Some Discussion on the Low Latency Patch for Linux [*]

Yu-Chung Wang and Kwei-Jay Lin

Department of Electrical and Computer Engineering
University of California, Irvine
Irvine, CA 92697-2625
{wangy,klin}@ece.uci.edu

Abstract

Some recent discussions in the Linux circle are on the low latency patch posted by Ingo Molnar. There are definitely pros and cons on the usage of preemption points (or "schedule points" in Ingo's terminology). The preemption point approach is to solve the problem that the Linux kernel is non-reentrant. In this paper, we present the justification for inserting preemption points and also study their distribution and risk levels.

1 The latency of Linux kernel

Recently there have been some interesting discussions in the Linux circle on the low latency patch posted by Ingo Molnar [3]. Arguments for and against the usage of preemption points [1] (or "schedule points" in Ingo's terminology) in Linux have been presented. The problem that preemption points are addressing is the non-reentrant nature of the Linux kernel. In Linux, a thread is executed in the kernel mode when the thread issues a system call or when there is an interrupt. In both cases, CPU will be held by this thread until the thread leaves the kernel mode. Therefore, other user threads will be blocked even when they have a higher "priority" or "urgency" than the thread in the kernel mode. For real-time applications, this causes priority inversion when another thread with an earlier deadline is waiting. For multimedia applications, some time-critical operation may experience jitters when another thread is executing long system calls.

2 Preemption points

In RED-Linux [2], the kernel is designed to lower its response latency by using *preemption points* (PP). When a normal system call executes to a PP and if a real-time thread is waiting for execution, the system call will be preempted. The kernel thread executing the system call will yield CPU to the real-time process waiting for execution by simply calling "schedule()" voluntarily. Each of these preemption points in RED-Linux looks like:

```
if (realtime_job_waiting) schedule();
```

By inserting the above statement in many places in the kernel, whenever a real-time job with a high priority competes for CPU time, a kernel execution may be preempted after a small block of code is executed. If no real-time job is waiting, the overhead of PP is just checking the flag.

A more critical issue for PP is that there is no mechanism to guarantee the kernel status will stay the same before and after the preemption. The thread calling "schedule()" must check if some variables of the kernel have been changed. Usually, the thread should discard all global variable values that were held in local variables before its call to "schedule()".

In fact, preemption point is a necessary mechanism even for a monolithic kernel. It is used by a kernel to wait for slower resources. For example, when a program issues a read system call, the kernel will send a request for disk blocks to the hard drive. However, these data blocks usually are not ready within a short time (e.g. 10 ms). The kernel can either enter a busy loop waiting for data or call "schedule()" to yield CPU. To improve the performance in multi-tasking systems, the latter approach is usually selected.

[*]This research was supported in part by UC/MICRO 99-073 and 99-074, Raytheon and GeoSpatial Technologies, and by NSF CCR-9901697.

3 Fully preemptive kernel

Instead of using preemption points, another approach is to make kernel preemptible at any time. However, even if a kernel is designed to be reentrant, it still must disable preemption in sensitive codes that access a shared data structure. Kernels protect their internal data structures by putting them in critical regions embraced with semaphores or spinlocks (although spinlock is not good for single CPU systems). Many real-time OS's, such as LynxOS and RT-Mach, are designed in this way. Assume a job is waiting to start, and another thread is executing in a critical region in the kernel mode. An OS cannot transfer the execution to the waiting job until the latter thread leaves the critical region. The kernel latency is determined by the length of the critical region with the longest execution time.
An interesting question is whether a fully preemptive kernel is a better solution than using preemption points. In our opinion, the answer is not always clear even if we ignore the code complexity of making a kernel fully preemptive. In both cases, we must prove that it is safe for preemption. The effort in showing the correctness of a fully preemptive kernel may not be easier than that of preemption points. For example, let us look at the following lines that show the variable access pattern in a program,

```
1. a
2. a
3. ab
4. ab
5. ab
6. abc
7. ac
8. c
9. c
```

Three variables, a, b and c, are used in 9 statements and must be protected. In a preemptive kernel, it is usually done in the following way.

```
lock(a)
1. a
2. a
lock(b)
3. ab
4. ab
5. ab
lock(c)
6. abc
unlock(b)
7. ac
unlock(a)
8. c
9. c
unlock(c)
```

In this case, the latency of the kernel is the whole segment. However, we can provide another solution by inserting the following code right before line 8 for a possible preemption:

```
{ unlock(c) schedule() lock(c) }
```

In this way, the latency is two statements less than the preemptive kernel solution described earlier.

Unfortunately, this naive solution may not be safe. The value of "c" may be changed during "schedule()" since other jobs are allowed to access "c". We must make sure that all actions performed in the "schedule()" statement do not change variables unexpectedly. One solution is to require the execution to always go back to line 1 to repeat all lines if there is a preemption. More detailed analysis could be performed so that the execution need not restart from line 1 but maybe from line 3 or even line 6. This will be the performance difference between a good preemption point and a not so good one.

The above example shows that preemption points are useful in most cases. In the next section, we study the preemption points proposed by Ingo Molnar [3] (and compare it to ours in [1]). We believe the exercise is important in order to make the Linux kernel a truly real-time kernel in the future.

4 Analysis of Ingo's patch

There are 46 preemption points in Ingo's patch for Linux 2.2.16. As we have discussed before, the most critical issues about the preemption point approach is the safety of the code. One of the biggest concerns against using preemption points is that preemption points seem to be selected randomly for reducing kernel latency. Therefore, we study the preemption points in Ingo's patch and classify them according to their locations and the level of risk. From the locations of preemption points, we can see where the latency bottleneck may be. We thus may have a better idea on where to insert additional preemption points to further improve the kernel latency.

