are things getting started?

```
nst = AtomicIncrement( &nstart );
```

We must initialize `nstart` in the beginning of each test pass to 0.

We can do this (and other initializations) in the instrumented `Setup_Test()` routine.

And how do things end?

```
void MC_ThrEnd(int tid)    // in ThreadWrapper()
{
  tcbs[tid].id = 0;    // or NULL;
  MC_sched();
}
```

POSIX Threads

48

In an earlier project (talk) I used Microsoft-Threads in the test example and in the Model Checking wrapper layer.

Now I want to use POSIX Threads (pthreads) in order to apply the Model Checking technique also in UNIX / Linux.

```
#include <pthread.h>
gcc ... -lpthread
```

The great picture is very similar in both environments:

- The thread‘s function is coded as a C function with one parameter, returning a status value (a void-pointer in pthreads). Just one argument is OK for the address of an arbitrary struct.

POSIX Threads

49

The great picture is very similar in both environments:

- The thread‘s function is coded as a C function with one parameter, returning a status value
- The thread is created (and started) by a function: `pthread_create()` with 4 parameters: the function of the thread, the argument for that function, additional attributes (0 for the standard) and a variable to store the ID of the created thread – later used to identify the thread
- The thread ends when its function returns or when another thread calls `pthread_cancel(ID)`.
- We can wait for threads: `pthread_join(ID, NULL);` (the second parameter is the address of the thread‘s return value)

POSIX Threads

50

Example using pthreads:

```
#include <pthread.h>

pthread_t  id[NUM_THREADS];
int        i, rc;

void * thr_fkt(void *);

for (i=0; i<NUM_THREADS; i++) {
  rc = pthread_create(&id[i], NULL,
                      thr_fkt, (void *)(i+1));
  if (rc != 0){ printf("Error at i=%d - rc=%d \n", i, rc);
}

for (i=0; i<NUM_THREADS; i++) {
  pthread_join(&id[i], NULL);
}
```

POSIX Threads

51

To inquire the ID of your thread use `pthread_self()`.

Don’t compare thread IDs by their value, use `pthread_equal()` instead.

```
void * thr_fkt(void * arg)
{
   int         b=0, a= (int) arg;
   pthread_t   id_own;

   id_own = pthread_self();
   while (1){
     if( pthread_equal(id_own, id[b++])){ break; }
   }
   // b == a ??
   return NULL;      // pthread_exit(NULL);
}
```

POSIX Threads - Synchronization

52

Using threads there is always the problem of synchronizing the use of shared resources – e.g. write to shared variables.

The Posix Thread system provides pthread **Mutexes** and pthread **Condition Variables**.

```
pthread_mutex_t  mutex = PTHREAD_MUTEX_INITIALIZER;
// static mutex  -  dynamic: pthread_mutex_init()

int pthread_mutex_lock(pthread_mutex_t *mutex);
int pthread_mutex_trylock(pthread_mutex_t *mutex);
int pthread_mutex_unlock(pthread_mutex_t *mutex);
```

We can also use **Semaphores**.

```
#include <semaphore.h>
```

POSIX Threads – Suspend and Resume

53

The great picture is very similar in both environments – but there is one big difference:

There is no `SuspendThread()` and no `ResumeThread()` in pthreads! There is nothing similar either. What can we do?

I could use pthread Condition Variables. I use `pthread_cond_wait()` to **suspend** the current thread and `pthread_cond_signal()` to **resume** it later.
And I probably get the problem of the early wake up call in `MC_sched()`

But I have a better idea – I use **semaphores**.
I use one semaphore per thread and initialize it to 0.

Then I use `sem_wait()` to suspend a thread and `sem_post()` to resume it.

POSIX Threads – Suspend and Resume

54

```
#include <semaphore.h>
```

In `MC_ThrBegin(int tid)` I call

```
tcbs[tid].id = id_own;   tcbs[tid].enabled = 1;
sem_init(&tcbs[tid].sema, 0, 0);   // shared, value
...
```

In `MC_ThrEnd(int tid)` I call

```
tcbs[tid].id = 0;
sem_destroy(&tcbs[tid].sema);
MC_sched();
```

```
void SuspendThread(int tid){ sem_wait(&tcbs[tid].sema); }
void ResumeThread( int tid){ sem_post(&tcbs[tid].sema); }
```

Excursion on Semaphores

55

Semaphores are a means of synchronization.
(It performs no 'active wait', the waiting threads are sleeping.)

