Real-time data collection in Linux: A case study

Steven A. Finney
Department of Psychology
Ohio State University

(published as Finney (2001): *Behavior Research Methods, Instruments, and Computers*, v 33, pp 167-173.

Abstract

Multi-user UNIX-like operating systems such as Linux are often considered unsuitable for real-time data collection because of the potential for indeterminate timing latencies resulting from preemptive scheduling. In this paper, Linux is shown to be fully adequate for precisely controlled programming with millisecond resolution or better. The Linux system calls that subserve such timing control are described and tested, and then utilized in a MIDI-based program for tapping and music performance experiments. The timing of this program, including data input and output, is shown to be accurate at the millisecond level. This demonstrates that Linux, with proper programming, is suitable for real-time experiment software. In addition, the detailed description and test of both the operating system facilities and the application program itself may serve as a model for publicly documenting programming methods and software performance on other operating systems.

Response times in human behavioral experiments are usually reported in milliseconds, and it is often considered both necessary and sufficient for stimulus presentation and response collection to occur with reliable millisecond timing. Achieving and guaranteeing such precision on a computer can be non-trivial, and this is particularly problematic with multi-tasking operating systems such as Windows, UNIX, or Linux (Murrel & Kowalski, 1980; Myors, 1998, 1999; Pavel, 1982). One concern is the potentially indeterminate delays resulting from an operating system which needs to schedule multiple processes, and which can preempt one process to allow another process to run. Such scheduling can impact timing of the critical data collection process. A second concern involves swapping and memory residency: if a program is partially resident on disk, there may be a delay while it is read into memory, and this delay may impact the timing of data collection. A third concern is

that program access to hardware I/O devices is usually mediated by the operating system; this decreases the programmer's control over input and output timing. In this paper, I demonstrate that Linux (Torvalds, 1999), a modern UNIX-like operating system, is fully capable of timing resolution at the millisecond level or better, and I describe the facilities used for such programming. I then describe an implemented data collection software system for tapping and music performance experiments (FTAP; see Finney, 2001), and show that this moderately complex program processes input and output with reliable millisecond precision.

This paper is not the first published description of software for psychology experiments on UNIX-like operating systems. For instance, Perlman (1982) described experimental control software for an early version of UNIX, but it relied on a timer with only 17 msec resolution.[1] Cohen and Massaro (1994) described a real-time speech system for a Silicon Graphics computer, but a high-end workstation was required. More importantly, neither Perlman nor Cohen and Massaro provided details on tests of actual system performance, leaving the timing accuracy of the systems open to question.

The current paper explicitly addresses the timing performance of Linux, a readily available operating system that runs on standard PC hardware. Linux is a multi-user, multi-tasking operating system kernel that has similar functionality to the UNIX operating system developed at Bell Laboratories in the 1970's (Ritchie & Thompson, 1978). Linux is quite robust and runs efficiently on PC's based on the Intel Pentium chip, as well as many other architectures including Apple MacIntosh and DEC Alpha. It is available at little or no cost in source-code form. This paper specifically addresses Linux on Pentium-based computers. Although much of the discussion should apply to Linux on other architectures, as well as to contemporary versions of UNIX, timing performance may need to be explicitly

[1] It is worth noting that Ulrich and Giray (1989) have argued that timing measurements with even coarser resolution than this may be adequate for human psychology experiments under some circumstances.

The program described here, FTAP, was written for my dissertation research at Brown University (Finney, 1999), and was originally implemented on a Silicon Graphics computer running the IRIX operating system. I thank Jim Anderson, David Ascher, Peter Eimas, Mike Tarr, and Bill Warren for their assistance and support during this period. The port to Linux and the tests reported here were completed during a postdoctoral fellowship at Ohio State University; I thank Caroline Palmer and Pete Pfordresher for their assistance and advice, as well as for comments on an earlier version of this paper. I also thank Paul Iddings and Doug Michels for my early UNIX education, and I extend special thanks to Shane Ruland for his invaluable and good-natured software and hardware assistance.

