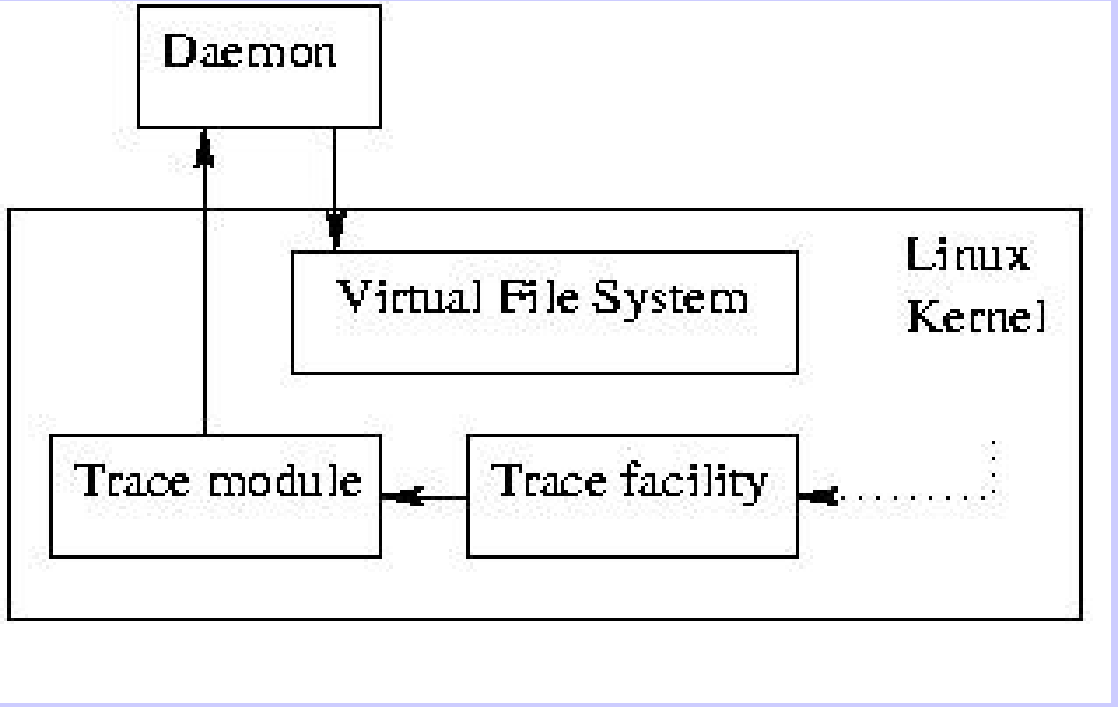
LTT (Linux Trace Toolkit)

http://www.opersys.com/LTT

Goals

- fully-featured tracing system for the Linux kernel
- Includes:
  - kernel components required for tracing
  - user-level tools required to view the traces
- Elements:
  - A **modified linux kernel** that enables events to be logged.
  - A **linux kernel module** that takes care of storing the events into its buffer and then signals the trace daemon when he's reached a certain limit of data.
  - A **daemon** that writes the data collected by the kernel module.
  - An **analysis tool**.

General operation diagram



Data collection architecture: Kernel instrumentation

- Core:
  - System call entry
  - System call exit
  - Trap entry
  - Trap exit
  - Interrupt entry
  - Interrupt exit
  - Scheduling change
  - Kernel timer
  - Bottom-half
- Non-Core :
  - Process
  - File-system
  - Timer
  - Memory
  - Socket
  - IPC
  - Network

Data collection architecture: Trace module

- Retrieves additional event information (time-stamp and CPU-ID)
- Filters events according to configuration
- Records events in trace buffer
- Provides daemon with a unified entry ponit to configuration and control

Data collection architecture:
Trace daemon

- Provides the user with full control of tracing process
- Configures trace module according to user options
- Reads /proc content at trace start to record system's initial state
- Commits trace buffers when full

Data collection architecture: Analysis tool

- Uses raw trace and system's state to reconstruct system behavior
- Uses event trace to extract system-wide and per-process statistics
- Presents user with graphic and text system behavior reconstruction
- Provides the user, when in graphical mode, with an interface to easily browse and analyze traces

Toolkit implementation

Kernel trace facility	=>	linux/kernel/trace.c
Trace statements	=>	inlined macros
Trace module	=>	syscall
Trace daemon	=>	Unix daemon
Analysis tool	=>	Unix application

Impact

- Given its high degree of precision it would be expected that LTT would bear a high cost in system performance. That is not the case. Practical tests have shown that when tracing core kernel events the **impact remains lower than 2.5%**.

