Observations from Uppsala

- Home
- About Jakob Engblom and this blog
-

Dekker's Algorithm Does not Work, as Expected

January 7th, 2008 | Categories: multicore computer architecture, multicore software, programming | Tags: code example, Dekker's Algorithm, Embedded Systems Conference, freescale, Intel, mpc8641d, race condition

Sometimes it is very reassuring that certain things do not work when tested in practice, especially when you have been telling people that for a long time. In my talks about Debugging Multicore Systems at the Embedded Systems Conference Silicon Valley in 2006 and 2007, I had a fairly long discussion about relaxed or weak memory consistency models and their effect on parallel software when run on a truly concurrent machine. I used Dekker's Algorithm as an example of code that works just fine on a single-processor machine with a multitasking operating system, but that fails to work on a dual-processor machine. Over Christmas, I finally did a practical test of just how easy it was to make it fail in reality. Which turned out to showcase some interesting properties of various types and brands of hardware and software.

Now to the code.

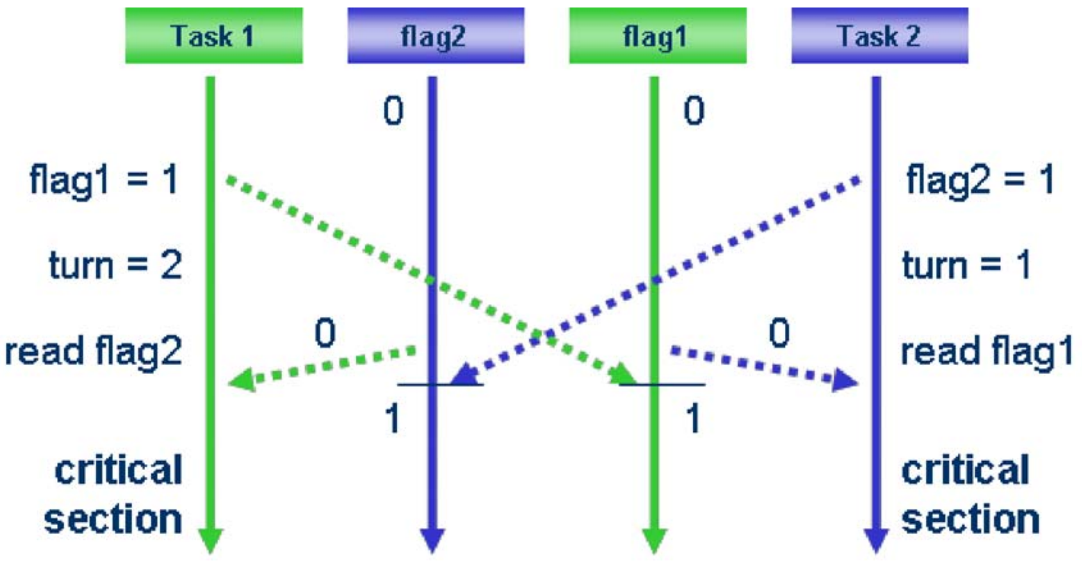
The core part of Dekker's Algorithm are two symmetrical pieces of code that access a set of shared variables in a way that makes it impossible for both codes to enter their critical section at the same time. As long as memory is sequentially consistent. Here is my implementation, warts and all:

```
static volatile int flag1 = 0;
static volatile int flag2 = 0;
static volatile int turn  = 1;
static volatile int gSharedCounter = 0;

void dekker1( ) {
        flag1 = 1;
        turn  = 2;
        while((flag2 ==  1) && (turn == 2)) ;
        // Critical section
        gSharedCounter++;
        // Let the other task run
        flag1 = 0;
}

void dekker2(void) {
        flag2 = 1;
        turn = 1;
        while((flag1 ==  1) && (turn == 1)) ;
        // critical section
        gSharedCounter++;
        // leave critical section
        flag2 = 0;
}
```