Author's current address: Department of Psychology, Ohio State University, 142 Townshend Hall, 1885 Neil Ave, Columbus, OH, 43210. Email: sf@dactyl.som.ohio-state.edu. FTAP web site: http://dactyl.som.ohio-state.edu/ftap.

verified for different hardware and operating system combinations.[2]

Real-Time Programming in Linux

Three capabilities are important for real-time programming: (1) the ability to determine the current time with high precision (at least to the millisecond), (2) the ability to keep a process from being pre-empted by another process, or being swapped out, and (3) the ability to suspend process execution for a short (and deterministic) period of time. Linux provides a number of system calls and interfaces for these purposes; these will be briefly described here. More information about each call can be found in the Linux documentation.

The use of some of these facilities is restricted to a particular privileged user (the superuser, also known as "root"). Linux provides a mechanism (the set-user-id, or "setuid" capability) which allows normal users to execute certain trusted programs which make use of such privileges. The system administrator decides whether to allow a program to be configured as setuid "root".

Scheduling and Current Time

Access to operating system services in Linux occurs when a program makes a system call. Three system calls provide for giving a process high scheduling priority and for retrieving the current time.

The *gettimeofday* call returns the current time, using a data structure which allows for microsecond precision. However, this does not necessarily mean that the internal clock that the reported time is based on is actually accurate to the microsecond. For instance, some early UNIX systems updated *gettimeofday* only once every 10 millseconds. The actual resolution of *gettimeofday* thus needs to be determined.

The *sched_setscheduler* system call puts a process in a special class (SCHED_FIFO) of high-priority processes; such a process will never be preempted by a process in the normal priority class. Similarly, as soon as a SCHED_FIFO process is ready to run, it will immediately preempt any running normal-priority process. If a data collection program is the only active SCHED_FIFO process (which will usually be the case; system processes run with normal priority), it will never be preempted by another process. Such a high-priority process will not prevent the kernel from servicing hardware device interrupts (which are designed to be processed very quickly); a high-priority process may also voluntarily

[2]Throughout this paper, I will use the term "real-time" to refer to the millisecond precision that is generally considered acceptable for human experimental psychology, and I will show that both the average and worst-case times under standard Linux can meet this criterion. However, in domains outside of experimental psychology the criteria for real-time may be much more stringent than the millisecond criterion used here. Linux users requiring more precise control can use specialized real-time versions of Linux (see http://rtlinux.org).

suspend processing (see below). *Sched_setscheduler* requires root privileges. (Linux also provides the *setpriority* system call, which gives a process high priority within the normal scheduling class. As will be shown below, the *setpriority* call is not adequate for real-time programming.)

The *mlockall* system call locks a process in memory, preventing it from being swapped out. This is important because even a high-priority process might be swapped out if it voluntarily relinquished control of the CPU, and time would then be required to read it back in from disk. *mlockall* requires root privileges.

There are thus two important features to be tested. First, what is the actual timing resolution of *gettimeofday*? Second, how well does *sched_setscheduler* work in giving a process reliable real-time scheduling without premption? These questions were addressed by a simple benchmark program which called *gettimeofday* 1,000,000 times and recorded the time returned by each call. Of particular interest is the elapsed time between successive calls. A small mean time difference would indicate high-resolution updating of the *gettimeofday* clock. A large maximum time difference would demonstrate a problem with worst-case scheduling performance. A low standard deviation would indicate that the mean time difference is an accurate characterization of average performance.

