

# **K42's Performance Monitoring and Tracing**

## **Infrastructure**

**Jonathan Appavoo**

**Marc Auslander**

**Dilma DaSilva**

**David Edelsohn**

**Orran Krieger**

**Michal Ostrowski**

**Bryan Rosenburg**

**Robert W. Wisniewski**

**Jimi Xenidis**

*K42's event tracing infrastructure provides for correctness debugging, performance debugging, and performance monitoring of the system. The infrastructure allows for cheap and parallel logging of events by applications, libraries, servers, and the kernel. This event log may be examined while the system is running, written out to disk, or streamed over the network. Post-processing tools allow the event log to be converted to a human readable form or to be displayed graphically.*

## **1. Motivation of Infrastructure**

We achieved several goals with the tracing infrastructure in K42. The goals are to:

1. Provide a unified set of events for correctness debugging, performance debugging, and performance monitoring.
2. Have the infrastructure always compiled into the system thereby allowing data gathering to be dynamically enabled.
3. Separate the collection of events from their analysis.
4. Allow events to be efficiently gathered on a multiprocessor.
5. Have minimal impact on the system when tracing is not in use and allow for zero impact by providing the ability to "compile out" events if desired.
6. Provide cheap but flexible collection of data for either small or large amounts of data per event.

We discuss the motivations of these goals and the trade-offs we made in implementing them in Section 1.1, Section 1.2, and Section 1.3.

In addition to a unified tracing infrastructure, there are other performance evaluation mechanisms that can be used to understand the operating system and machine performance. These fall into two classes:: 1) operating system counters/record keeping data, e.g., the number of page faults in a given process and 2) hardware counters, e.g., cache misses. Both of these mechanisms are outside the scope of the tracing facility, though trace events may be written to log information gathered by such counters.

### **1.1. Flexible and unified events for achieving goals 1 and 3**

Other operating systems have used different infrastructures to gather events for correctness debugging as opposed to performance debugging or monitoring the system. Some had multiple ways to gather information for either or both of the latter. There is a trade-off in determining the number and kind of tracing facilities to incorporate. If the tracing facility is tailored to the specific needs of what it will be used for, then it will more closely match requirements of that use (correctness debugging for example). However, we believed that a unified and flexible system could provide the same capabilities while yielding advantages. Multiple systems for gathering events have a couple of disadvantages. In places where an event is important to multiple systems, more than one event needs to be logged. Also, to log that event correctly at each trace point in the code, the programmer has to know which system the event is intended for. This places an unnecessary burden on the programmer and can lead to errors. In code with multiple trace systems, it is typical to find more than one event logged at many places throughout the code causing additional negative performance implications. Multiple trace logs complicate post-processing tools. Designing one flexible efficient system provides better performance, is simpler to code and simpler to update, and yields more understandable trace logs. With the K42 tracing infrastructure, the programmer logs all important events to a unified trace log, and separately the analysis tools decide the events that are important to display.

### **1.2. Variable sized events for achieving goal 6**

Each event that is logged in K42 can be of a different size. There are trade-offs when deciding whether to use a fixed-size event or variable-sized event. Fixed-size events allow for simpler logging and reading out as the consumer of events always knows the starting point in memory for an event. This allows valid bits to be used allowing invalid events to be skipped (since the consumer knows the beginning address of the next event - explained in greater detail later). Fixed events allow easy random file access aiding reading and displaying large files. They allow an event displayer to skip to events later in the file without having to read the intervening portion of the file. The disadvantages of fixed sized events are that: they are wasteful in space, in many cases require longer to log because extra data needs to be written for short events, and, they make it complicated to log larger data for a given event. In K42 we have chosen to use variable sized events, but we ensure that an event always starts on a predetermined alignment boundary. For example, our current scheme makes sure that at 8K boundaries in a file an event is guaranteed to start. In this way we can handle garbled events, and we can provide close to random file access. Thus, we get the advantages of both fixed and variable sized events. This does imply that filler events need to be added if an event is about to be logged that would cross this boundary, but they are only one word, and statistically (at least thus far in our system) waste very little space. Most of the events in K42 are two or three (64 bit) words, with some events only one word. It therefore turns out that thirty to forty percent of the time events end at exactly our 8K boundary. This technique of using variable sized events, but guaranteeing event boundaries, provides the advantages of quick logging and simple logging of larger events, while still allowing fast random access to a file.