For the level of risk, we tried to verify whether preemption points are safe or not. From our study, we find that most of the preemption points used by Ingo are quite intuitive. Some of his preemption points simply replace those existing preemption code in the original Linux kernel.

4.1 Level of risk

The levels of risk are defined as follows.

- Safe: A safe preemption point presents no risk at all. It may be before a native preemption point like interruptible_sleep_on or some well-known safe situation like copy_from_user or wait_on_buffer.
- High: It means that the safety of these points is doubtful.
- Medium, Low: These preemption points usually present no problem inside the function containing these points. But more rigorous study should be conducted. Since the caller of the function may save some global data on its local stack, if some global data are changed during the preemption, there is a possibility for incorrect behaviors afterward. In our study, there i no clear distinction between medium and low risks. They are classified simply by intuition.
- Undecided: It means that we haven not yet done a detailed study on these preemption points to clearly understand the possible effect.

Out of the 46 preemption points, 34 of them are safe, one of them is considered as high risk, seven of them are low or medium risk and the other four are undecided.
The only high risk preemption point is in "fs/ext2/namei.c" in the function "ext2_find_entry()". The ext2 file system puts a linked name list inside a directory node. This function is used to search for a specific file name in an inode. This preemption point tries to break down the search into smaller pieces by allowing preemptions inside the loop. However, it is dangerous unless the inode has been locked before preemption. Otherwise other processes may delete the current inode and casue invalid execution after the preemption.
To fix this problem, our suggestion is just to remove it since a preemption point have already been defined inside "wait_on_buffer". The size of one node is only 4096 bytes for the ext2 file-system. It does not take that long to scan the data in a single node.

4.2 Location of preemption points

Table 1 shows the preemption point distribution. 10 preemption points are inside the console driver because Ingo has rewritten the console driver. Therefore, it may be considered as just one preemption point.
The preemption points in the kernel core are related to "fork" and "exit". In these two cases, we need to copy a lot of data structures like file descriptors, page table, file-systems and signals to the forked process from its parent process in "fork", and to remove them in "exit". In both cases, it may take a long time when a lot of files or memory are allocated by the processes.
The preemption points in the memory management are all related to page deallocation. When a process requests a page, if there is not enough number of free pages in the system, the memory system will try to either get a page from buffers or swap a page to the secondary storage system. Both of them require scanning the list to pick up the candidate. Since there are a lot of pages in the system, it may take a long time to do that.
Others are related to memory copy to and from user space. Many of these preemption points may soon be included in the mainstream Linux kernel since Linus Torvalds has expressed the opinion that he would like to adopt them in the near future.
In summary, the first group of preemption points are used to reduce the time spent on list traveling. In the memory system, the list is the page table. In the file system, it is buffer cache or directory cache. In the kernel, it is the file descriptor table or the signal table. The second group is related to memory copy. The console driver needs to copy the physical screen data to and from private buffers. The kernel needs to copy data to and from user space.
Comparing Ingo's patch with the preemption points in RED-Linux [1], we find many of them are the same. There are more in Ingo's patch than ours. However, at least two of our preemption points are not in Ingo's patch. One is in the keyboard driver code. The other is in the Unix socket code. The keyboard driver preemption point may reduce the kernel latency in the order of several hundred microseconds. We believe these two preemption points should be included to make Ingo's patch more complete [1].

5 Preemption points in the original kernel

Some may question whether only 46 preemption points would be enough to make Linux kernel "preemptive". Since Linux kernel is a big software with more than 2 million lines of codes, many expect that much more preemption points should be needed. One reason why 46 PP's may be enough is that Linux already has many preemption points in the code even before the Ingo's patch. A simple study using the

[1] Both Ingo's patch and RED-Linux's preemption points are included in the REDICE-Linux 2.0 distribution from REDSonic

Table 1. Preemption Point Distribution

fs	14
core kernel	9
memory management	9
IPC	1
console driver	10
memory copy to user	3

following shell command can show how many preemption points already exist in the original kernel.

```
grep need_resched $(find . -name '*.c')|wc
```

The result from the above command on Linux version 2.2.15 is 42. For example, the following is taken from "drivers/char/mem.c" in Linux 2.2.15 without any low latency patch.

```
do {
  unsigned long unwritten
       = clear_user(buf, PAGE_SIZE);
  if (unwritten)
      return size + unwritten - PAGE_SIZE;
  if (current->need_resched)
      schedule();
  buf += PAGE_SIZE;
   size -= PAGE_SIZE;
} while (size);
```

The line calling "schedule()" is in fact a preemption point. In other words, preemption points have been used by kernel developers for a long time to improve the responsiveness of Linux kernel. They may not be written like "conditional_schedule()" as in Ingo's patch or "MBP()" on RED-Linux 2.0.35. But the idea and the structure are the same.

6 Summary

In this paper, we have discussed the reasons why preemption points are useful to reduce the kernel latency. We have also analyzed the preemption points included in the Ingo's patch. Although the patch may look random at first, we believe Ingo's patch (together with ours in RED-Linux) is very useful if kernel latency is a concern for real-time applications. Since the inclusion of preemption points causes only minimum extra delay in normal kernel operations, we suggest the adoption of these patches in all Linux-based systems.

References

[1] Y.C. Wang and K.J. Lin. Enhancing the real-time capability of the Linux kernel. In *Proc. of 5th RTCSA'98*, Hiroshima, Japan, Oct 1998.

[2] Y.C. Wang and K.J. Lin. Implementing a general real-time framework in the RED-Linux real-time kernel. In *Proc. of RTSS'99*, Phoenix, Arizona, Dec 1999.

[3] Ingo Molnar, http://people.redhat.com/mingo/lowlatency-patches/.