A semaphore can be thought as an **integer variable** with 2 functions: `up()` and `down()`.
(In pthreads they are called `sem_post()` and `sem_wait()`.)

An additional function `init()` is used to set the variable to a value ≥0.

```
down(sema) {
 if (sema.value > 0) sema.value -= 1;
 else { put thread to sleep (in a queue) }
}

up(sema) {
 if (thread(s) in the queue){ wake up (one) thread }
 else sema.value += 1;
}
```

Suspend and Resume

56

```
sem_init(&tcbs[tid].sema, 0, 0);   // shared, value

SuspendThread(int tid){ sem_wait(&tcbs[tid].sema); }
ResumeThread(int tid) { sem_post(&tcbs[tid].sema); }
```

Every thread has its own semaphore (in its `tcb`), initialized to 0.

`SuspendThread()` tries to decrease the value.

Since it is 0, the thread is put to sleep (is suspended).

`ResumeThread()` finds the sleeping thread and wakes it up (resumes it).

Suspend and Resume

57

```
sem_init(&tcbs[tid].sema, 0, 0);   // shared, value

SuspendThread(int tid){ sem_wait(&tcbs[tid].sema); }
ResumeThread(int tid) { sem_post(&tcbs[tid].sema); }
```

What about the problem, if the new thread schedules the old thread again, before the old thread reached it's suspend – will it then hang? (It gets the wake-up call before it starts sleeping.)

`ResumeThread()` increases the value to 1.

`SuspendThread()` decreases the value to 0.

The thread is not put to sleep.
It is still running – as it was intended.

MC Scheduler, Generating Interleavings

58

The code of `MC_sched()` just shows the big picture.

But we left the task of systematically generating all possible interleavings to `find_new()`.

We use a **Backtracking Algorithm**
similar to generating all permutations of a set of numbers.

Problem here:
Backtracking means to go back in a list sometimes –
but in our list of steps of threads we cannot simply go back
– we (usually) cannot undo a performed step of computation

So instead of going back in the list, we replay the list
from the beginning up to the point where the changes start.

MC Scheduler, Generating Interleavings

59

Backtracking Algorithm
similar to generating all permutations of a set of elements.

All positions in the list are initialized to 0 (empty) and k=1 (1st pos.)

\# When at position k>0 in the list:

a := list[k]; if (a>0) free[a] += 1; // mark a as free;

choose the lowest free element b>a
(the thread (enabled threads only) with the lowest number b>a)

- o if there is such an element/thread b
  list[k] := b; free[b] −= 1; // mark b as used
  go to the right (k := k+1)
- o otherwise: list[k] := 0; go to the left (k := k–1)
  (if a was 0, we found a new permutation / our test run is complete,
  go to the left for one more permutation / a new test run)

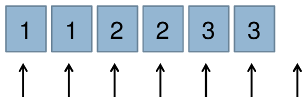
MC Scheduler, Generating Interleavings

60

a := list[k]; choose the lowest free element b>a

```
if (b)
  list[k] := b;  k := k+1;
else
  list[k] := 0;  k := k−1;
```

1 1 2 2 3 3



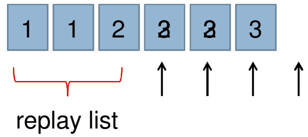
MC Scheduler, Generating Interleavings

61

a := list[k]; choose the lowest free element b>a

```
if (b)
  list[k] := b;  k := k+1;
else
  list[k] := 0;  k := k−1;
```

1 1 2 2 3 3



MC Scheduler, Generating Interleavings

62

We cannot undo a performed step of computation.

Instead of going back in the list, we replay the list from the beginning up to the point where the changes start.

(The scheduler keeps track of this point.)

Is a perfect replay always possible?

