A Study of OS Schedulers for Concurrency Bugs

Faisal Khan
University of Wisconsin-Madison
faisal@cs.wisc.edu

Abstract

Concurrency related bugs are hard to detect and even harder to reproduce due to their time dependent nature. One big factor in non-determinism of multi-threaded applications is process scheduler of an operating system. In this work, we presented results of our study to understand the behavior of some of the selected concurrency bugs under different OS schedulers. The generalization of this study to a larger set of bugs caused by concurrent execution of a program is hard and may not be very precise in its nature. Still, the consistent nature of our results for a chosen set of separate bugs on multiple schedulers suggests that shorter time-slices and uniform inter-mixing of the execution units of a program by a scheduler can lead to the early manifestation of concurrency bugs in a software.

1. Introduction

Application programmers have no control over how an operating system schedule multiple execution units within a program and occasionally the order intended by the programmer is violated. According to a comprehensive study done on the characterization of concurrency related bugs a major cause of such bugs is either due to atomicity violation or order violation[1]. Additionally, concurrent bugs are often result of some rare and complex inter-leavings that can even skip the usual rigorous testing process. One main source of such non-determinism is process scheduler in operating system.

A process scheduler in OS is responsible for giving an illusion of simultaneous execution of multiple process by switching between them fairly quickly. A set of policies generally govern the behavior of these schedulers. In O(1) scheduler [20], separate queues are maintained to keep track of two sets of processes, one that has still time ticks left in their time quantum and the other set of processes that have exhausted their current time slice. Processes in each of these queues are given their share of CPU based on their priority and their current average sleep time. An interactive process (one that spend more time doing I/O) is given a higher time slice under O(1) scheduler. The CFS (Completely Fair Scheduler)[4] is a recent addition to Linux and has adapted quite radical changes then its previous counterpart. The CFS scheduler tries to balances CPU usage among competing processes based on how much an individual process is behind in its fair share of CPU.

Although, a strong link between OS scheduling parameters and concurrency bugs is not that well established but still we ran into instances where a certain system is either more or less favorable in reproducing a given bug [5]. We understand that establishment of such a link requires a great deal of effort. Specially, a very comprehensive study involving identifying and studying a large set of concurrency bugs will be required. Our work can be considered as an initial step to characterize behavior of OS schedulers with respect to concurrency bugs.

We choose three different programs each containing a known bug caused by concurrent execution. One of our selected bug is from a 'real world' bug report related to version 4.0.23 of MySQL server running on some variants of Linux kernel. The other two bugs are included as a benchmark to generalize the behavior of our chosen schedulers. The versions of Linux kernel we choose for this study are 2.6.11 [21] and 2.6.23 [22]. Both of these have very different schedulers, 2.6.11 is based on O(1) priority schedulers while 2.6.23 contains implementation of CFS scheduler. The parameters that we explicitly studied for each of our schedulers include: ability to reproduce a given bug, length of time-slices for different execution units (threads) allocated by each scheduler, general pattern of eviction points for each thread in a program e.g by looking at stack information when a context switch happens. A general description of these bugs is presented in section 2 followed by results and analysis of our experiments in section 4.

Another contribution of this work is to come up with a set of a tools and a design that could allow one to study the interaction between different components in Linux system e.g user process and scheduler in a way that is non-intrusive to a live system. This was particularly hard given that Linux lacks a powerful tool [18][19] that could provide consolidated view of the system. We formally stated our contributions in next subsection. Section 3 explains our methodology and tools used in detail. The related work is given in section 5 and finally we concluded our work in section 6.

Query	Number of Operations
INSERT DELAYED INTO id0(X, Y, Z)	500,000
INSERT DELAYED INTO id1(A, B, C)	500,000
SHOW PROESSLIST; FLUSH TABLE WITH READ LOCK; UNLOCK TABLE	100,000

Table 1. Concurrent execution of these queries can cause a deadlock for MySQL server's version 4.0.23. The second column shows how many times these queries were repeated. The alphabets X, Y, Z and A, B, C represent a randomly generated integer.

1.1 Contributions

We consider that this work has made following contributions:

1. An experiment driven study of different parameters of at least two separate schedulers that could possibly effect the manifestation of a concurrency bug.
2. A practical analysis of OS scheduling policies in specific context of concurrency bug. It is particularly significant as understanding the theoretical model of policies that are entirely based on heuristics is generally challenging and may not be able to fully capture the behavior of an implementation.
3. A set of tools and a design to take an insight look into a running system.

1.2 Deviation from Proposal

We originally proposed to study operating system schedulers under multi-core systems but during our initial testings the case for multi-core systems became less interesting. In all cases for our selected bugs, multi-core system easily lead to desired incorrect result in presence of bug. Also, we dropped the idea of deterministic scheduling, as mentioned in proposal, due to lack of time.

2. Concurrency Bugs

We selected one real world bug related to MySQL server and two small benchmark programs containing race conditions. The purpose of such benchmarks was to to observe presence of any general pattern that could favor early or delayed manifestation of bugs in our chosen schedulers.

2.1 Concurrency Bug-I: MySQL Server

MySQL server is a popular open source database server by Sun Microsoystems [6]. One relatively older version (4.0.23) of MySQL server has a concurrency bug that can lead to deadlock when given a set of specific queries. It is also known that manifestation of this bug is OS dependent e.g it is easy to reproduce it under 2.4.x kernel and not on 2.6.9 kernel [5] . The actual cause of bug is some complex interleaving of concurrent execution of three set of queries in table 1.