Two variables were manipulated in this test. The first variable was priority condition, with three values: (1) a process running with normal user privileges, (2) a process with root privileges running with high priority set by *setpriority* (using the maximum value of -20), and locked in core with *mlockall*, or (3) a process with root privileges running as a high priority SCHED_FIFO process (using *sched_setscheduler*), and locked in core. The second variable was system load, that is, the amount of activity on the system. Running on a loaded system allows verifying that an intended real-time process actually gets highest scheduling priority; a real-time process should not be affected by such a load. Although experimental data collection should be done on a relatively quiescent system, it is still important to understand how a data collection process might respond to unanticipated system activity (e.g., background system processes). There were 2 different load conditions: (1) a relatively unloaded system (a single logged-on user running the test program), or (2) a system with heavy process contention, with a load consisting of 3 simultaneous normal-priority programs doing continuous arithmetic. Each of the three priority conditions was tested under both load conditions. The tests were run on a Gateway Computer with a 350 MHz Pentium II processor and 128 megabytes of RAM, running a Linux 2.2.12 kernel (RedHat distribution 6.1). There was little or no network activity, X Windows was not active, and the machine had been recently rebooted; these factors can all contribute to system load and affect worst-case performance times in low priority conditions.

The data from 10 runs of the test program (a total of 10,000,000 calls to *gettime-*

Table 1: Times between calls to *gettimeofday* under different priority and load conditions

Configuration	Mean Difference (microseconds)	Maximum Difference (microseconds)	Standard Deviation (microseconds)
User, no load	1.29	246	1.17
User, load	5.23	840019	1601
Setpriority, no load	1.23	322	.48
Setpriority, load	3.11	630031	1079
SCHED_FIFO, no load	1.25	72	.46
SCHED_FIFO, load	1.25	69	.46

ofday) are shown in Table 1. The mean time difference when running with normal user priority without any process load was acceptable (1.29 microseconds), as was the worst case time (246 microseconds). However, performance degraded dramatically when there was a process load, with a worst case time of 840 milliseconds (unacceptable by any reasonable standards). When running with high priority set by the the *setpriority* system call, the unloaded configuration gave acceptable mean and maximum time differences of 1.23 and 322 microseconds; the timing had less variance than with normal user priority. However, performance again degraded dramatically under process load, with an unacceptable worst-case time of 630 milliseconds. The privileged *setpriority* call, although giving improved performance relative to a normal user process, is not adequate.

In contrast, a real-time SCHED_FIFO process had a worst case time of less than 75 microseconds, even under heavy process load. When the *sched_setscheduler* call is used to give a process high priority, Linux provides timing resolution that is more than sufficient for millisecond resolution programming.

Process Suspension

Another useful facility in real-time programming is the ability to suspend processing for a short time (a millisecond or less); such suspension will ideally allow other processes or process threads to execute. Linux provides at least three mechanisms for suspending processing, but two of them have significant drawbacks when fine timing resolution is required.

The first mechanism is the *setitimer* system call, which generates signals at fixed time intervals specified in microseconds. A process can set a timer and then suspend itself; the process will be reactivated when the signal arrives. However, with the default Linux configuration the minimum time interval at which such interrupts actually occur is an unacceptable 10 msec (based on the kernel HZ variable), even if a user with root privileges requests a smaller time value.

The second mechanism is the *nanosleep* system call (or the related *usleep* library function), which suspends the calling process for a time increment specified in nanoseconds or microseconds. When called from a process in the normal priority class (even one with root privileges), the smallest resolution of these calls is on the order of 10 or 20 msec, even when a smaller interval is specified. When called from a high-priority SCHED_FIFO process, these routines allow pausing accurately for times from 2 ms down as low as 5 microseconds but such fine resolution is implemented as a busy wait in the kernel that does not allow other processes to run. In addition, if a time *greater* than 2 msec is requested by such a real-time process, the minimum resolution returns to 20 msec.

The third (and best) mechanism for implementing process suspension uses the real-time clock device "/dev/rtc". "/dev/rtc" can be programmed so that a *read* system call on the device will return after a specified interval. When called from a process with root privileges, the specified time can be as little as .12 milliseconds (the period of an 8192 Hz clock), and this timer is quite accurate. When a process is blocked on a *read* on "/dev/rtc", other processes or threads will be scheduled for execution. Although "/dev/rtc" is included with standard Linux (at least on Pentium-based computers), it is not widely documented; a useful description and code fragment can be found in the "rtc.txt" file in the kernel source documentation.