No, if there are other sources of nondeterminism:

- Different input values or initial states
- Calls to `time()` or `random()`
- Asynchronous I/O

We always enforce the same initial state using `Setup_Test()`

MC Scheduler, Blocking Operations

63

The model checker must keep track of the **set of enabled threads** in the presence of potentially **blocking operations**.

The wrapper function for `pthread_mutex_lock` (for example) cannot simply call `pthread_mutex_lock()` directly:

```
int Wrapper_pthread_mutex_lock(mutex);
{
    MC_SyncVar(mutex, ACQUIRE);
    return pthread_mutex_lock(mutex);     // wrong!!
}
```

If the mutex is currently held by another thread, the MC-scheduler will deadlock!

The calling thread is the only running thread, **it must not block**.

MC Scheduler, Blocking Operations

64

The wrapper function for `pthread_mutex_lock` cannot simply call `pthread_mutex_lock()` directly (it must not block)
– instead it just tries with the non-blocking function …`_trylock()`

```
int Wrapper_pthread_mutex_lock(mutex);
{
  while(true){
    MC_SyncVar(mutex, ACQUIRE);
    if(pthread_mutex_trylock(mutex)==0) return 0;
    MC_SyncVar(mutex, BLOCKED);    // MC_sched()
  }
}
```

If the lock is held by another thread …`_trylock` returns `EBUSY` .

MC Scheduler, Blocking Operations

65

If the mutex is held by another thread, `MC_SyncVar(mutex,BLOCKED)`

- disables the current thread,
- adds it to the set of threads waiting on `mutex`
- and schedules a new (active) thread.

Later on

```
int Wrapper_pthread_mutex_unlock(mutex)
{
   MC_SyncVar(mutex, RELEASE);
   return pthread_mutex_unlock(mutex);
}
```

re-enables all threads waiting on `mutex`.

First Results

66

Same example as with stress test:

We have a queue with 4 nodes and then concurrently pop 3 nodes and push one additional node.

But instead of `Sleep(wait_rand)` we now call `MC_sched()`

pass 300 # 1.1 2.1 1.4 3.1 3.4 4.1 4.4 2.4 ## Error!!

Analysis:

1.1 … 3.4 – Number_of_Thread . Operation

Operation:

1 - Read, 3 - CAS–, 4 - CAS+

First Results

67

We have a queue with 4 nodes and then concurrently pop 3 nodes and push one additional node.

pass 300 # 1.1 2.1 1.4 3.1 3.4 4.1 4.4 2.4 ## Error!!

Analysis (is now simple, steps happened sequentially):

pop1.read, **pop2**.read, pop1.CAS+ (+ free Node),
pop3.read, pop3.CAS+ ,
push4.read (+ new Node), push4.CAS+ ABA-prone
pop2.CAS+ ABA occurred !

top --> A --> B --> C --> D --> / pop2.read: next = &B
top --> A* --> C --> D --> / top --> B!! --> ??

First Results

68

We have a queue with 4 nodes and then
concurrently pop 3 nodes and push one additional node.

Sometimes ABA is correct: read, push(), pop(), CAS+

pass 31 # 1.1 1.4 2.1 3.1 4.1 4.4 2.3 2.4 3.4 (no Error)

Analysis:
pop1.read, pop1.CAS+
pop2.read, pop3.read,
push4.read, push4.CAS+
pop2.CAS–, pop2.CAS+, pop3.CAS+ ABA here correct

First Results

69

We have a queue with 4 nodes and then
concurrently pop 3 nodes and push one additional node.

With **ABA-prevention**
(we don't call new() and delete() within the threads,
but use a pool of Nodes – each thread has it's own Node)

Stress Test (with random delays) showed no results (no failures).

Also the **Module Checker Test** runs without failure.

Since we systematically tested all possible interleavings,
this is more a **proof**, an (automated) formal verification than a test.
– If the test scenarios are thoroughly chosen and all
essential scheduling points are utilized.

Tackling the State-Space Problem

70

The problem of **state-space explosion**:

the number of thread interleavings even for small systems
can be astronomically large.