The server handling of these queries involve creating different kernel threads. Each delayed insert query is handled by two threads, one handling user queries and replying instantly to user due to 'DELAYED' keyword and one delay handler thread that is actually inserting the rows in database. Both these thread share a mutex lock to co-ordinate their activities. The flush query is handled by a single thread and its job is to close all open tables and kill all active delay handlers. To ensure that delay handlers don't exit while flush thread is operating on them latter acquires the same mutex shared by delay and insert thread. The cause of deadlock is a circular wait where flush query thread is waiting for one of the delayed handler's mutex while holding a global read lock on database tables. Whereas, the same delayed handler is waiting for acquiring this global read lock while holding its mutex. The wait for graph for this situation is shown is figure 1. We will refer to these five threads as 'interesting' later in this text.

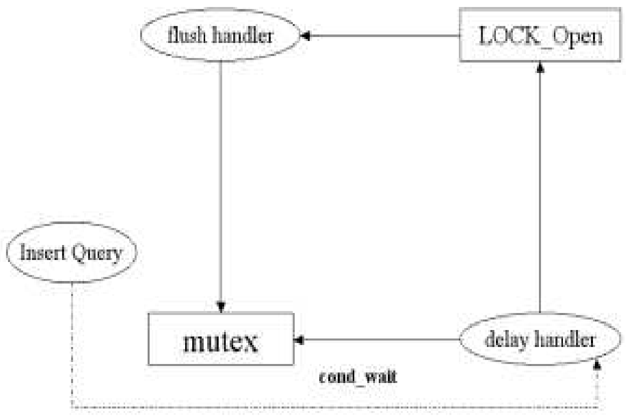


Figure 1. Wait-for graph showing deadlock state of MySQL server version 4.0.23 on Linux kernel version 2.6.23

Parameter	Source of Information
Interleaving	Looking at the user stack while in the kernel's schedule() method
Time Slice	Recording time for processes entry and exit in the context switch method

Table 2. Parameters related to a running program and how they are extracted.

```
#include <pthread.h>
//Global counter
long counter = 0;
void * worker(void *arg) {
   int x;
   for (x=0; x<ITERATION_PER_THREAD; x++)
      counter++;
}
int main(int argc, char *argv[]) {
   pthread_t p[THREADS];

   int i;
   for (i=0; i<THREADS; i++)
       pthread_create(&p[i],  NULL,
           worker, NULL);

   for (i=0; i<THREADS; i++)
       pthread_join(p[i], NULL);

   return 0;
}
```

Figure 2. Non-synchronized access to a shared variable.

2.2 Concurrency Bug-II

Our first benchmark program is related to presence of a race condition where multiple threads are trying to increment a shared variable. The absence of any synchronization primitive between these threads can produce wrong result. The code snippet for this bug is given in figure 2.

2.3 Concurrency Bug-III

The second of our benchmark program is a typical example of order violation in acquiring the lock that can lead to a deadlock depending on the interleaving chosen by scheduler. The code snippet for this bug is given in figure 3.

3. Methodology

We executed all these three programs under different version of Linux kernel that were known of having different schedulers. These executions were repeated multiple time to gain some confidence about our results. Once discovering at-least one interesting case (i.e MySQL server bug), we moved on to instrument the Linux kernel to gather more insight into each of our chosen scheduler e.g interleaving - how concurrent parts are executed, time slice - CPU time given to a process before its evicted by some other process. Now, It is worth mentioning here that the parameters affecting the chosen interleaving can be enormous e.g different type of interrupts, presence of higher priority processes in run queue etc. So, to keep the scope of this project within some manageable constraints we only focused on the time slice information and interleaving. Table 2 summarize how these two pieces of in-

```
#include <pthread.h>

long counter = 0;

pthread_mutex_t lockA;
pthread_mutex_t lockB;
void *  worker1(void *arg) {
 int x;
 for (x=0; x<ITERATIONS_PER_THREAD; x++) {

   pthread_mutex_lock(&lockA);
   pthread_mutex_lock(&lockB);

   counter++;

   pthread_mutex_unlock(&lockB);
   pthread_mutex_unlock(&lockA);
 }
}

void *  worker2(void *arg) {
 int x;
 for (x=0; x<ITERATIONS_PER_THREAD; x++) {

   pthread_mutex_lock(&lockB);
   pthread_mutex_lock(&lockA);

   counter++;

   pthread_mutex_unlock(&lockA);
   pthread_mutex_unlock(&lockB);
 }
}

int main(int argc, char *argv[]) {
  pthread_t p[2];

  pthread_create(&p[0],  NULL,
                   worker1, NULL);
  pthread_create(&p[1],  NULL,
                   worker2, NULL);

  pthread_join(p[0], NULL);
  pthread_join(p[1], NULL);

  return 0;
}
```

Figure 3. Reverse order locking

formation for each our test subjects were extracted from a live system.

3.1 Experimental Setup

One of the challenges faced by us was to come up with a way to gather information shown in table x with minimum overheads on system as to not effect our results. Unfortunately, Linux still lacks a powerful analysis tools (e.g like D-Trace[18]) that could provide a consolidated view of a running system. We overcome this hurdle by building a custom solution based on off the shelf components. Figure 4 shows a schematic follow of the overall layout of our solution. Below we describe each of the individual components mentioned in the figure followed by few words on flow of the information in this setup.