### **1.3. Achieving goals 2 and 4**

Trace events are divided up into major and minor classes. By allowing a maximum of 64 major classes, we can perform a single comparison of a major class bit against a trace mask variable to

determine whether an event should be logged. The major id is a constant value, and since the trace mask variable is frequently referenced it will remain hot and we will not suffer any cache misses. This provides for an inexpensive method for determining whether to log an event. To run well on a multiprocessor all the tracing data structures that are frequently referenced are stored in processor-specific memory. K42 processor-specific memory guarantees only processes running on the processor where the memory was allocated from will access the memory. This allows all accesses to trace structures on separated processors to be independent and thus scale well.

## **2. Description of Infrastructure**

Events in K42 are part of the code and added by developers during implementation. Events are logged into a circular buffer in monotonically increasing timestamp order. The events represent actions or positions in the code deemed important by the developer. Examples include context switch, page fault, user-mode program start, etc. Each event can have data associated with it. These events can be read out of the buffer and written to disk or sent out over a network stream. They are then analyzed and displayed by a tool.

### **2.1. Using events in programming**

For efficiency, events are divided up into a maximum of 64 major classes. Events that are related to a common purpose are placed in the same major class. For example, events associated with the memory management infrastructure are placed in the `traceMem` major class. There are currently 15 major classes in K42 allowing ample room for additional classes. Events are added to the K42 source with the use of a set of macros associated with a major class. These events are placed statically in the code, there is currently no capability to add a new event once the system is running. Adding an additional event to a major class causes the recompilation of all files that use trace events from that class. Since events from major classes are associated with a portion of the system, the addition of an event will generally trigger only local recompilation. The addition of a major class affects all files that use tracing and triggers a significant system wide recompilation. It is not common to add a new major class.

### **2.2. Performance Implications**

Events affect performance on three levels: no effect, negligible effect, small effect. It is possible to "compile out" events by major class, thus eliminating any effect these events have on performance. This is not recommended. The infrastructure is designed to be cheap enough to leave all the events in, but disabled if there is a performance concern. This way the events can all be dynamically enabled. The cost of checking to see if a major class is disabled is a check of a mask variable against a constant. Since the mask is frequently accessed, it will be cached, making the check to determine whether a major class is being logged inexpensive. For trace events at the lowest levels in the kernel, we have written assembly that is 3 instructions (to check the mask and determine whether or not to log the event) plus 21 instructions (to log the event if necessary) for a total of 24 instructions. We use efficient C code for the rest of the trace events. These are the direct impacts on performance. There are a couple of second order effects such as I-cache pollution and additional code that is introduced by the tracing system but more difficult to directly measure.

Part of the important aspect of determining how expensive it is to log events is the cost of reading a fine granularity clock. On PowerPC, as with many architectures, this clock is accessible from user space. Architectures that don't provide the functionality of a cheap clock-read from user space will suffer a performance hit when accessing the clock to obtain the timestamp to log user events. This is something that needs to be kept in mind when considering the tracing infrastructure on different platforms.

We used a 32-bit field to log the value of the clock. At 1GHZ 32 bits wrap in a little over four seconds. We chose to use only 32 bits to save significant space in both the memory image as well as the file. The result of this decision is that the analysis tools have to understand timer wraps. Further, there is an unlikely scenario in which no event is written for the entire four plus seconds, and thus time would be lost. To avoid that we write a heartbeat event at least once per timer wrap.

### **2.3. Control of Tracing**

Associated with each per-processor buffer are two per-processor structures. The `traceControl` structure is writable by both the kernel and user, and primarily contains a pointer into the `traceBuffer` to where the next event is to be logged. The `traceInfo` structure contains a set of fields that define characteristics of the given trace setup such as the number of buffers in the array, and the physical location of beginning of the array. The important field the `traceInfo` contains the mask that indicates which major events should be traced. The mask could have been in the control structure thereby allowing the user to write it as well. However, we wanted the ability to log events at exception level in the kernel. To accomplish this, the mask is mirrored in a structure accessible at this level. To guarantee consistency between these two structures, a call must be made to the kernel to modify the tracing mask. Since this is an infrequent event it poses no performance difficulties.