Discussion

Three Linux system calls (*sched_setscheduler*, *mlockall*, and *gettimeofday*) provide for accurate process control at the millisecond level or better; in addition, the "/dev/rtc" real-time clock allows process suspension for sub-millisecond time increments. A number of other system calls have been shown to be inadequate for real-time purposes. Use of the *sched_setscheduler* and *mlockall* system calls (as well as high-resolution use of "/dev/rtc") requires root privileges, but Linux provides the ability for normal users to run programs using these facilities.

Two additional points are worth making. First, the overhead for Linux system calls is extremely low; the time to process a *gettimeofday* system call is on the order of 1 microsecond (this agrees with the system call execution time reported by Rubini, 2000, who measured time by counting processor cycles). Second, programming high-priority processes with *sched_setscheduler* must be done with some care. Such processes genuinely get high scheduling priority; if not programmed properly, they can completely monopolize the system, requiring a system reset to regain access to the computer.

The FTAP Program

Although the facilities provided by Linux appear suitable for real-time programming at the millisecond level, the tests described above did not involve complex data processing,

nor was any data actually input or output. The adequacy of Linux for real-world data collection has not yet been established. This section will describe an implemented Linux-based data-collection program for an interesting class of human behavioral experiments; the real-time performance of the program will be demonstrated in a rigorous way.

FTAP (Finney, 2001) is a program for tapping and music performance experiments; it collects finger movement data (key presses) from an electronic musical keyboard and manipulates the auditory feedback to keystrokes in interesting ways. FTAP can run a wide range of experiments, including synchronization/continuation tasks (Wing & Kristofferson, 1973), synchronization tasks with delayed auditory feedback (Aschersleben & Prinz, 1997), continuation tasks with isolated feedback perturbations (Wing, 1977), and complex alterations of feedback in music performance (Finney, 1997). It is available at no cost in source code form from http://dactyl.som.ohio-state.edu/ftap.

FTAP uses a MIDI (Music Instrument Digital Interface) keyboard for input and a MIDI tone generator for auditory output. For current purposes, MIDI can be characterized as a serial data format with 3 byte messages that specify either Note On (e.g., a key press on a musical keyboard, which would trigger a tone onset) or Note Off (a key release, or tone offset). Each message specifies the key pressed (i.e., the note or pitch value) and the keystroke velocity (loudness). The MIDI protocol itself does not provide time-stamping of input messages nor scheduling of output, so one important requirement for a program is accurate recording of the times of MIDI input, and accurate timing of MIDI output.

The basic architecture of FTAP is shown in Figure 1; processing involves a continuous loop in which the program checks if there is any pending input (if so, it is timestamped and processed) and then checks whether there are any messages scheduled for output. FTAP runs with root privileges and uses the *sched_setscheduler* and *mlockall* system calls. The disk is not accessed during a trial; data is stored in RAM and written to the disk at the end of the trial. Following standard procedure for UNIX-like operating systems (see, e.g., Cohen & Massaro, 1994), data collection should be done on a dedicated machine with no other users logged on, and with unnecessary services (e.g., network activity) kept to a minimum.

The question of whether FTAP provides the desired millisecond resolution can be addressed at two levels. The first question is whether FTAP achieves reliable millisecond scheduling in its central processing loop. The second question is whether the input and output of MIDI data occurs with millisecond precision, i.e., does the underlying Linux MIDI I/O system provide adequate performance. I describe tests addressing both of these issues; importantly, FTAP provides the capability for any user to replicate these tests on their own system.