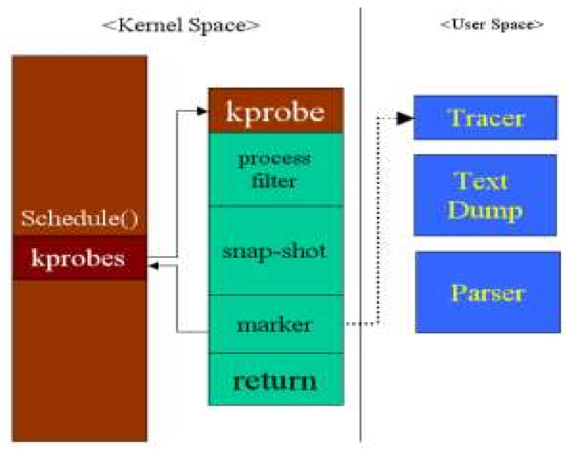


Figure 4. Tracing system setup

3.1.1 KProbes

Kprobes[7] is a way to insert breakpoint at a running kernel and gather information in non-disruptive way. Kprobes achieves this by dynamically re-writing breakpoint instruction at desired location and pass control to our code. Using this we can look at different data structures of kernel before restoring normal execution of kernel. For example, by placing kprobe module at 'schedule()' method we can known about the current processing that is currently running etc.

3.1.2 LTT trace markers

LTT or Linux Tracing Toolkit[8] is capable of handling a large amount of debugging events in a non-intrusive fashion. This toolkit is available as a set of patches for different version of kernel. It uses specialized markers, that can be inserted anywhere in the kernel including user processes, to produce events that can later be read by user level utilities. There are two separate versions, one for kernel version 2.6.11 [9] and lower and other for version 2.6.12 [10] and upwards. Both these version were separately modified by us to include extra information such process stack.

3.1.3 Stack Snapshot

The stack information is gathered by reading the user memory page currently mapped to the virtual page pointed by stack pointer at time of context switch.

3.1.4 Parser

A custom parser was written to analyze the LTT event information for obtaining results given in next section.

3.1.5 Information Flow

A brief flow of information is given here. A kprobe module is inserted at the kernel's 'context_switch()' method to read the process descriptor of current process and the one which is going to replace the current one. This information along with some 'useful' entries from user stack pages are passed to LTT. LTT's user level tools read these events from the running system and record the event information to a log file. Later we parse these log files for our analysis.

4. Experimental Results and Analysis

We conducted different sets of experiments for each of the bug given in section x. In this section we present different set of results for each of these bugs including reproducing these bugs under separate schedulers, studying time-slice information etc.

4.1 MySQL Server

In the next few subsections we give details of different set of tests that we conducted to understand the behavior of MySQL bug on two selected platforms 2.6.11 and 2.6.23 that are known to have different schedulers as stated before.

4.1.1 Bug Manifestation

The table 2 shows the results of running the problematic queries on different Linux versions along with occurrence of deadlock. The first column shows the version of kernel that we used. All of these kernel versions were running on a single CPU machine except for the one which have 'smp' in their name. The 'smp' versions were running on a multi-core configuration of the machine.

Kernel version	Scheduler	Deadlock Occurrence
2.6.11	O(1)Scheduler	Almost None
2.6.11-smp	O(1)	Always
2.6.23	CFS	Always
2.6.24-smp	CFS	Always
2.4.x	O(N) Scheduler	Always

Table 3. Summary of results

We ran queries mentioned in section 2 multiple times and result was surprisingly very consistent. Specially on 2.6.11 and 2.6.23 kernel version we conducted these tests couple dozen times with consistent results. There was only one instance on 2.6.11 where we saw occurrence of a deadlock.

For multi-core configuration of these kernels it was apparent that we are not going to get anything interesting as far as this bug is concerned. So, we decided to focus our attention on 2.6.11 and 2.6.23 kernel. More formally stating, we picked these two kernels because:

1. Both kernel versions are known to have different schedulers.
2. It is extremely hard to reproduce the MySQL server deadlock on 2.6.11, whereas, running on 2.6.23 always leads to a deadlock.
3. CFS (Completely Fair Scheduler) is a recent addition to Linux kernel. It would be interesting to study its behavior for concurrency bugs.

4.1.2 Interleaving-I

We looked at each 'interesting' thread's stack for multiple execution of this bug to generalize the pattern of successful and non-successful interleaving. A successful interleaving pattern can be defined as the one when the normal execution continues to produce desired results. On the other hand, a non-successful interleaving will be the one which leads to deadlock. In other word a non-successful interleaving exposes the bug. The table 4 shows the sequence of context switches that lead MySQL server to a deadlock in multiple runs of this experiment. The third column in the table shows the state of the program just before a context switch. This state information was extracted by looking at the last user method at the top of the the threads' stack at time of context switch.

Seq	Thread name	Stack Overview
1	Delay thread	Waiting for insert thread
2	Flush thread	Executed pthread_lock on delay handler's mutex without acquiring it.
3	Insert thread	Waiting for delay handlers
4	Delay thread	Blocks on global read lock after acquiring its own mutex.
5	Flush thread	Completes pthread_lock statement and blocks as mutex was acquired by delay handler in previous context switch (seq 4).

Table 4. Description of interleaving that leads to dead of MySQL server on 2.6.23 kernel.

