

Increasing Timing Resolution for Processes and Threads in Linux

Stephen Hines, Bartow Wyatt, and J. Morris Chang

Department of Computer Science
Illinois Institute of Technology
Chicago, IL, 60616-3793, USA
{hineste | wyatbar | chang} @charlie.iit.edu

Abstract

Abstract goes here

Keywords: `getrusage()`, `getpinfo()`, Linux, kernel

1. Introduction

The goal of the project was to develop a method for obtaining timing information for Linux processes and threads that is more precise than the one currently provided by `getrusage()`. The `getrusage()` system call is a fairly standard one, that returns the time a process has been running (CPU time) in terms of seconds and microseconds. With even the most recent builds of the stable Linux 2.2.14 kernel, the timing is obtained by sampling the internal clock tick of the Linux operating system. This type of sampling can lead to imprecision in the timing of processes and threads as the basic clock tick unit in Linux (the constant HZ) is only operating at 100Hz. With this 100Hz system tick, it is possible to only approximate times within 10ms of the actual time for each sampled time period. Thus several of the digits in the microsecond portion of the `getrusage()` return values are inaccurate from the beginning. The current implementation of `getrusage()` uses these ticks to calculate the actual user time spent on a process. Whenever data is collected in this fashion, there is always the possibility that a sampling error is induced. When the Linux kernel samples this 100Hz system tick, it is possible to obtain inaccurate timing results, as the process may swap in right before the tick and swap out right after the tick. This situation will then cause an incorrect 10ms to be added to the user time of the process, when in fact the actual time spent is much less than 10ms. With any long term process it is apparent that this imprecision in data acquisition might not amount to much of a difference, however with very fast operations, this error can add up quickly, and the accuracy of the timing can be affected greatly.

This type of sampling error can show up especially in multi-tasking operating systems such as Linux. The 2.2.14 build of the Linux kernel will swap processes and threads every 210ms. The processes to be swapped in and out are kept as a list of `task_struct` objects. The `task_struct` contains a wide variety of information pertinent to the process or thread, including its priority, its process identification number, and its running status. The process list is then put into order by the scheduler, using a complex algorithm involving priority and previous running time. In order to obtain accurate timing information for processes and threads, while not overburdening the kernel, a scheme using the time stamp counter of Intel's Pentium processor architecture can be developed. By reading the time stamp counter before and after process swapping, it is possible to keep track of the number of clock cycles that a particular process has been active for. Using this method within the scheduler and updating the `task_struct` in the kernel for the new time stamping information, a new system call `getpinfo()` can be developed. This system call can then be used to obtain more precise results for the actual CPU time spent on a thread or process.