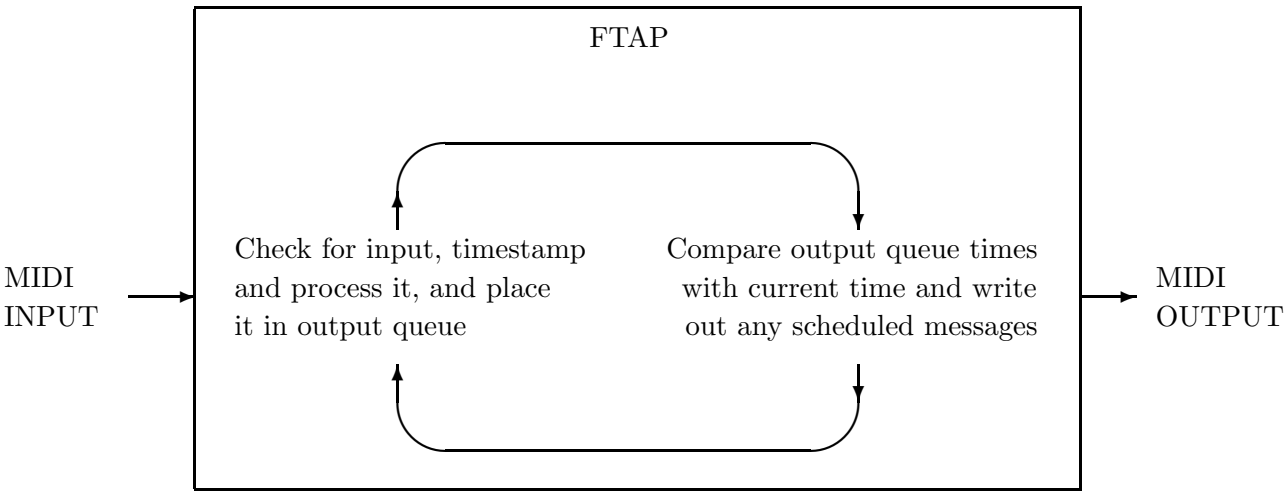

Figure 1. Basic architecture of the FTAP program; conceptually, input and output occur simultaneously. The internal loop indicates control flow, not data flow.

Scheduling tests

Adequate performance of FTAP requires that the input and output routines each be called at least once a millisecond; this will allow FTAP to process I/O with millisecond resolution. Because FTAP runs as a continuous loop of input/output, it is sufficient to test the timing of either the input or the output routine; the output routine was arbitrarily chosen for this purposes. During each execution of FTAP, the time between successive calls to the output routine is measured, and summary statistics (mean and maximum time differences) are provided to the experimenter. If the maximum (worst-case) times between scheduling calls are near 1 msec, adequate performance will be achieved.

As a concrete example of scheduling performance, Finney and Warren (2000) reported a synchronization/continuation tapping experiment run with FTAP, using a 200 MHz Pentium computer and a Linux 2.2 kernel. Twenty subjects performed 66 trials each, giving a total of 1320 trials. The mean trial length was 28 seconds, and data for 60 to 65 keystrokes was collected during each trial. In total, there was more than 10 hours of actual data collection, with more than 80,000 keystrokes.

For each of the 1320 trials, the mean time between calls to the output routine was .49 msec, well under a millisecond.[3] On 1270 out of the 1320 trials, the maximum time

[3] The fact that the time between calls was as large as .49 msec is actually due to an intentional programmed *slowdown* of the FTAP program. FTAP runs in a tight loop as a real-time SCHED_FIFO process; such a process takes priority over any other processes on the computer. A careless or malicious user could

between calls to the output scheduler was 1 msec; on 35 trials there was a single instance of a 2 msec difference, and on 15 trials there was a single instance of a 3 msec difference. These times seem quite acceptable for 10 hours of data collection, particularly because a slight discrepancy between calls to the output scheduling routine does not mean that any data was actually compromised by even these small amounts (there was no input or output data to be processed during the majority of scheduling calls). This test demonstrates that FTAP's internal scheduling is adequate at the millisecond level.