The mask is used to determine the set of major events that are currently being logged. K42 events are divided up into a maximum of 64 major IDs. These IDs are intended to represent major subsystems of K42. For example, `TRACE_MEM` is used for events in the memory sub-system such as a page fault, `TRACE_PROC` for events associated with processes, etc. At each place in the code where an event is logged, the tracing macros perform a quick check against this mask to determine if logging for this particular major ID is enabled. This mask is dynamically settable. The tracing infrastructure also allows a similar operation to be performed on major ID classes that use the data field of the first word of the trace event for minor IDs (currently all major classes in the system). Currently the only way we control which events are logged is to enable and disable events by majorID. When tracing is enabled on a processor, all events generated on that processor are logged. While we envision adding support for the ability to enable tracing on a per-address-space basis, currently all processes on a given processor are traced.

### **2.4. Memory for Tracing and Protection Issues**

There is a single, shared, writable, buffer between the kernel and all the user address spaces on a given processor. This is important in K42 as many system services, such as the file system and name server, have been moved into user space. A single buffer provides a unifying place for all these events. In addition to the shared buffer, there are also structures that are writable by both the user and kernel. We have taken care to ensure that if a user enters invalid data

in the tracing structures, then that user can not cause another process (including the kernel) to crash or perform "wild writes" outside of the trace buffer. A process can, however, scribble garbage throughout the trace buffer invalidating data written by other processes. Also, when tracing is enabled, processes can discover significant information about the activities of all the other processes on its processor. As mentioned above we are considering allowing tracing to be enabled on a per process basis.

## 2.5. MP Issues

To achieve good performance on a multiprocessor, there is one trace buffer per processor. Events logged on a given processor are written to the buffer associated with that processor. This avoids cache lines ping-ponging between different processors and ensures fast access to memory on the processor when the line is not in the cache. To synchronize data from different processors during post-processing analysis, we assume a synchronized clock across the complex. In most architectures this is provided by hardware. For the others, a software synchronized clock must be maintained. In K42 this clock is used for other reasons as well thus the tracing infrastructure creates no additional demands. A separate buffer per processor implies that if a post-processing tool wishes to produce a single stream there is additional work to do. In reality, most tools display the streams from different processors next to each other since understanding the semantics of a single stream with multiple real simultaneous events is more difficult than seeing them visually next to each other on the same graph. There is also no processor number written into any event, it is implicit in the stream.

## 2.6. Using Tracing

In this section we present an overview of how tracing is used in K42. A step by step example of how to insert a new trace event, or a new trace class can be found in `kernel/trace/trace.H`.

All trace structures, variable names, and macros start with `trace`. An example trace event in K42 is:

```
traceMem2(TRACE_MEM_REG_CREATE_FIX, vaddr, size);
```

The macro (actually inlined function) is composed of three parts: the word `trace`, the major class (`Mem`), and the number of data entries to be logged, which in the above example is two. The majority of trace events logged contain one or two data words, but the tracing infrastructure allows any number of data words to be written. All trace events in K42 have a unique constant representing a particular event. The constant names start with `TRACE` followed by their major class.

The "Mem" in the above example is due to the fact that the example trace event was taken from the memory subsystem. Trace events are divided up into major classes based on the subsystem they are recording data for. While this is not a correctness issue it allows all `Mem` events, for example, to be enabled or disabled simultaneously. The tracing infrastructure does allow for events to be enabled and disabled by minor classes, but we have no existing use of that currently in the system. Some of the major classes defined in K42 include: `Alloc`, `IO`, `Lock`, `Scheduler`, `Mem`, `Exception`, `Proc`.

For a given major class, say `Xxx`, there are several macros available. As mentioned above there are `traceXxx0`, `traceXxx1`, though `traceXxx6`. Creating macros up to 6 is just a default and any

major class that needs more may do so. In addition to efficient macro events where the number of data words is known, there are three more generic macros: `traceXxxBytes`, `traceXxxStr`, and `traceXxxGeneric`. The "Bytes" macro takes a pointer and a length and writes that many bytes into the tracing stream (because tracing always logs 8 byte entries, any remaining bytes up to the next 8 byte boundary are not defined). The "Str" macro is for convenience when the data is NULL terminated and it's length unknown. Finally the "Generic" macro takes any number of data words followed by any number of strings.