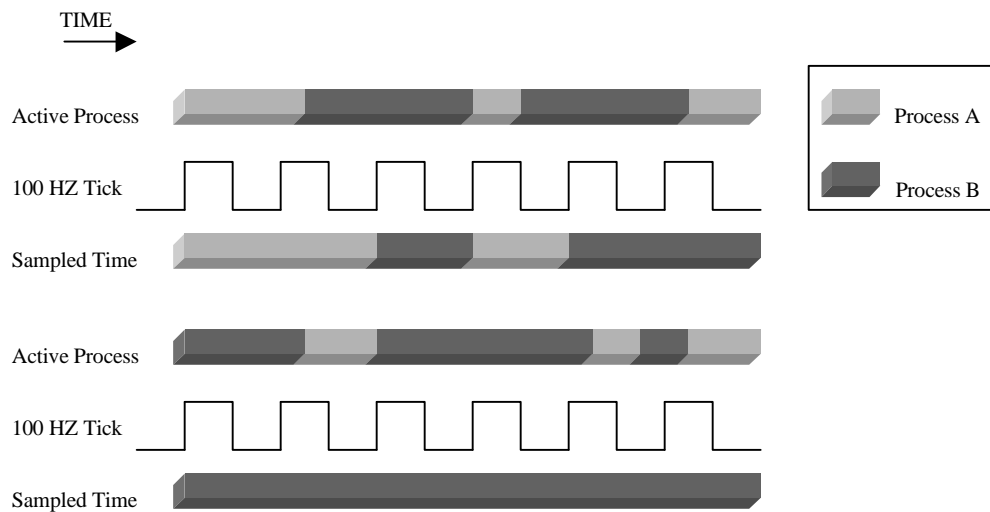
2. Previous Work

The Linux kernel itself provides one method for obtaining time information for particular threads and processes: the system call `getrusage()`. This system call returns the resource usage of a given task, including messages sent and received, as well as the amount of system and user time it has spent being processed. The `getrusage()` system call provides a considerable amount of versatility, providing access for the programmer to an invaluable collection of information about a given process or thread. However, this function suffers from lack of precision in some respects, and thus is not as useful as it could be for current computer science research purposes. The most recent implementation of `getrusage()` in the Linux 2.2.14 kernel uses an internal system clock to sample the various data that the system call provides. This system clock operates at only 100Hz for an i386 build of the kernel, and thus provides

a timing resolution of only 10ms. This resolution is clearly not enough for current research needs, as accurate timing becomes all the more important when dealing with such hardware/software areas as memory allocation and heap management.

In addition to this, sampling errors can add up seriously when trying to obtain fine timing results. Figure 1 depicts two scenarios involving the sampling of the Linux clock tick. In the top example, it is apparent that although the sampled timing information is incorrect at times, the end result is a fairly accurate one. This is true for most sampled systems, as the minor errors will balance out in the long term. The second picture displays a worst case scenario for such a sampling method, as process B is credited for all of the processing time, just because process A swaps in and out before the clock ticks occur. In both cases, minor errors occur during the collection of data, and thus this method is imprecise for timing events that occur in less time than one timer tick (10us). For long-term processes and threads, the system call *getrusage()* may obtain an accurate run time, however this type of sampling error will cause many problems with the precise collection of timing data required to do systems research. Other methods have been developed to take precise time measurements using external hardware. However methods using this type of hardware are both difficult and expensive to set up, and thus are impractical for many researchers.

Figure 1. Sampling error with *getrusage()*



3. Approaching the problem

Several considerations had to be made before attempting to obtain a greater resolution in process and thread timing under Linux. One of the most important considerations was to decrease the amount of overhead induced in the kernel execution for managing the new clock cycle timing addition to the *task_struct*. The *task_struct* is a special type of data structure in the Linux kernel that is maintained for each particular process or thread. It is usually accessed by the *getrusage()* system call to return some resource information back to the user. The *task_struct* however is updated by the kernel, so the user is only capable of reading these resource values from the structure. The initial method called for a reworking of the *getrusage()* system call to return the number of actual clock ticks that a given process has been running for. While this method appears both simple and reasonable, it was finally decided against, since making major modifications to *getrusage()* would cause it to no longer be usable by older programs. By keeping the timing information as clock ticks, the value is much more flexible for programming, and also incurs less overhead in the kernel. The idea of future incompatibility seemed to make this approach less appealing, considering that *getrusage()* is a fairly popular and standard system call for many Linux programs. In addition to this, *getrusage()* is currently implemented to only return information concerning the active calling process. For maximum flexibility, it would be necessary for the new timing information to be accessible by all processes.

The second method introduced was to create a brand new system call that could be used to return the necessary timing information for either the calling process, or another process specified by its process identification

number (PID). By creating a brand new system call, the new timing method is easier to integrate into existing kernels, as well as being less processor-intensive for obtaining the necessary time stamp information. Using the standard *getrusage()* would have incurred all sorts of additional overhead, since it will collect all resource usage data, even the data that is not necessary for the accurate timing proposed. In addition, by implementing the additional feature to access the timing information in another process, this system call gains even more potential applications. For example, it can be used to implement new scheduling algorithms for Linux based on current process runtime information. Thus option two was selected for its support of previous software, its ease of integration, as well as its new application possibilities.

3.1 Implementing a precise individual process and thread timer

In order to achieve the increase in timing resolution, the assembly language *rdtsc* command was necessary. This instruction returns a 64-bit integer in two separate 32-bit registers. These numbers are the upper and lower portions of the Intel Pentium architecture's time stamp counter (TSC) register. The 64-bit time stamp counter register is initialized to zero upon boot, and is incremented with each clock tick of the processor. Using the *rdtsc* instruction, it is then possible to take readings from the processor at various times. With this method, it is possible to set up the Linux scheduler such that processes being swapped in and out keep track of their individual clock cycle counts. The time spent between swap out and swap back in is subtracted off of the total, and thus wait time is not factored into the active running time of a process. Using this method, it is then possible to implement the new system call *getpinfo()* to retrieve the clock cycle count for a given process or thread.