Now, for the successful pattern of interleaving we observed that flush thread gets evicted at the point when it is waiting for inputs from the client process or trying to write output for the results of 'show processlist' of the user query. Whereas, delay and insert threads generally become inactive when one is waiting for other using separate condition variables. Our guess at what might be the culprit in 2.6.23 that unlike in 2.6.11 caused deadlock is shorter time-slice. The reason for reaching at this preliminary conclusion was the difference between two kind of interleaving pattern we noticed in these two systems and nature of CFS scheduler that is used in 2.6.23. Under 2.6.11 a task doing I/O is favored in terms of getting higher time slice then compute intensive task. This implies that both flush and delay threads must have gotten a bigger time-slice on 2.6.11, thus minimizing their chances of being evicted 'prematurely'. In later subsections we conducted more tests to confirm this hypothesis.

4.1.3 Measure of Time-slice

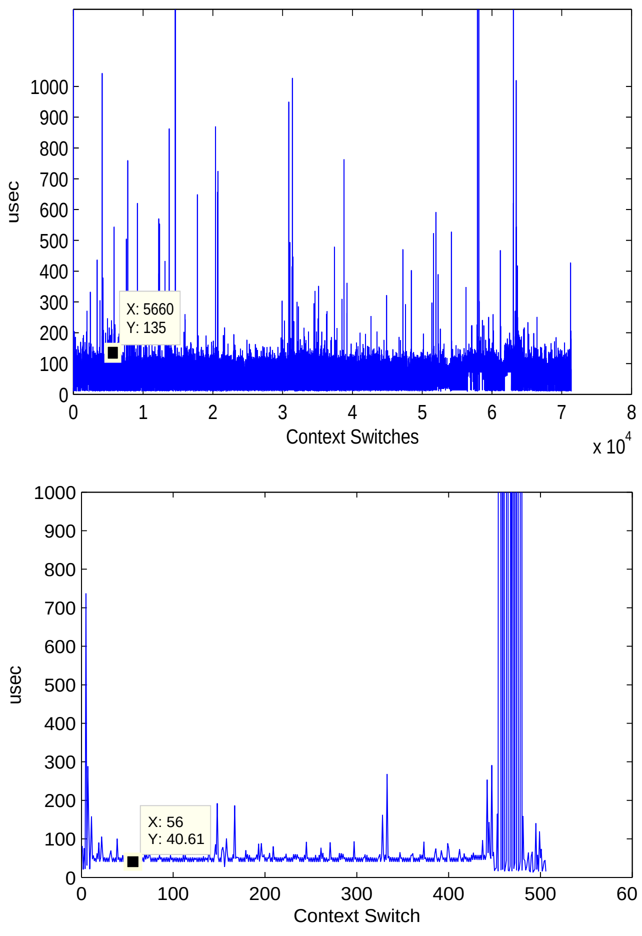


Figure 5. Time-slices for insert threads on 2.6.11 (top) and 2.6.23 (bottom) schedulers

As we suggested in previous section that smaller time-slice in CFS might be one of the causes for manifestation of MySQL bug on 2.6.23, we looked at time-slices against each context switch for our 'interesting' threads under these two schedulers. The plots for insert, flush and delay threads is given in figure 5, 6 and 7 respectively. The apparent difference between these threads on 2.6.11 and 2.6.23 is the length of time-slice. The CFS starts a thread by giving it a relatively higher time-slice but then suddenly drops it in an effort to balance off the CPU usage among all child of a given thread.

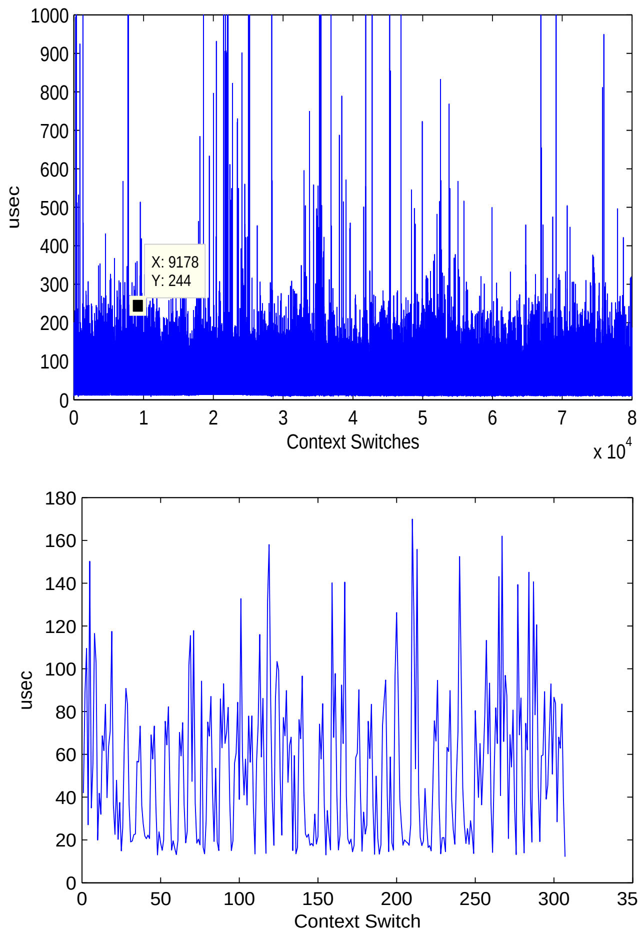


Figure 6. Time-slices for flush threads on 2.6.11 (top) and 2.6.23 (bottom) schedulers

4.1.4 Interleaving - II

To see how uniform a share of CPU was allocated by these scheduler among 'interesting' threads, we looked at the the accumulative time given to a specific 'interesting' thread aggregated over multiple context switches before any one of the other 'interesting' thread is scheduled. The rational behind this analysis was to see how much uniformly time slices are distributed among our 'interesting' threads. The result for this analysis is given in figure 8, 9 and 10. All these plots are based on the same data that we studies for time-slice information.