This code can fail on a machine with weak memory consistency since there is no constraint in most memory systems about the order in which the updates to "flag2″, "flag1″, and "turn" become visible to the other processor. In particular, there is no guarantee that the read from "flag2″ in dekker1 will happen after the write to "flag1″ and "turn" propagates to dekker2. Doing this argument symmetrically, you get something like the following sketch:



From this basic faulty code, I then use pthreads to create a parallel program. In the program, I loop many million of times in each thread trying to get into the critical section:

```
int gLoopCount;
void *task1(void *arg) {
        int i;
        printf("Starting task1n");
        for(i=gLoopCount;i>0;i--) {
                dekker1();
        }
}
void *task2(void *arg) {
        int i;
        printf("Starting task2n");
        for(i=gLoopCount;i>0;i--) {
                dekker2();
        }
}
```

If it happens that both enter the critical section at the same time, the construction of increasing a shared counter in dekker1 and dekker2 will likely result in a missed update to "gSharedCounter" as both threads read the same value, increment it, and then write the same value back (this is the kind of error mutual exclusion is supposed to protect against). Given this, by checking the value of gSharedCounter at the end of the program run, I can tell if any failures to lock happened. Note that this is likely conservative, since it is quite possible that one task manages to do the entire read-increment-write operation implicit in the ++ operation before the other task does. So the number of missed updates is actually a lower bound on the number of failed lockings.

So what happened when I tried this on real machines?

I must admit that I did not expect to see very many instances of errors, since I kind of assumed that modern hardware is so fast in communicating between cores that the window of opportunity to catch a bug would be pretty small. But it turned out to be quite frequent, and very variable across machines.

- On my Core 2 Duo T7600 laptop, with Windows Vista, I got on average one error in every 1.5 million locking attempts.
- On an older dual-processor Opteron 242 machine, I got on average one error every 15000 locking attempts. 1000 times more often than on the Core 2 Duo!
- On a Freescale MPC8641D dual-core machine, I got on average one error every 2000 locking attempts.
- On a range of single-core machines, not a single error was observed.

So the theory is validated. Always feels good to have proof in practice that I have been telling the truth at the ESC 😃 What else can we tell from the numbers? I think this actually demonstrates a few other theories in practice:

- Communication between cores on a single chip is much faster than between separate chips, and the longer latencies between chips makes Dekker more likely to fail. This is shown by the difference between the dual-processor and dual-core x86 systems.
- The PowerPC has a weaker memory consistency model by design than x86 systems, so the greater occurrence of locking failures there is also consistent with expectations.

Now all I need to find are a few more machines to test on. It is always fun to do microbenchmarking of real machines, as evidenced in my RTAS 2003 paper on branch predictors and their effect on WCET predictability.

If you want to try this yourself, the source code is attached to this post: dekker.c . Just compile it with "gcc -O2 -o dekker dekker.c -lpthread". It has been tested on Windows with Cygwin as well as various Linuxes. Run it with the argument 10000000 (10 million), which appears to give a good indication of the prevalence of errors without running for too long.

2 Like 1 Tweet 2 Share

Leave a comment | Trackback

1. Jakob
January 22nd, 2008 at 12:44
Reply | Quote | #1

   Update: I just tried this on a Freescale MPC8572E dual-core chip, and it has a far more aggressive memory system… I get locking failures every 5 to 20 attempts on that one. Very interesting and very unexpected.

2. foo bar
August 31st, 2009 at 23:42
Reply | Quote | #2

Good analysis.
I'm assuming to fix the program would require some architecture specific inline-asm, to create a memory barrier.

3. Johnny
August 15th, 2010 at 23:28
Reply | Quote | #3

Your testcase is faulty. Volatile cannot proect against operations like "++". It can only protect against load and store. Using of gSharedCounter++ as an indicator is misleading. Therefore, your result is wrong. Use a mutex(or some other mechanism) to protect against the gSharedCounter and try again.

4. Jakob
August 16th, 2010 at 07:07
Reply | Quote | #4

> **Johnny** :Your testcase is faulty. Volatile cannot proect against operations like "++". It can only protect against load and store. Using of gSharedCounter++ as an indicator is misleading. Therefore, your result is wrong. Use a mutex(or some other mechanism) to protect against the gSharedCounter and try again.