Possibilities:

- Scope preemptions to code regions of interest
- Different Modes – speed vs coverage
- Don't analyze redundant interleavings

Tackling the State-Space Problem

71

Different Modes: speed vs coverage

Fast mode - Introduce schedule points only before
synchronizations and possibly volatile accesses
(also called preemption bounding)

Finds many bugs in practice (Less often is more!)

Data-race mode - Introduce schedule points before memory accesses
Finds race-conditions due to data races

Tackling the State-Space Problem

72

Don't analyze **redundant** interleavings.

Two steps are **independent** (and can change their place) if

- They are executed by different threads and
- either they access different variables
  or READ (not WRITE !) the same variable

Interleavings which only differ in the order of independent steps have the same result – only one of them needs to be analyzed.

Unsuccessful CAS operations also only READ a variable, but we cannot easily know in advance, whether the CAS will be successful or not.

Tackling the State-Space Problem

73

We cannot know in advance, whether the CAS will be successful.

My approach:
Try the CAS.
Deliver the CAS status to the scheduler (`MC_CAS_Result(mem,re)`)
Cancel this run if the CAS was unsuccessful and the interleaving is redundant.
The point up to where the list will be replayed is shifted to the left of the position of the CAS.
So all following interleavings with the same reason of redundancy are skipped automatically.

Results: LIFO: 1 488 instead of 36 936 – 4%
FIFO: 65 964 instead of 11 887 944 – 0.5%

Tackling the State-Space Problem

74

We must be careful –

In our example **push4.read** means: run the thread until the read operation (`top`), but not until `CAS`.
But that includes the `new(Node)` operation in the `ThreadPush` thread.

As we have seen, the order of `new` and `delete` operations can be important for occurring the ABA problem.

So our push_x.read operations are not really independent, interleavings which only differ in the order of such operations are not redundant.

Fortunately in our LIFO stack example we have just one push thread, there is no problem. But otherwise one has to pay attention.

FIFO Queue

75

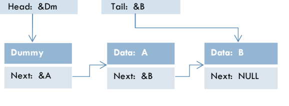
Two entry points (pointers): Node ***Head**, ***Tail**;

To avoid special cases (the empty queue)
the queue always includes a **dummy node** as the first node

Introduced by Michael and Scott (→ the MS-queue)
included in the standard JavaTM Concurrency Package (JSR-166)

We **enqueue** at the **tail** (after the so far last node)

we **dequeue** at the **head** (unless the queue is empty)
we read the next node
after the dummy,
then this node becomes
the new dummy



FIFO Queue

76

We **enqueue** data at the **tail**

we create a new Node:

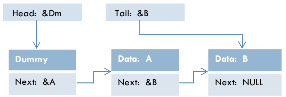
```
Node * node = new(Node);
node->Data  = data;
node->Next  = NULL;      // important!
```

to enqueue this node we have to **change two pointers**:

first – the Next-field of the so far last node (was NULL)

second – Tail

(not possible
in one single
atomic step)



FIFO Queue

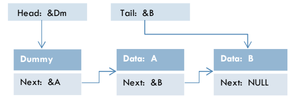
77

To enqueue a node we have to **change two pointers**: Next and Tail
A first (incomplete) routine looks like this:

```
   void Enqueue(Type data)
   { Node *node, *t, *next;

1:   node = new(Node);
     node->Data = data; node->Next = NULL;

     while(true){
2:     t = Tail;
4:     if (CAS(&t->Next,NULL,node)) break;
     }
5:   CAS(&Tail,t,node);
   }
```



FIFO Queue

78

```
    while(true){
2:    t = Tail;
      next = t->Next;
3:    if (next != NULL) {CAS(&Tail,t,next); continue}
4:    if (CAS(&t->Next,NULL,node)) break;
    }
5:  CAS(&Tail,t,node);
```

If one thread has performed step 4, but not yet step 5 (when it is blocked)
other threads cannot succeed in step 4 (t->Next != NULL)
→ the algorithm (so far – without step 3) is **not lock-free** !