The new system call was implemented using an Intel Pentium Pro 166Mhz processor with 96MB of memory. This machine was then set up with Red Hat Linux 6.2. The Linux 2.2.14 kernel was then modified to handle the newly developed system call. Newer Intel chips such as the Pentium II support out-of-order-execution of instructions, and thus modifications need to be made to the system call for these systems to obtain accurate timing information. One such method for fixing this problem is using a serializing instruction such as *cpuid*, before calling *rdtsc*. This method works well, however it does add additional overhead to the system call, and thus the Pentium Pro system was chosen for its optimal implementation of *getpinfo()*.

3.2 Test procedures for *getpinfo()*

One of the important requirements for implementing a cycle counting mechanism for process and thread timing is that the overhead of the system call needs to be minimized, or too much execution time will be wasted just acquiring the timing data. Because of the large amount of work that *getrusage()* does when it is called, the new *getpinfo()* system call should execute in fewer clock cycles. To measure this, a program named *test_cycles* can be constructed to call *rdtsc* before and after each of the system calls. Then, by subtracting the start cycle count from the end cycle count, the total overhead of the system call can be calculated in terms of clock cycles. Also by looking at the standard deviation of the overhead measurements, the boundedness of execution time for each system call can also be analyzed.

In order to test the accuracy and precision of the new system call, another test method had to be devised. Accuracy is an easily tested quality for these system calls, as it only requires that each of the system calls be made before and after a given procedure is executed. By using a long-running finite execution time procedure, the accuracy of both methods can just be compared using the average execution time. Precision of the system calls is also easily tested for, since the collected execution time data can be analyzed using a simple standard deviation. By taking the standard deviation for all of the collected data, it is possible to obtain an accurate reading of just how precise each of the system calls are. For the test program, a prime number generation scheme was chosen. The program *primes* would generate the first N prime numbers a set number of times, recording the execution time data for each particular trial. This algorithm is suitable for comparing execution time accuracy since the execution time of *primes* can be varied widely by changing the value of N.

The final test is a procedure designed to show the accuracy and precision of the new system call *getpinfo()* in a situation that has previously been impractical to measure with software methods such as *getrusage()*. The test program *malloc_stats* is constructed to measure the cycle count of Doug Lea's *malloc* routine using the *getpinfo()* system call. The program will perform an S-byte allocation 1000 times and collect all the cycle count data, and then calculate the average execution time as well as its standard deviation. It is unnecessary for this routine to be executed

using *getrusage()*, as its resolution is too low to measure any of the information and will only return 0ms or 10ms (depending on where the Linux tick falls) as the overall time of any allocation. The end result of this program will show just how powerful the resolution increase of *getpinfo()* actually is.