I/O tests

Data collection with FTAP involves a complex system beyond the FTAP program itself, including access to a hardware device for MIDI I/O (a sound card or serial port); see Figure 2. The diagnostics described in the previous section can detect internal scheduling inconsistencies, but they cannot detect problems at the driver or hardware level, e.g., discrepancies between intended output time and actual output time. Access to the MIDI interface is mediated by device-specific driver code in the operating system; Linux does not allow direct access to a hardware device from an application program. The programmer is dependent on the driver for adequate timing, which is potentially problematic for real-time programming.[4]

It is therefore important to verify that the drivers and hardware can process input and output with millisecond resolution, as otherwise the performance of FTAP as a whole will not have accurate timing. In the general case, determining the temporal resolution of input and output on a computer can be very difficult, as it requires the ability to precisely control the timing of input to the system as well as the ability to measure the timing of output. However, two aspects of FTAP simplify the verification of I/O performance. First, FTAP uses MIDI for both input and output; that is, the input message format (MIDI keystroke data) is identical to the output message format. Second, FTAP's functionality includes providing auditory (MIDI) feedback to input keystrokes. These two features facilitate a rigorous test of FTAP using a loop configuration in which the MIDI output is connected (via a MIDI cable) back to the MIDI input; an output message will then be immediately received as input (and interpreted as an input keystroke). If FTAP is configured to give

potentially program an experiment in such a way that no other processes could run and FTAP itself would not be interruptible from the keyboard. To prevent this, the "/dev/rtc" timing mechanism described earlier is used to insert a .49 msec *pause* during each processing loop of FTAP; this pause doesn't significantly impact millisecond resolution measurements, but it allows for a keyboard interrupt or for terminating FTAP from another login.

[4]However, this has the advantage of allowing a program to be independent of the particular hardware device. That is, a program like FTAP which accesses a raw MIDI data stream though the generic Linux device "/dev/midi" is hardware independent because the Linux MIDI driver translates user I/O requests into the necessary hardware commands.

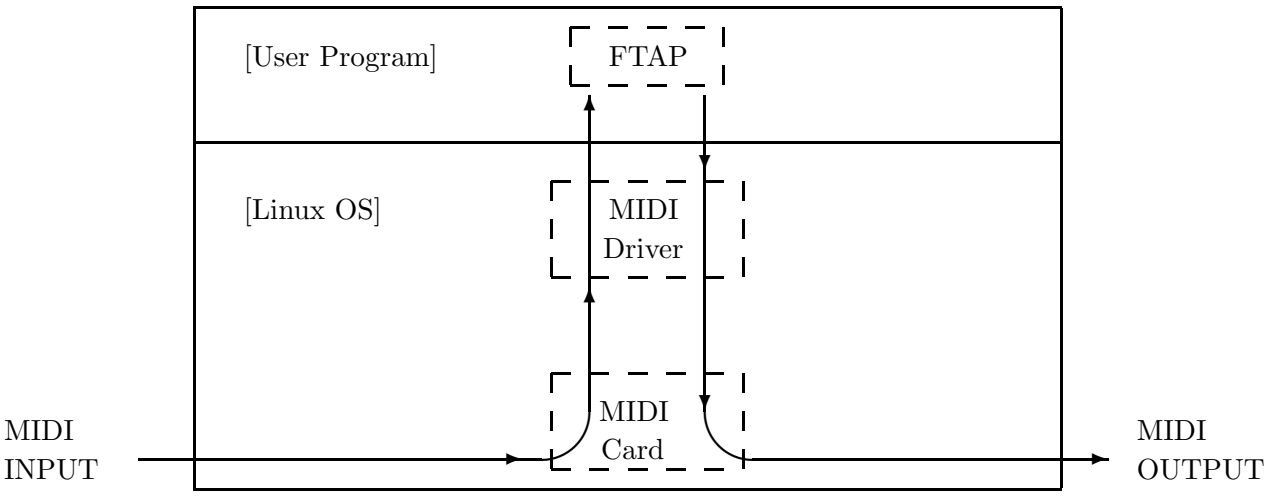

Figure 2. FTAP's interaction with the Linux I/O system. The accuracy of data collection depends upon performance of the entire configuration, not just the FTAP program itself.

immediate feedback to MIDI input messages, a single "priming" message can be repeatedly cycled through the system, and the speed with which such a message is processed gives a measure of FTAP's overall performance. The processing in such a loop test includes a heavy load on the kernel I/O system of virtually simultaneous MIDI input and output at a very rapid rate, as well as FTAP program overhead of reading, assembling, and timestamping MIDI input messages, storing them in memory for later data file output, placing the messages in the output queue, outputting the messages at the proper time, and saving the output messages in memory for writing to a data file. This benchmark provides a much more demanding processing and input/output load than will be encountered in a human behavioral experiment.