To repair this, threads must be able to adjust Tail
(step 3 in our tread instead of step 5 in the blocking thread)

– our thread assists the obstructing tread

FIFO Queue

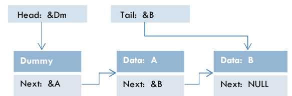
79

The complete, lock-free routine:

```
void Enqueue(Type data)
{ Node *node, *t, *next;

  node = new(Node);
  node->Data = data; node->Next = NULL;

  while(true){
    t = Tail;
    next = t->Next;
    if (next!=NULL) {CAS(&Tail,t,next); continue}
    if (CAS(&t->Next,NULL,node)) break;  //lin. point
  }
  CAS(&Tail,t,node);
}
```



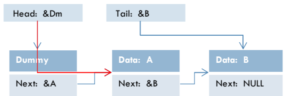
FIFO Queue

80

To dequeue we (usually) change only one pointer - Head
(step 3 is analogous to step 3 in Enqueue)

```
    Type Dequeue(void)
    {  Node *h, *t, *next; Type data;

       while(true){
         h = Head; t = Tail;
1:       next = h->Next;
2:       if (next == NULL) return EMPTY;
3:       if (h == t) { CAS(&Tail,t,next); continue}
         data = next->Data;      // next behind dummy
4:       if (CAS(&Head,h,next)) break;  // new dummy
       }
       return data;
    }
```



Results

81

The **Module Checker Tests** of several test scenarios using Enqueue() and Dequeue() were running without failure.

Why is this interesting?

- We have here much more **complicated code**, with 2 or even 3 CAS operations instead of just 1 in push/pop
- In literature the algorithms of the MS-queue are shown with one **additional if-clause** (which is superfluous in my opinion, just an additional 'VALIDATE')

  The tests showed I was right. The algorithms do work correctly without that additional if-clause.

Results

82

Is such Module Checking Test Tool useful **for large systems**? (Robustness and Usability ?)

Mine is not – there are much more cases to be considered
– but **CHESS** probably is !

CHESS has been integrated into the test frameworks of many code bases inside Microsoft and is being used by testers on a daily basis. (Yes – by testers, not only by the authors of CHESS)

CHESS has found numerous previously unknown bugs in systems that had been stress tested for many months prior to being tested by CHESS.

CHESS works with Win-API, .NET and Singularity.

Hints for Testing Concurrent Software

83

When your multithreaded software is intended to run both on multi-processor and on single-processor machines:

(stress-) test it on a machine with the highest available number of processors (increase the likelihood of interferences)

It has shown that it is advantageous when the number of threads is a (small) multiple of the number of processors.

Be aware that your test program can mask potential negative interactions.

Stress testing with random delays is easy to accomplish and often shows good results (i.e. finds failures).

Hints for Writing Concurrent Software

84

Try to encapsulate concurrent interactions in a few well tested functions.

Concurrency mechanisms, such as our FIFO queue, often act as a conduit for moving objects from one thread to another.

Make the generation of the objects on one side and the further work with them on the other side **thread-safe**,

and treat the objects as immutable while in the queue.

References, Shortlist

85

Nir Shavit
Data Structures in the Multicore Age
Communications of the ACM, Vol. 54, No. 3, March 2011, pp. 76-84

ISTQB
Certified Tester, Foundation Level Syllabus, Version 2011

Madanlal Musuvathi, Shaz Qadeer, Thomas Ball
CHESS: A Systematic Testing Tool for Concurrent Software
Technical Report MSR-TR-2007-149,
Microsoft Research, Redmond, WA 98052

Sebastian Burckhardt, Madan Musuvathi, Shaz Qadeer
CHESS: Analysis and Testing of Concurrent Programs
Microsoft Research, Tutorial at PLDI 2009

CHESS homepage: http://research.microsoft.com/en-us/projects/chess/

References, Shortlist

86

Maged M. Michael, Michael L. Scott
Simple, Fast, and Practical Non-Blocking and Blocking Concurrent Queue Algorithms
Proceedings of the 15th Annual ACM Symposium on Principles of Distributed Computing (PODC '96), New York, USA, ACM (1996)
pp. 267-275

Maurice Helihy, J. Eliot B. Moss
Transactional Memory: Architectural Support for Lock-Free Data Structures
Proceedings of the 20th annual international symposium on computer architecture (ISCA '93), New York, USA, ACM (1993)
pp. 289-300

All the papers can be found as pdf-files in the internet.