When a programmer creates a new trace event they also make an entry into a `traceXxxEventParse` array. The Parse structure defines how this event should be interpreted by post-processing tools. For example, the matching parse entry for the above event is:

```
{ __TR(TRACE_MEM_REG_CREATE_FIX), "64 64",
  "Region default created fixlen "
  "with address %0[%llx] and size %1[%llx]" },
```

More details appear in `kernel/trace/trace.H`, but briefly the `%0` matches the first 64 bit quantity and means to print out the first data word as a long long hex value. The `%0` and `%1` could have been switched allowing the data words to be printed out in any order. Further, if smaller quantities are packed into the data words then the 64's could be replaced with 32's, 16's, 8's, or Str's as appropriate. The language is simple but allows enough flexibility for the event creator to define the intended output format and information to the post-processing tools.

The three simple steps that need to be performed to add a new trace event are: add a minor ID in `traceXxx.H`, make the matching entry in `traceXxxEventParse`, and add the event macro in the code. The tracing infrastructure can be enabled or disabled while the system is running. There is a mask value that is defined in the per-processor `traceInfo` structure. The mask is a 64-bit quantity that determines the major classes that are currently enabled. Each macro resolves to an initial cheap check against the mask value to determine if tracing for that class is currently enabled. This value may be modified by using the `SystemMisc` interface. Individual classes may be enabled or disabled by or'ing or and'ing the appropriate values, or tracing across the system may be enabled or disabled by clearing or setting the mask. While the mask is stored in a per-processor structure, the `SystemMisc` interface also allows the request to affect `traceInfo` structures across all the processors or a set as specified by the caller.

From the console kernel prompt the user may request events be printed to the console. The choices allow the last `n` events, or events of a given major class, or a variety of other options. In addition, when the kernel crashes a memory dump of the tracing buffer may be gathered and fed to a tool that will print the events leading up to where there kernel was.

## 2.7. Implementation Details

In this section we'll provide a brief overview of the details; more detailed information can be found in the documentation files in the two directories that contain the tracing infrastructure in K42: `libc/trace` and `kernel/trace`. Also see `kernel/trace/trace.H`.

A trace event is divided up into a 64-bit header followed by its data. The first word contains a 32 bit timestamp, the length of the event, the major ID of the event, and data generally used as the minor ID of the event. While the length of a trace event comes in 8 byte increments, 32 bit or smaller quantities of data may be packed into a word, and the tracing infrastructure has macros that support these operations.

All macros are defined in an include file named for the subsystem they are a part of. The name is consistent with the major class. For example, in `traceMem.H` there are definitions of macros for the `TRACE_MEM_MAJOR_ID`. Major class definitions, constant definitions, macros for processing trace events, structures, etc are found in `kernel/trace/trace.H`. There is also a description of the language used to print out the data from tracing events in this file. While most trace events appear in K42 architecture-independent code, there is nothing that precludes them from being used in architecture-specific code as well.

All the major class macro definitions resolve down to a `traceDefault` implementation. For example a `traceMem` macro will check to see if memory subsystem tracing is on with a cheap mask check, and if so call the matching `traceDefault` function.

Trace events are entered into the shared buffer via non-blocking atomic operations. The code sequence involves atomically reserving a number of entries (64-bit quantities) in the buffer, writing data to the buffer, and (optionally) indicating the data was written. In the normal fast-path code (not the last event in a buffer) a timestamp is retrieved and an attempt made to reserve a position in the buffer based on the recorded value of the current buffer pointer. This is performed via a non-blocking `CompareAndStore` operation that simultaneously reserves a place in the buffer for the current event and updates the buffer pointer. Once this operation successfully returns, the tracing code has reserved a place in the buffer for event it is logging and proceeds to log it into the buffer. Because the space is reserved, i.e., the next event will be logged in memory starting after the length of this event, it is not a problem if the tracing code is preempted. Though note that if the process running the tracing code is killed there is an error condition that needs to be handled. This algorithm eliminates the need for locking thus preventing priority inversion and making the logging of tracing events efficient.

### 3. Tracing Daemon

K42 provides a daemon that will read the information logged into the trace buffers and write it out either to a file or over a network stream. This daemon coordinates with the logging infrastructure via fields in the `traceControl` structure. The daemon wakes up periodically (adjustable depending on the frequency of logged events) and writes out buffers that have been filled in by the trace logging. Under normal operation, the daemon writes all the buffers that are available that have been completed. There are a couple of problems that can occur. If the logging of events is occurring very rapidly, the daemon can get a full buffer behind, and the logging facility can "lap" the daemon. The daemon can detect this situation by checking the current index against the number of buffers it knows it has written. In the situation where the daemon is lapped, it writes this information into an event indicating the amount of lost data so that analysis tools are made aware of what happened. Under normal operation there are no lost events.