As it was expected, time-slice distributions on 2.6.11 is not as uniform as under 2.6.23. This indicates that CFS scheduler tends to given each thread fair amount of chance

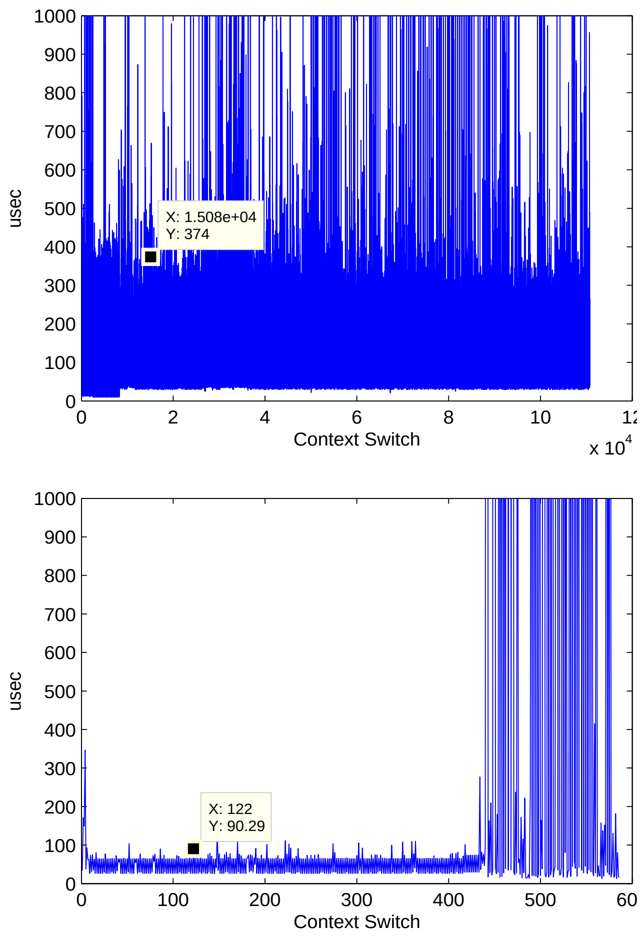


Figure 7. Time-slices for delay threads on 2.6.11 (top) and 2.6.23 (bottom) schedulers

in getting the CPU by well mixing the execution units. This can explain some of the difference in behavior of these two schedulers.

4.1.5 Environmental Changes

We tried two different kind of environmental changes to influence the behavior of these bugs. On 2.6.11 kernel, we ran the MySQL server with lowest possible priority using 'nice' utility. This resulted in producing deadlock for each test runs of these queries. Additionally, we tuned the CFS scheduler e.g by changing sched_wakeup_granularity_ns. The sched_wakeup_granularity_ns governs how much unfairness is allowed when balancing the CPU share among processes. This along with many other parameters defines a time-slice in CFS. This change avoided the deadlock but resulted in lower throughput of the MySQL server. We still need to do more testing to present any useful result for this change.

The time-slice information for running MySQL server with lowest possible priority for each of the insert , flush and

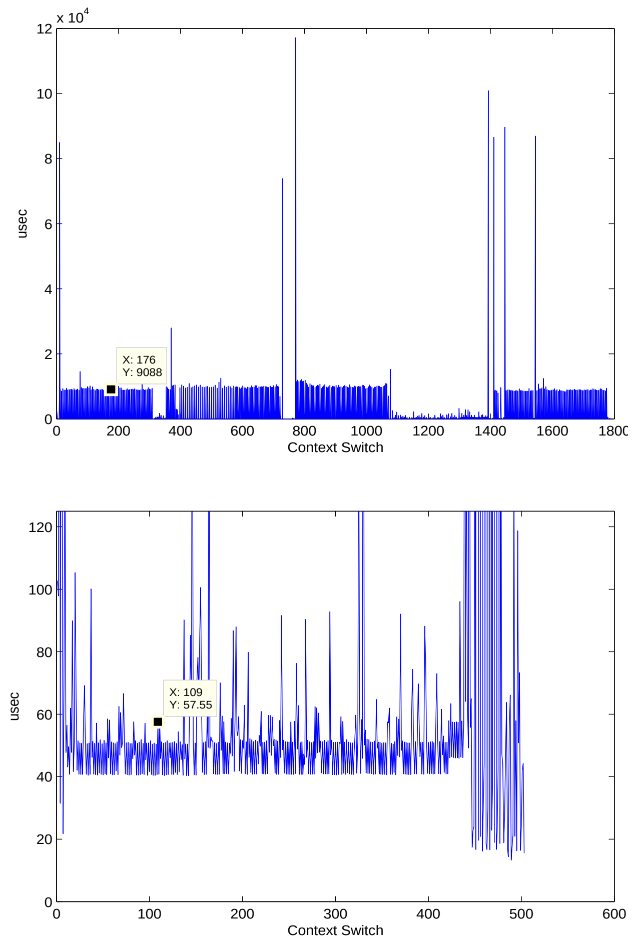


Figure 8. Time-slice distribution for one of the insert threads on 2.6.11 (top) and 2.6.23 (bottom) schedulers

delay thread is given in figure 11, 12 and 13 respectively. These results are very interesting and some what correspond to our earlier conclusion of having uniform distribution of time-slice that should have exposed the bug.