4. Results and Comparison

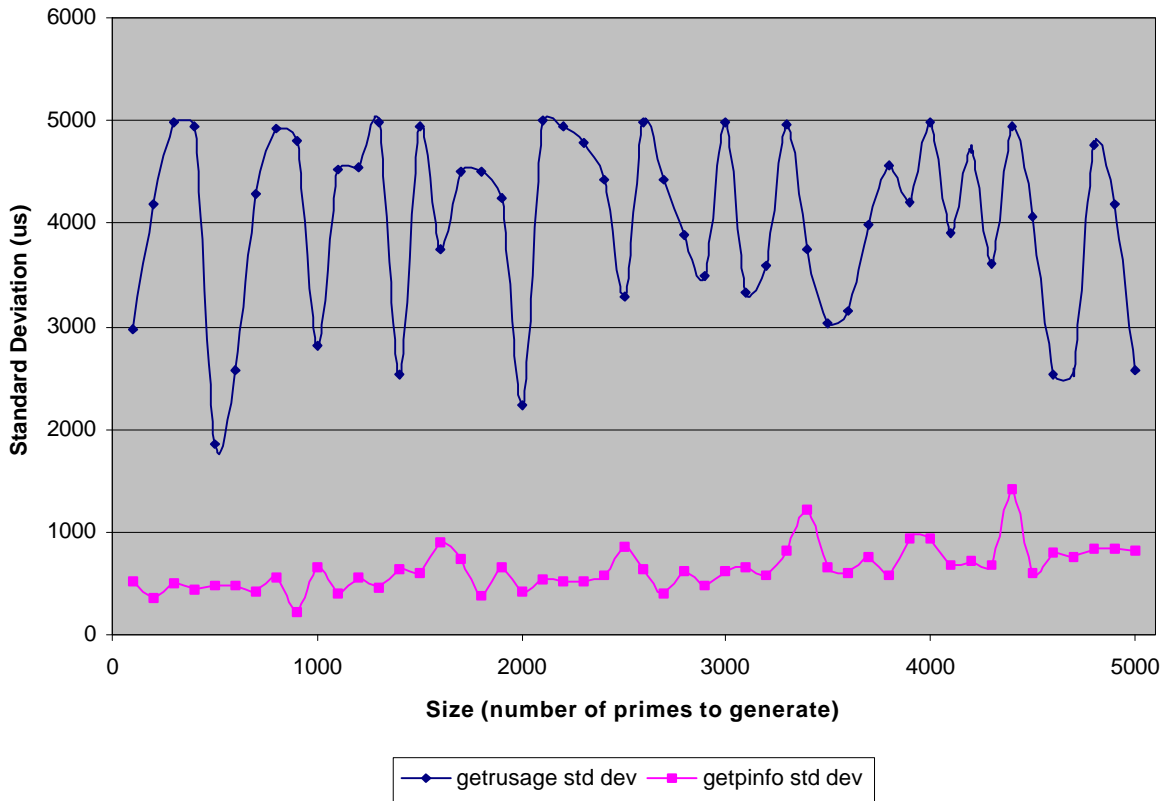
One of the most important considerations in implementing *getpinfo()* is making sure that the measurement overhead is minimal and will not interfere with the measured data. Table 1 compares the overhead in actual clock cycles for using both *getrusage()* and *getpinfo()* as generated with the program *test_cycles*. The results show that *getpinfo()* takes approximately half of the time it takes *getrusage()* to complete. It is also noticeable that the standard deviation for *getpinfo()* is lower, which shows that it occurs within a tighter timing boundary than *getrusage()*. Thus the *getpinfo()* system call has a more deterministic execution time than *getrusage()*. This cycle count overhead measurement is very important since the average cycle count for a system call can then be subtracted from collected data to obtain more precise timing information. This data clearly supports the goals mentioned above in that the timing measurement overhead can be further reduced by introducing a new system call, rather than just modifying the bulkier system call *getrusage()*.

Table 1 Comparing *getrusage()* and *getpinfo()* overhead

<i>getrusage()</i>		<i>getpinfo()</i>	
Average Cycles per call	Standard Deviation	Average Cycles per call	Standard Deviation
416	161	213	66

An important consideration for the new system call *getpinfo()* was increasing the resolution of process and thread timing under Linux, while not overburdening the processor with too much data acquisition. To measure the precision of this new system call, a comparison can be made with *getrusage()* when running a standard C program. To do this, a program that generates the first N prime numbers is constructed and the system calls *getrusage()* and *getpinfo()* are injected into the code to determine the average run time of the procedure when run a set number of times. Figure 2 shows a comparison of the standard deviation of both *getrusage()* and *getpinfo()* when calculating the run time of the prime number generating program *primes*. In this trial, the number of primes to generate N is varied from 100 to 5000 in increments of 100, and each of the values of N is tested 1000 times. The average run time can then be obtained from this collected data for each particular value of N. Looking at the results for each system call, it is apparent that both obtain similar results for the average run time. The minor errors occurring in the sampling of data by *getrusage()* mask each other in the long term, which then provides a fairly accurate average running time for the program. However, the standard deviation can then be calculated and used to compare the overall precision of each system call. Based on the standard deviation data collected for the prime number program, *getpinfo()* provides a 527% increase in precision compared to *getrusage()*.

Figure 2. Comparison of *getrusage()* and *getpinfo()* precision



The full impact of the *getpinfo()* system call can be clearly seen in the results obtained from the *malloc_stats* program. For this trial, the *malloc_stats* program was used to allocate S-byte chunks with S varying from 100 to 5000 in increments of 100. These allocations all fit under the small block allocation scheme (requested size is less than size of a page) for Doug Lea’s *malloc* function. For each of these allocations, the time (cycle count) should be approximately the same, as the majority of the work for the *malloc* function will be spent managing the appropriate bins (locking and unlocking). Based on the data collected by the *malloc_stats* program, Table 2 presents the cycle count for a small block allocation as 591, which works out to approximately 3.56 microseconds on the 166MHz machine. When attempting to measure the timing for *malloc* using the *getrusage()* system call, the only values returned are 0ms and 10ms, once again based on where the Linux system tick falls. These values are clearly wrong and thus *getrusage()* is both inaccurate and imprecise for performing such fine timing measurements.