What level of performance should be considered adequate? The transmission rate specified by the MIDI standard is 31.25 Kilobaud, equal to 320 microseconds per byte, or .96 msec for a 3 byte MIDI key press or key release message (giving 1040 messages/second). Achieving this hardware-specified rate with such a loop test is the best performance possible, and would achieve the overall goal of millisecond accuracy.

With the appropriate cable, the above test can be run in FTAP using a simple 10 line experiment description file; such a file is included in the FTAP distribution. The performance of this test can be verified by inspection of the FTAP output file, which contains a listing of all MIDI input and output events with millisecond timestamps. The test of system performance is how accurately FTAP can process input messages relative to the rate specified by the MIDI hardware specification (i.e., one message every .96 ms). This test was run on a 350 MHz Pentium with a Creative Soundblaster-16 Sound/MIDI

card, using a Linux 2.2 kernel but replacing the standard Linux MIDI driver with a low-cost commercial driver from 4Front Technologies (http:www.4front-tech.com; see discussion below). The test started with FTAP generating one MIDI Note On and one MIDI Note Off output event (separated by 1 ms); this corresponds to a key press and key release. These events were cycled through the system as described above, and the test terminated after 10,000 MIDI input events were received.

For each of 100 runs of this test, the resulting mean time between input MIDI events was .96 msec, exactly the hardware-specified MIDI transmission rate. The maximum time difference between input events in any run was 2 msec; this occurred rarely. Maximal MIDI throughput was achieved in this test, and the millisecond level precision of the entire FTAP system is verified.

I/O Performance Issues: MIDI Drivers and Hardware

There were some problems in achieving maximal MIDI throughput with the above loop benchmark; not all MIDI card/driver combinations were able to maintain adequate performance. For example, the combination of the Linux drivers and a Roland MPU-401 card failed with overrun errors when the card was used in the UART mode which allows accessing the raw MIDI data stream. In addition, the combination of the Soundblaster-16 sound card and the standard Linux MIDI driver (the OSS/Free driver included with the RedHat 6.1 distribution) only processed one MIDI message every 10 milliseconds with the loop test. This behavior turned out to be a result of polling in the MIDI driver output code ("midibuf.c"); the driver places MIDI messages in an output queue, and only once every 10 msec (based on the kernel HZ variable) is the accumulated output sent to the hardware device. Using the 4Front MIDI driver mentioned above is one solution to this problem, as this driver achieves maximal MIDI throughput on the loop benchmark.[5]

Thus, for real-time performance the particular MIDI hardware interface and driver need to be tested; inclusion of the loop benchmark test with the FTAP distribution provides users with the capability of doing this.

[5]The output polling behavior of the standard Linux MIDI driver does not appear to be documented, and has apparently not been of major concern to the Linux music community – a 10 msec time difference in auditory signals, although detectable in idealized laboratory situations (Hirsh, 1959), may be less salient under more complex auditory conditions. It is worth noting that diagnosing the problem with the standard Linux driver would not have been possible were Linux not open source. It is also worth pointing out that the 4Front driver appears to use busy waits in the kernel for high-resolution output timing control on MIDI cards without interrupt capability. In addition to the solution of using the 4Front driver, the open source Advanced Linux Sound Architecture drivers (http://www.alsa-project.org) are claimed to have full millisecond precision for both MIDI input and output on a wide range of hardware.

Conclusion

Standard Linux provides facilities for real-time programming that are fully adequate for timing precision at the millisecond level or better. This has been demonstrated by tests of the Linux system calls themselves, and by tests of an implemented Linux-based experiment program. A novel feature of the FTAP program is the inclusion of performance tests which any user can run; this allows experimenters to validate program performance on their own system and hardware.