No, that is not a correct statement. Volatile ties a variable to memory and forces it to be loaded and stored each time it is used, and never stored in a register. That I do a "++" operation on it is irrelevant, doing "x=x+1″ would have done the same.

This code is interesting since it works on a single-processor machine, but not on a multiprocessor.

Using a mutex is silly, since Dekker's is a way to try to implement a mutex using user-level operations. But certainly, that would make things safe.

5. Jakob
August 31st, 2010 at 05:45
Reply | Quote | #5

The story of how Dekker came to invent this algorithm can be found in an interview with Edsger Dijkstra. See page 8 of the interview at http://www.cbi.umn.edu/oh/pdf.phtml?id=296 .

6. Sven Almgren
October 26th, 2011 at 15:49
Reply | Quote | #6

I wounder if using __sync_synchronize() would make this "correct"? Right before while().

turn = 2;
__sync_synchronize();

```
while((flag2 == 1) && (turn == 2)) ;

turn = 1;
__sync_synchronize();
while((flag1 == 1) && (turn == 1)) ;
```

or is it just luck that it works for me on an Intel Core i7? Have run it a few times with ./dekker 1000000000

/S

7. [Jakob](#)
October 29th, 2011 at 19:20
[Reply](#) | [Quote](#) | [#7](#)

at http://gcc.gnu.org/onlinedocs/gcc/Atomic-Builtins.html , __sync_synchronize is described as a "full memory barrier". I.e., all memory operations before the barrier will complete before moving on. With two barriers like this in place, the execution is essentially serialized and the code will be "correct".

8. liu
November 20th, 2011 at 16:40
[Reply](#) | [Quote](#) | [#8](#)

I got deadlock on my machine, even through I've bind the two threads on one cpu core using sched_setaffinity routing.

My machine:
Intel(R) Xeon(R) CPU E5520 @ 2.27GHz
8 cores

9. [Jakob](#)
November 27th, 2011 at 11:53
[Reply](#) | [Quote](#) | [#9](#)

Getting a deadlock is kind of unexpected. Tying each thread to a core should make the error occur more often, as we are not at the mercy of OS scheduling then. But it is strange that it deadlocks, as nothing in this code should be able to spin forever. I would suggest attaching a gdb to the process and intercepting it to see what it is doing. Is it spinning in user land or is it off in the kernel somewhere?

10. memon
March 21st, 2012 at 10:24
[Reply](#) | [Quote](#) | [#10](#)

Hi,Jakob, I tried your sample code and i got some problem that the code runs very slow. I modified the code a little to output "gSharedCounter" every 5 seconds, and here is the result:

```
./dekker 10000000
Starting task2
Starting task1
dekker 2: 8942980
dekker 1: 8942981
dekker 1: 8944243
dekker 2: 8944244
dekker 2: 8945532
dekker 1: 8945533
dekker 1: 8946799
dekker 2: 8946800
```

you can see the counter quickly gets to 8 million and then slow down…
i am new to this and i am still trying to find out why.
Due to some limitation, i use Ubuntu oneiric on virtualbox and set the processor number to 1. When i use 2 processors the code seems ok.
My host OS is Windows 7 with Intel Core i5 2520.

11. memon
March 21st, 2012 at 15:07
Reply | Quote | #11

Hi, Jakob, i have another question: is there any tool/method to debug/observe such problems? i believe that if some program fails due to "there is no constraint in most memory systems about the order in which the updates to "flag2?, "flag1?, and "turn" become visible to the other processor.", it is not easy to find the root cause.

12. Jakob
March 23rd, 2012 at 22:02
Reply | Quote | #12

> **memon :**
> Due to some limitation, i use Ubuntu oneiric on virtualbox and set the processor number to 1. When i use 2 processors the code seems ok.

Two things. First, running in a VM could have interesting effects, depends on how the VM works. A typical x86 host VM probably does not affect the outcome from what I know of how they work.