Table 2 Cycle count and time for small block *malloc*

<i>getrusage()</i>		<i>getpinfo()</i>	
Average Cycles	Time (166Mhz)	Average Cycles	Time (166Mhz)
Immeasurable	Immeasurable	591	3.56us

5. Future Work

Looking at the functionality of the system call *getpinfo()*, a great many possible applications can be seen. First of all, increased timing resolution paves the way for more in-depth research on time-intensive applications. Possibilities include advancements in the fields of process scheduling, distributed computing and clustering, and

system performance analysis. With this type of precision, it is possible to measure the timing for many events within the processor that had previously only been approximated.

One application for *getpinfo()* is using it to develop new process scheduling algorithms. By looking at the total run time of a given process or thread, new scheduling algorithms can be developed to take advantage of this information. Advancements can be made in the field of real-time applications, as the accurate timing information can be used to determine if the process is actually getting the proper amount of CPU time. Distributed computing is another field in which the system call *getpinfo()* can be used to enhance the overall performance of the system. Load balancing among the processors of a distributed system is one of the primary topics of distributed computing research. Once again the scheduler can be modified using the timing data, allowing for the workload to become more balanced for the various processors. Consequently, this type of even distribution can allow for each processor in the distributed system to be more productive than before.

With *getrusage()*, process and thread timing was not accurate enough to measure the time cost of memory allocation and reclamation. The *getpinfo()* system call can provide the necessary precision, allowing better research to be done concerning memory management and garbage collection. According to the data collected using Doug Lea's *malloc()* function, it takes approximately 591 clock cycles to allocate a small block of memory. Even with a 166MHz Pentium, this amounts to a mere 3.56 microseconds. This value is unattainable with methods using *getrusage()*. The *getpinfo()* system call allows for the further fine-tuning of the *malloc()* function, since it is now possible to more accurately measure the time of allocations.

One of the biggest applications of the *getpinfo()* system call is in profiling the heap memory management schemes of multi-threaded Java programs. Object oriented programming languages such as Java have become quite popular, and rely heavily on using the heap to allocate their necessary objects. By using specifically placed calls to *getpinfo()* in the heap memory management section of a Java Virtual Machine (JVM), it is possible to obtain accurate timing information for the various heap allocations of each thread and deallocations from the garbage collector. This type of research can help to develop a better scheme for partitioning the heap area of the JVM or to improve the garbage collection scheme.

The system call *getpinfo()* can also be expanded for use with other processor architectures that are supported by Linux. Both the Alpha and the SuperSPARC II offer assembly instructions similar to the *rdtsc* instruction of Intel's Pentium architecture. The SuperSPARC II possesses a performance counter that can be modified to return the number of clock cycles spent executing instructions. The Alpha has an instruction known as *rpsc* (Read Process Cycle Counter), which also works the same way that *rdtsc* works. Modifications can be easily made to the *getpinfo()* routine to allow for the use of these architecture's particular assembly instructions to keep the cycle count and update the *task_struct* as necessary.

6. Conclusion

The *getpinfo()* system call provides an accurate, simple and free solution to the problem at hand for many computer science researchers, precision timing for processes and threads. Of course, many other fields of computer science research can benefit from this development as well, since high precision timing is becoming more important as computer processors attain higher clock speeds, which allows more room for previously mentioned *getrusage()* errors. Although a small amount of overhead is incurred by retrieving the timing information via the system call *getpinfo()*, the overall resolution of process and thread timing measurement in Linux has been vastly improved.

7. References:

<http://www.innotts.co.uk/bsdi/9812/0011.html> - *Getrusage()* resolution

<http://g.oswego.edu/dl/html/malloc.html> - Doug Lea's memory allocator

http://developer.intel.com/software/idap/resources/technical_collateral/pentiumii/RTDSCPM1.htm - Using RDTSC

<http://www.sun.com/microelectronics/manuals/STP1021UG.pdf> - SuperSPARC II counter

http://www.unix.digital.com/faqs/publications/base_doc/DOCUMENTATION/V40D_PDF/APS31DTE.PDF - Alpha