4.2 Concurrency Bug-II

Kernel Version	Threads	Iterations	Correct Results
2.6.11	2	1000000	All
2.6.11	100	100000	All
2.6.11	200	1000000	All
2.6.23	2	1000000	None
2.6.23	100	100000	None
2.6.23	200	1000000	None

Table 5. Results of execution of concurrency bug given in section 2.2 on different kernel versions.

The result of running one of the two benchmarks programs is given in Table 5. The middle two columns state different combination of threads and per thread iterations for

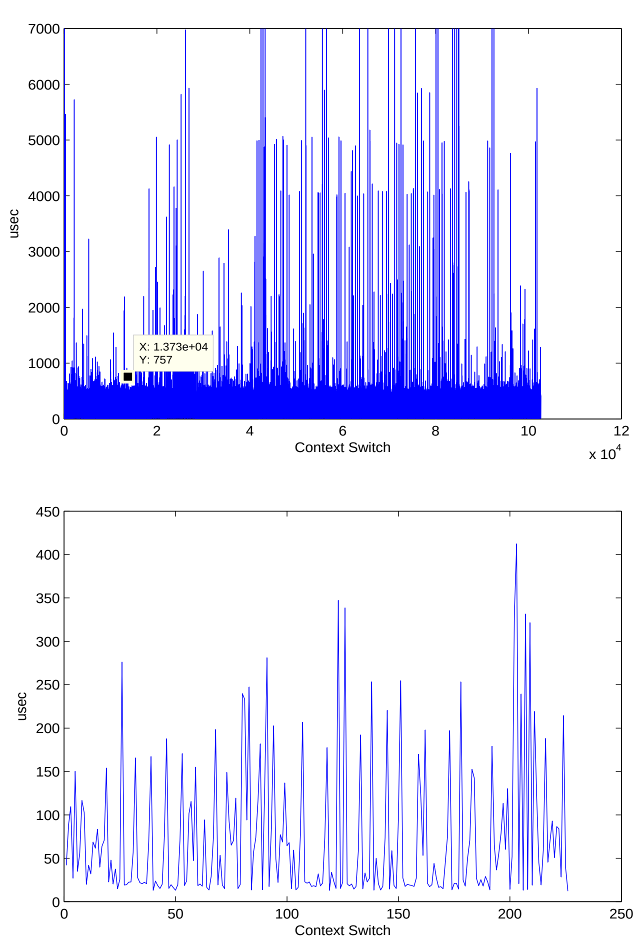


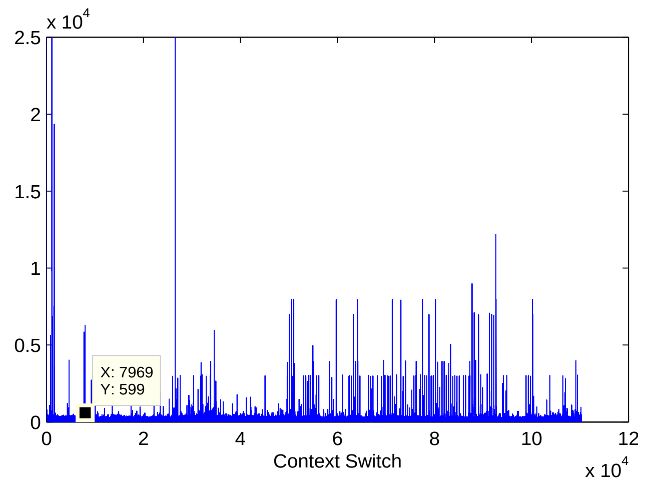
Figure 9. Time slice distribution for flush threads on 2.6.11 (top) and 2.6.23 (bottom) schedulers

Kernel Version	No. of Iterations	Deadlocked
2.6.11	10000	0%
2.6.11	100000	30%
2.6.11	1000000	50%
2.6.23	10000	0%
2.6.23	100000	90%
2.6.23	1000000	100%

Table 6. Results of execution of concurrency bug given in section 2.3 on different kernel versions.

incrementing the counter. The last column shows how many times we got the expected results for the global 'counter' variable. If all set of execution for a given number of threads and iterations were correct (or incorrect) we just reported 'All' (or 'None') otherwise a fraction of successful results is reported.

It is quite evident from these results that there is something particularly interesting about CFS scheduler that is causing earlier manifestation of two very different bugs. In



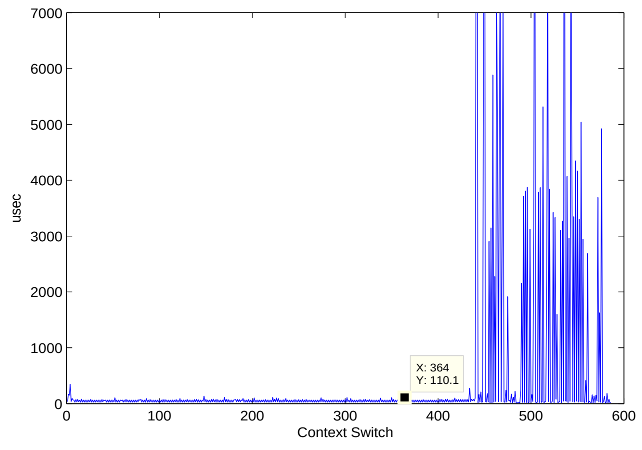


Figure 10. Time slice distribution for delay threads on 2.6.11 (top) and 2.6.23 (bottom) schedulers

the next subsection we looked at how the time-slices for a two thread case differ among 2.6.11 scheduler and CFS.