Second, if you use 1 virtual processor, the program will not show its effect. There is no way to get a missed lock on a single processor as this requires true concurrency — that is the whole point here. Dekker's works nicely on an interleaved execution on single core, but fails when subject to true concurrent execution on a truly parallel machine.

13. Jakob
March 23rd, 2012 at 22:03
Reply | Quote | #13

> **memon** :Hi, Jakob, i have another question: is there any tool/method to debug/observe such problems? i believe that if some program fails due to "there is no constraint in most memory systems about the order in which the updates to "flag2?, "flag1?, and "turn" become visible to the other processor.", it is not easy to find the root cause.

Exactly

Finding totally missing locks around shared data is detective work. There might be some static analysis tools that could warn that you are accessing data from multiple threads without using known lock operations.

14. memon
April 1st, 2012 at 05:09
Reply | Quote | #14

> **Jakob** :
>
> > **memon** :
> > Due to some limitation, i use Ubuntu oneiric on virtualbox and set the processor number to 1. When i use 2 processors the code seems ok.
>
> Two things. First, running in a VM could have interesting effects, depends on how the VM works. A typical x86 host VM probably does not affect the outcome from what I know of how they work.
> Second, if you use 1 virtual processor, the program will not show its effect. There is no way to get a missed lock on a single processor as this requires true concurrency — that is the whole point here. Dekker's works nicely on an interleaved execution on single core, but fails when subject to true concurrent execution on a truly parallel machine.

Yes, for multiprocessor, i see the inconsistency there…
./dekker 10000000
Output the cpu affinity
cpu=0 is set
cpu=1 is set
cpu=2 is unset
Starting task1
Starting task2
Both threads terminated
[-] Dekker did not work, sum 19999997 rather than 20000000.
3 missed updates due to memory consistency races.
it is quite impressive to explain the ideas in your article.

For the problem that running dekker.c with one processor, i tried to find a PC with single-core cpu other than using VM, but i failed… my oldest friend has a dual core cpu… Then i tried to run dekker.c using only one core by setting the cpu affinity. It seems i got the same result as in the VM:
./dekker 1000000
Output cpu affinity
cpu=0 is unset
cpu=1 is set

```
cpu=2 is unset
Starting task2
Starting task1
dekker2 gSharedCounter:422491.
dekker1 gSharedCounter:422492.
dekker1 gSharedCounter:423240.
dekker2 gSharedCounter:423241.
…
```

For 5 seconds, gSharedCounter only increases from 422492 to 423240. I think there might be some problem, but i still could not figure out why…

Now i run all the test on real machine with Pentium(R) Dual-Core CPU E5800 @ 3.20GHz and ubuntu 11.10.

15. Jakob
April 1st, 2012 at 20:53
Reply | Quote | #15

> For 5 seconds, gSharedCounter only increases from 422492 to 423240. I think there might be some problem, but i still could not figure out why…

That is very interesting. My guess is that the OS gets into some chaotic scheduler state where the thread that holds the "lock" gets interrupted just after it grabs it, and then the other thread sits in a spin loop for its entire time slice… basically, a "priority inversion" where the thread that should run to get progress is blocked, and the other thread just spins and spins and spins eating lots of CPU time to no avail.

Actually, you would almost expect to get that effect a few times in any sufficiently long single-processor execution of this program.

Name (required)
E-Mail (will not be published) (required)
Website

Subscribe to comments feed
Submit Comment
Theme Styles :

- Search for: Search

- **Recent Posts**

  - Wind River Blog: Simics and Flying Piggies
  - Dragons can be Useful – when AT Models Make Sense
  - Logging (Some More Thoughts)
  - Reverse Execution History Updates
  - Wind River Blog: Exposing OS Kernel Races with Landslide
  - S4D 2012 – Notes
  - SiCS Multicore Day 2012
  - Paper & Talk at S4D 2012: Reverse Debug
  - Negative Results
  - Speaking at Embedded Conference Scandinavia
  - Wind River Blog: Testing Multicore Scaling with a Simics QSP
  - Speaking at SiCS Multicore Day 2012
  - Buying High Technology
  - "Eagle" Cycle-Accurate Simulator Anno 1979
  - Some Fun Cache Results from Carbon