Although Linux provides support for real-time programming, the ability to achieve millisecond resolution is also greatly facilitated by developments in computer hardware. Performance issues of major concern in the 1980's and early 1990's have largely been mitigated by increases in CPU speed (as well as the ready availability of large amounts of RAM). For example, with a 500 MHz CPU chip there are 500,000 hardware clock cycles *per millisecond*. Although a clock cycle does not correspond to a single machine-level instruction (much less a line of code in a high-level language), the large amount of processing that can be completed in a millisecond means that CPU usage is a relatively minor concern in real-time programming for psychology experiments.

The view that multi-tasking operating systems are inadequate for real-time programming is thus incorrect, at least in the case of Linux. I am not claiming here that Linux is a better operating system for real-time experiments than alternatives such as DOS or MacOS, but this paper has shown that Linux is a suitable platform for such work (and one which can run on a wide range of hardware). More generally, the explicit description of the performance tests of both the operating system itself and the application program (and the ability for any user to replicate those tests) may serve as a useful standard for documenting the performance of any purported real-time systems in psychological research.

References

Aschersleben, G., & Prinz, W. (1997). Delayed auditory feedback in synchronization. *Journal of Motor Behavior*, *29*, 35-46.

Cohen, M. M., & Massaro, D. W. (1994). Development and experimentation with synthetic visual speech. *Behavior Research Methods, Instruments, and Computers*, *26*, 260-265.

Finney, S. A. (1997). Auditory feedback and musical keyboard performance. *Music Perception*, *15*, 153-174.

Finney, S. A. (1999). *Disruptive effects of delayed auditory feedback on motor sequencing.* Unpublished doctoral dissertation, Brown University.

Finney, S. A. (2001). FTAP: A Linux-based program for tapping and music experiments. *Behavior Research Methods, Instruments, and Computers*, *33*, 63-72.

Finney, S. A., & Warren, W. H. (2000). Delayed auditory feedback and rhythmic tapping: Evidence for a critical interval shift (manuscript submitted for publication).

Hirsh, I. J. (1959). Auditory perception of temporal order. *Journal of the Acoustical Society of America*, *31*, 759-767.

Murrel, S., & Kowalski, T. (1980). A real-time satellite system based on UNIX. *Behavior Research Methods and Instrumentation*, *12*, 126-131.

Myors, B. (1998). A simple graphical technique for assessing timer accuracy of computer systems. *Behavior Research Methods, Instruments, and Computers*, *30*, 454-456.

Myors, B. (1999). Timing accuracy of PC programs running under DOS and Windows. *Behavior Research Methods, Instruments, and Computers*, *31*, 322-328.

Pavel, M. (1982). Introduction to UNIX. *Behavior Research Methods and Instrumentation*, *14*, 135-136.

Perlman, G. (1982). Experimental control programs for the UNIX operating system. *Behavior Research Methods and Instrumentation*, *14*, 417-421.

Ritchie, D. M., & Thompson, K. (1978). The UNIX time-sharing system. *The Bell System Technical Journal*, *57*, 1905-1929.

Rubini, A. (2000). Making system calls from kernel space. *Linux Magazine*, *November*, 84-93.

Torvalds, L. (1999). The Linux edge. In C. DiBona, S. Ockman, & M. Stone (Eds.), *Open sources: Voices from the open source revolution.* Newton, MA: O'Reilly.

Ulrich, R., & Giray, M. (1989). Time resolution of clocks: Effects on reaction time measurement – good news for bad clocks. *British Journal of Mathematical and Statistical Psychology*, *42*, 1-12.

Wing, A. (1977). Perturbations of auditory feedback delay and the timing of movement. *Journal of Experimental Psychology: Human Perception and Performance*, *3*, 175-186.

Wing, A., & Kristofferson, A. (1973). The timing of interresponse intervals. *Perception and Psychophysics*, *13*, 455-460.