4.2.1 Time-slice

We measured time slices at each context switch in similar fashion as with MySQL server bug. The result for two thread case for 1 million iterations per each thread is given in figure 14. Surprisingly, the pattern is consistent with our earlier assumption that CFS's use of shorter time-slice, in an effort to balance the computing between all the processes in system, can be seen as a cause of easier manifestation of these bugs.

4.3 Concurrency Bug-III

The result of running our second benchmark program is given in Table 6. The middle column state iterations per thread for incrementing the shared counter. The last column shows percentage of times that execution of this sample program resulted in a deadlock on a given kernel version. We didn't do any further analysis on this bug, but, the results

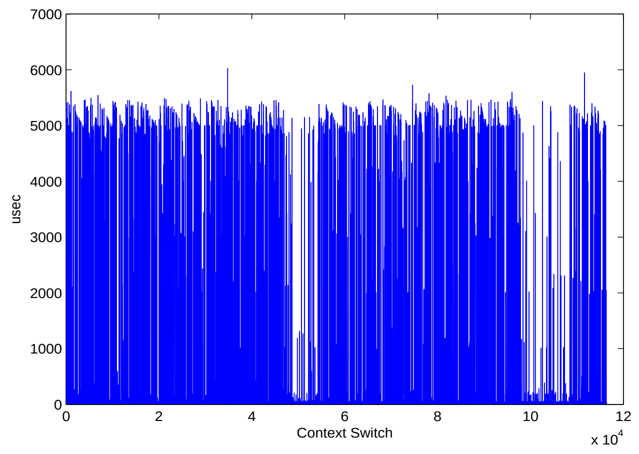


Figure 11. Time slice for insert thread under with priority MySQL server under 2.6.11

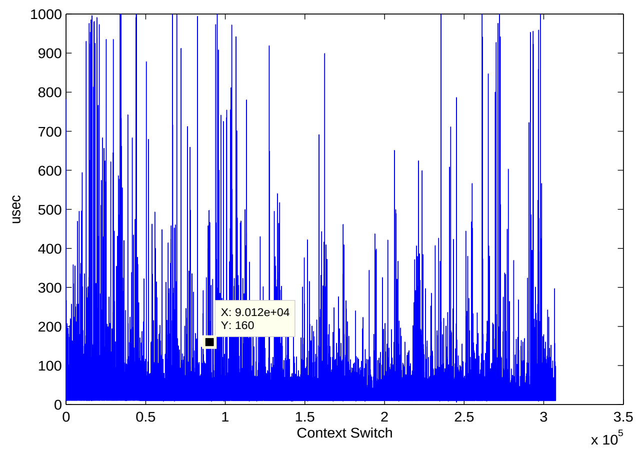


Figure 12. Time slice for flush thread with low priorty MySQL server under 2.6.11

supports, at least in part, our hypothesis that CFS favor earlier exposure of concurrency bugs.

5. Related work

The tools and techniques for detection and prevention of bugs in general and concurrency bugs in particular are topic of lot of recent work including [15, 17, 14, 3, 2]. However, we are unaware of any work that is a direct study of schedulers for relating them to concurrency bugs. One reason for this is probably because generalization of a heuristic based systems is very hard if not impossible.

The Rx[16] tool developed by Feng Quin et. al has included scheduling parameters as part of environmental changes necessary to avoid concurrency bugs. Our study also confirms their heuristic that certain environment changes can help to avoid certain interleaving thus reducing the chances

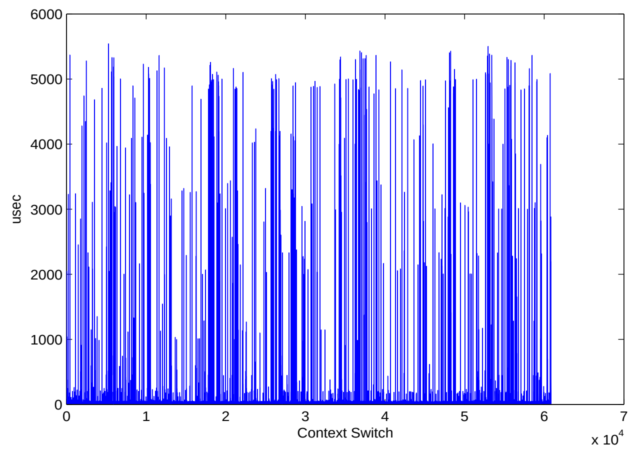


Figure 13. Time slice for delay handler thread with low priority MySQ server under 2.6.11

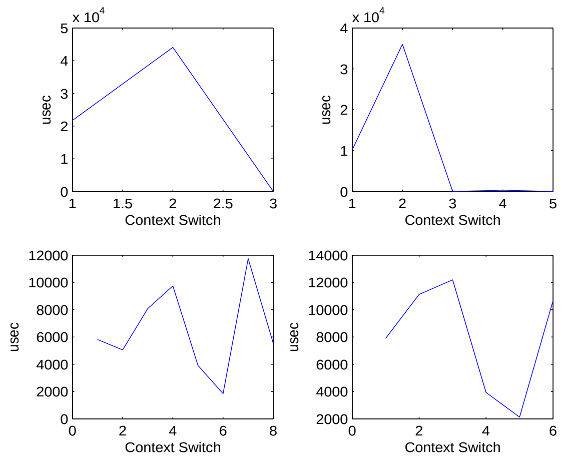


Figure 14. (a) top left: thread 1 running under 2.6.11 (b) top right: thread 2 running under 2.6.11. (c) bottom left: thread 1 running under 2.6.23. (d) bottom right: thread 2 running under 2.6.23

of occurrence of a failure in presence of a concurrency bug. But, it may have some significant side effects as exemplified in our own study.