A second and more complicated error can occur. If a user was in the process of writing a trace event, having started but not yet completed logging and then is interrupted, it is possible a long period of time goes by before the user finishes writing that event. It is also possible the user program is killed before finishing that event. Variable-sized events complicate this story as there is no way of determining where the next event starts in the situation that an event has not been completed. An early attempt to deal with this situation involved counting the amount of data that had been written to the entire buffer on the logging side and not writing the data out until the amount of data logged equaled the size of the buffer. Unfortunately, this does not solve

the problem in the general case and adds significant complexity to the code. It was therefore decided to use the heuristic of the daemon remaining two buffers behind where the logging facility was writing events. In all but the most extreme conditions, this allows any event to be logged before the daemon attempts to write out that event. The analysis tools, however, should be written to deal with incompletely logged events.

The tracing daemon is implemented in `os/servers/traced`.

## 4. Analysis Tools

Currently, K42 provides three simple ways of viewing the events that have been logged. Inside the kernel, accessible from the test prompt or debugger, there is tool that allows the user to print out events. The user may choose to print out events from the current buffer, the previous buffer, a certain number of events, or several other criteria. These events are printed to the console and are from the buffer on the processor that has control. A second tool called **traceTool** is built and installed. This tool will take a binary trace file and using the `traceParse` structure will produce ASCII human-readable output on standard out. A third GUI tool called **kmon** (K42 performance monitoring tool), written in java, has a series of useful visualization capabilities allowing the user to understand in time the behavior of the system.

Below is some sample output from our `traceTool`. The first event is the start of a user level process. For demonstration purposes this is a shell starting. The format of the output is the time in the first column followed by the constant defining the trace event that occurred. The right column represent the output the programmer (that added the event) has indicated should be printed with this event. Note: in displaying postscript, this column wraps for a few of the events.

```

21.474735000 TRACE_USER_RUN_UL_LOADER          process 6 created new process with id 7 named
21.474742200 TRACE_EXCEPTION_PGFLT            PGFLT, kernel thread 80000000c12b0f90, fault-
Addr 405e628, commID 600000000
21.474788200 TRACE_EXCEPTION_PGFLT_DONE        PGFLT DONE, kernel thread 80000000c12b0f90, f
Addr 405e628, commID 600000000, rc 0
21.474809100 TRACE_EXCEPTION_PPC_CALL          PPC CALL, commID 0
21.474853000 TRACE_MEM_FCMCOM_ATCH_REG        Region 800000001022cc98 at-
tached to FCM e100000000003f30
21.474870900 TRACE_MEM_FCMCRW_CREATE                TRACE_MEM_FCMCRW_CREATE ref e100000000003f90
21.474914200 TRACE_EXCEPTION_PPC_RETURN        PPC RETURN, commID 600000000
21.474924700 TRACE_EXCEPTION_PPC_CALL          PPC CALL, commID 0
21.474957300 TRACE_MEM_REG_CREATE_FIX          Region default 10000000 cre-
ated fixlen with address 113000 and size cccd24d01020014
21.474977300 TRACE_MEM_REG_DEF_INITFIXED        region default init fixed 80000000102b7c00 ad
21.474987300 TRACE_MEM_ALLOC_REG_HOLD          alloc region holder addr 10000000 size 113000
21.474996200 TRACE_MEM_ALLOC_REG_HOLD          alloc region holder addr 10000000 size 113000
21.475029300 TRACE_MEM_FCMCOM_ATCH_REG        Region e100000000003fa0 at-
tached to FCM e100000000003f90

```

Below is a screen capture of the GUI tool called `kmon`. `kmon` is a graphical tool that displays a binary trace file. The trace file is taken from an entirely empty system except for the start of a

shell and the subsequent run of a "hello world" program and then the exit of a shell. Some of the menus and tool boxes are left open on the window for display. One of the tool boxes allows the user to display events. For demonstration purposes we have chosen events to be displayed when a program starts and when a program ends. The start event is indicated by the event `TACE_USER_RUN_UL_LOADER` and the end event by `TACE_USER_RETURNED_MAIN`. In the top part of the tool there is a count of the number of times the event occurs. The middle portion of the display (the portion that is mostly red) is a timeline showing all the events that occurred during the approximately five and half seconds of this run. In particular the events we have selected to be displayed (user start and stop) are displayed - the user start in green and the user stop in pink.