- **Categories**

  - appearances (30)
  - articles (21)
  - blogging (10)
  - books (6)
  - business issues (29)
  - computer architecture (33)
  - conferences (34)
  - EDA (50)
    - ESL (35)
  - embedded (77)
    - embedded software (57)
    - embedded systeme (49)
  - general research (5)
  - history (31)
    - general history (6)
    - history of computing (26)
  - off-topic (90)
    - biking (5)
    - board games (1)
    - computer games (2)
    - desktop software (35)
    - food and drink (1)
    - funny (11)
    - gadgets (24)
    - Politics (3)
    - popular culture (5)
    - trains (4)
    - transportation (9)
    - travel (10)
    - websites (3)

- parallel computing (90)
  - multicore computer architecture (50)
  - multicore debug (21)
  - multicore software (64)
- programming (104)
- review (8)
- security (19)
- teaching (7)
- testing (9)
- uncategorized (12)
- virtual things (123)
  - computer simulation technology (67)
  - virtual machines (17)
  - virtual platforms (93)
  - virtualization (14)
- Wind River Blog (36)

- **Tags**

  ARM blog commentary Cadence Checkpointing clock-cycle models Communications of the ACM computer architecture conference cycle accuracy debugging DML Domain-specific languages embedded freescale G900 heterogeneous homogeneous IBM Intel iPod lego linux mobile phones multicore off-topic office 2007 operating systems p4080 podcast commentary power architecture rant research reverse debugging reverse execution S4D SiCS Multicore days Simics simulation software tools Sun SystemC video virtualization Vista Windows

- **1**

  - F-Secure Blog

- **Blogs and news**

  - Andras Vajda's blog (on multicore)
  - Embedded in Academia (John Regehr)
  - Grant Martin
  - Jack Ganssle
  - My Wind River Blog
  - Security Now podcast
  - Secworks (Joachim Strömbergson)
  - Simon Kågström
  - Synopsys View from the Top
  - Worse Than Failure

- **Archives**

  - November 2012 (2)
  - October 2012 (1)

- September 2012 (6)
- August 2012 (4)
- July 2012 (4)
- June 2012 (3)
- May 2012 (4)
- April 2012 (2)
- March 2012 (3)
- February 2012 (1)
- January 2012 (6)
- December 2011 (2)
- November 2011 (3)
- October 2011 (4)
- September 2011 (5)
- August 2011 (4)
- July 2011 (3)
- June 2011 (4)
- May 2011 (7)
- April 2011 (1)
- March 2011 (3)
- February 2011 (5)
- January 2011 (1)
- December 2010 (4)
- November 2010 (3)
- October 2010 (5)
- September 2010 (5)
- August 2010 (5)
- July 2010 (6)
- June 2010 (5)
- May 2010 (3)
- April 2010 (4)
- March 2010 (3)
- February 2010 (4)
- January 2010 (7)
- December 2009 (6)
- November 2009 (6)
- October 2009 (7)
- September 2009 (6)
- August 2009 (7)
- July 2009 (11)
- June 2009 (5)
- May 2009 (10)
- April 2009 (7)
- March 2009 (8)
- February 2009 (9)
- January 2009 (12)
- December 2008 (8)
- November 2008 (9)
- October 2008 (9)
- September 2008 (10)

- August 2008 (13)
- July 2008 (12)
- June 2008 (8)
- May 2008 (9)
- April 2008 (10)
- March 2008 (7)
- February 2008 (8)
- January 2008 (5)
- December 2007 (5)
- November 2007 (7)
- October 2007 (7)
- September 2007 (12)
- August 2007 (9)
- July 2007 (2)

« Multithreading Game AI Brilliant Virtualization Comic »
TOP

- Log in