A characterization of pattern of concurrency bugs is done by [1] and [11]. However, our work differ from them as they tried to study pattern of bugs based on the mistakes in program's source code that can cause a concurrent program to fail. Whereas, we tried to characterize schedulers in certain way to help identify policies that could help early exposure of such mistakes in source code.

One common approach prevalent in literature to detect concurrency bugs is to capture the non-determinism in application by replying the execution of program by forcing same interleaving [12], controlling scheduler's noise [13] or exploring all possible interleaving space [14]. All these efforts indicate the importance of a study as such of ours to establish a relation between OS schedulers and concurrency bugs.

6. Conclusion

We studied different concurrency related bugs, including one from a real world application, under separate schedulers. The main goal was to find any relation between manifestation of bug and schedulers. Although, a generalization of scheduler behavior will require a broader study involving investigation of multiple bugs, but, we were able to drew some handsome conclusion by doing a comprehensive analysis of multiple parameters involved in influencing the scheduling policies e.g time slice, accumulative time, interleaving etc. Our results suggests that under CFS scheduler, chances of manifestation of a concurrency bugs are higher then O(1) scheduler. The main cause of these bugs to easily show up on CFS can be attributed to shorter time-slice and uniform intermixing of threads. A future study analyzing difference between CFS and O(1) scheduler can shed some more light on the causes and effects of these schedulers on occurrence or avoidance of bugs related to multi-threaded environment.

References

[1] S. Lu, S. Park, E. Seo, and Y. Zhou. Learning from mis- takes a comprehensive study on real world concurrency bug characteristics. In Proc. 13th Intl. Conf. on Architec- tural Support for Programming Languages and Operat- ing Systems, 2008.

[2] S. Savage, M. Burrows, G. Nelson, P. Sobalvarro, and T. Anderson. Eraser: A dynamic data race detector for multithreaded programs. ACM Transactions on Computer Systems, 15(4):391411, 1997.

[3] Horatiu Jula, Daniel Tralamazza, Cristian Zamfir, George Candea. Deadlock Immunity: Enabling Systems To Defend Against Deadlocks - OSDI'08.

[4] CFS : http://people.redhat.com/mingo/cfs-scheduler/sched-design-CFS.txt - last visited 22 Mar 2009.

[5] http://bugs.mysql.com/bug.php?id=7823

[6] MySQL Server: http://www.mysql.com

[7] Kprobes - http://www.ibm.com/developerworks/library/l-kprobes.html

[8] Karim Yaghmour and Michel R. Dagenais. Measuring and characterizing system behavior using kernel-level event logging. In Proceedings of the 2000 USENIX Annual Technical Conference, 2000.

[9] LTT - http://www.opersys.com/LTT/

[10] LTTng - http://ltt.polymtl.ca/

[11] E. Farchi, Y. Nir, and S. Ur. Concurrent Bug Patterns and How to Test them. Workshop on Parallel and Distributed Systems: Testing and Debugging, 2003.

[12] George W. Dunlap , Samuel T. King , Sukru Cinar , Murtaza A. Basrai , Peter M. Chen, ReVirt: enabling intrusion analysis

through virtual-machine logging and replay, ACM SIGOPS Operating Systems Review, v.36 n.SI, Winter 2002

[13] Y. Eytani and T. Latvala. Explaining Intermittent Concurrent Bugs by Minimizing Scheduling Noise. Haifa Verification conference, Haifa, Israel, 2006, Revised Selected Papers. Lecture Notes in Computer Science 4383, Springer, 2007.

[14] M. Musuvathi, S. Qadeer, T. Ball, and G. Basler. Finding and reproducing heisenbugs in concurrent programs. In OSDI, 2008

[15] C. Cadar, D. Dunbar, and D. R. Engler. Klee: Unassisted and automatic generation of high-coverage tests for complex systems programs. In OSDI, 2008.

[16] Feng Qin , Joseph Tucek , Jagadeesan Sundaresan , Yuanyuan Zhou, Rx: treating bugs as allergies—a safe method to survive software failures, Proceedings of the twentieth ACM symposium on Operating systems principles, October 23-26, 2005, Brighton, United Kingdom

[17] Soyeon Park, Shan Lu, and Yuanyuan Zhou. CTrigger: Exposing Atomicity Violation Bugs from Their Hiding Places. To appear in the proceedings of the 14th International Conference on Architecture Support for Programming Languages and Operating Systems (ASPLOS'09), March 2009.

[18] Bryan M. Cantrill, Michael W. Shapiro, and Adam H. Leventhal. Dynamic Instrumentation of Production Systems USENIX 2004

[19] Chapter 4: Linux Device Drviers, 2nd Edition - http://www.xml.com/ldd/chapter/book/ch04.html

[20] Chapter 5: Understanding the Linux kernel, 3rd Edition.

[21] 2.6.11: http://www.kernel.org/pub/linux/kernel/v2.6/linux-2.6.11.tar.bz2

[22] 2.6.23: http://www.kernel.org/pub/linux/kernel/v2.6/linux-2.6.23.tar.bz2