The window on the bottom left displays all the different PIDs, their names if the file contains them or their PIDs, and the settable color they are currently being displayed in. Also in the background is the window we used to select to show events. The window in the foreground is the pull-away Show menu demonstrating the different options available to the user. Finally the white window in the background is for displaying more detailed information, such as the statistics option - or an ASCII output of the events being viewed. There are many more options available that can be found with documentation of the tool.

The analysis tool are in `tools/misc/kmon`.

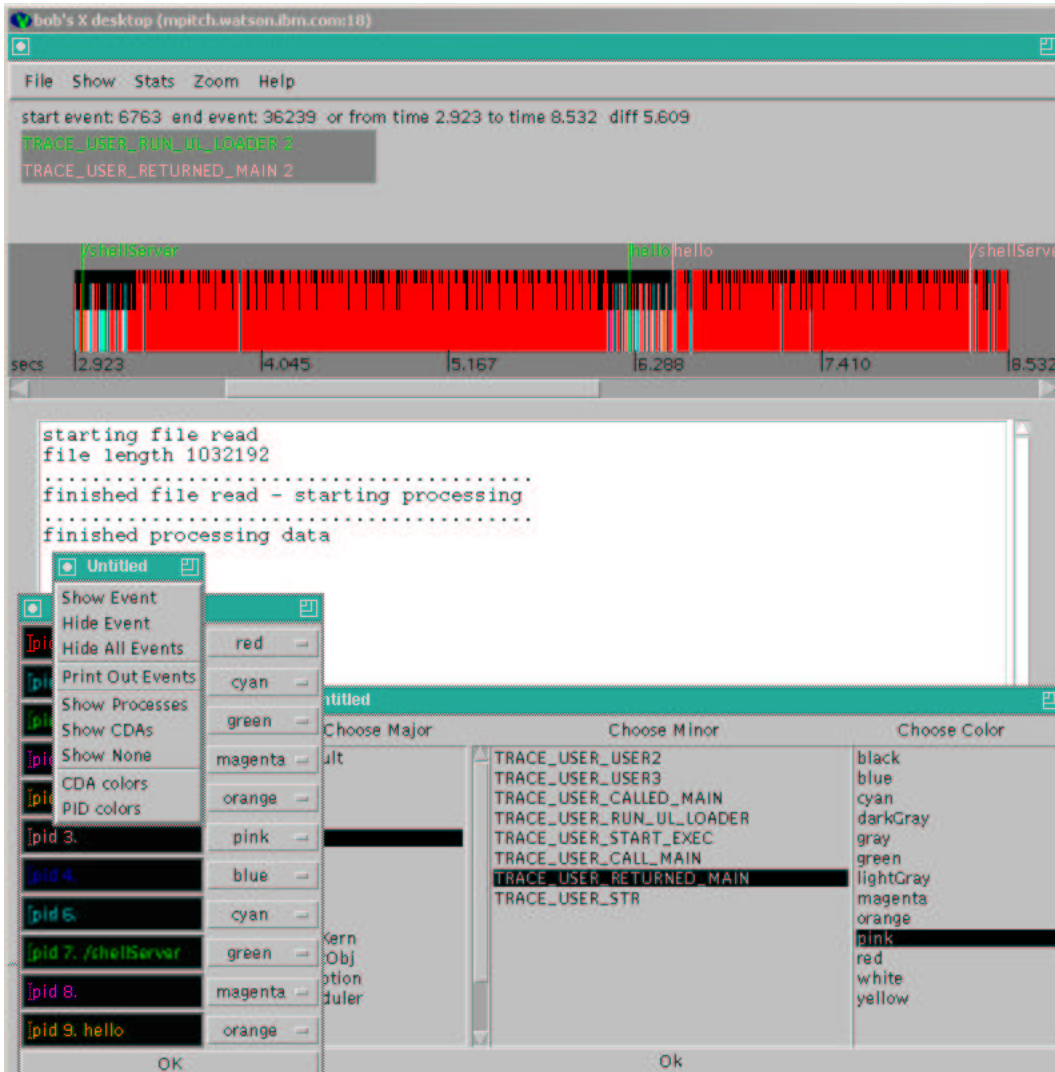


Figure 1. Screen capture from kmon trace display tool