- On the other hand, t**he size of the generated traces can be large**. Tracing a system on which a GUI such as KDE or Gnome is running can yield 0.5MB of trace per second. A "plain" system will typically yield less than 0.1MB of trace per second. The size of the traces can be greatly reduced by tracing only the events necessary to the underlying analysis.

Toolkit usage: Tracing the system

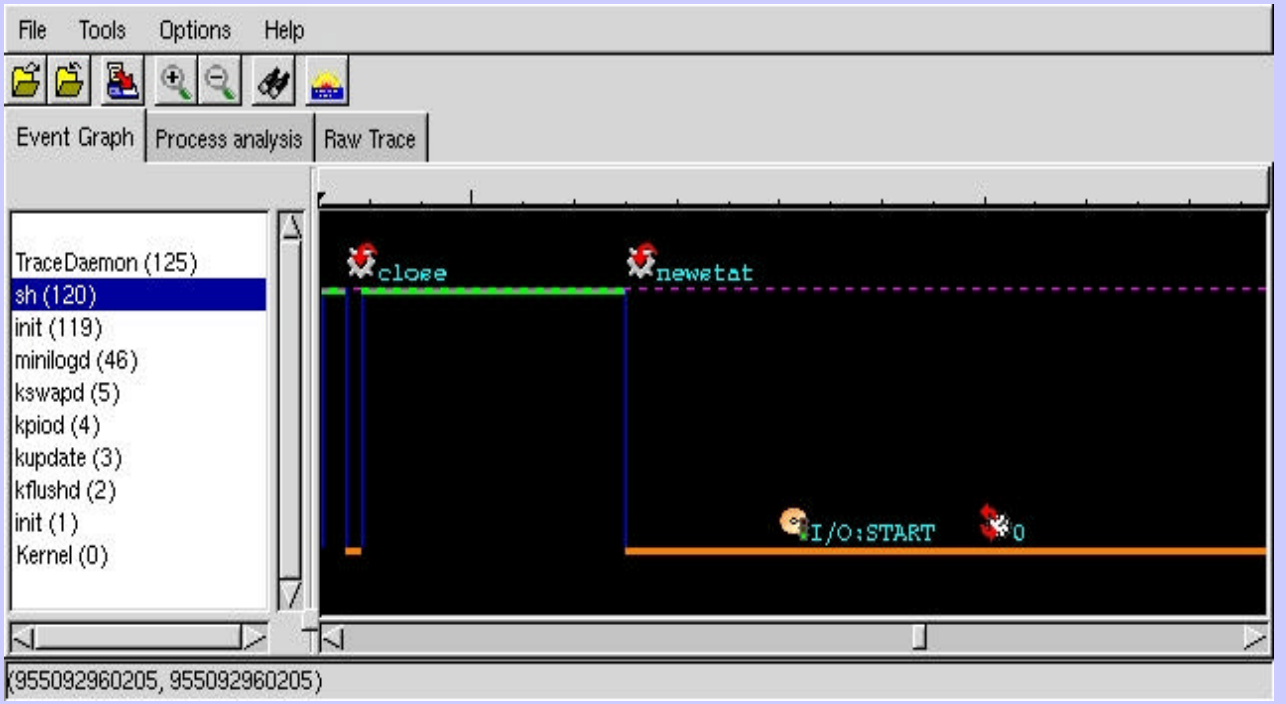
- To generate a trace, the system must be running a patched kernel. To start tracing, the trace daemon is launched with the appropriate options. Here is an example trace daemon launch command:

  tracedaemon [opts] out.trace out.proc

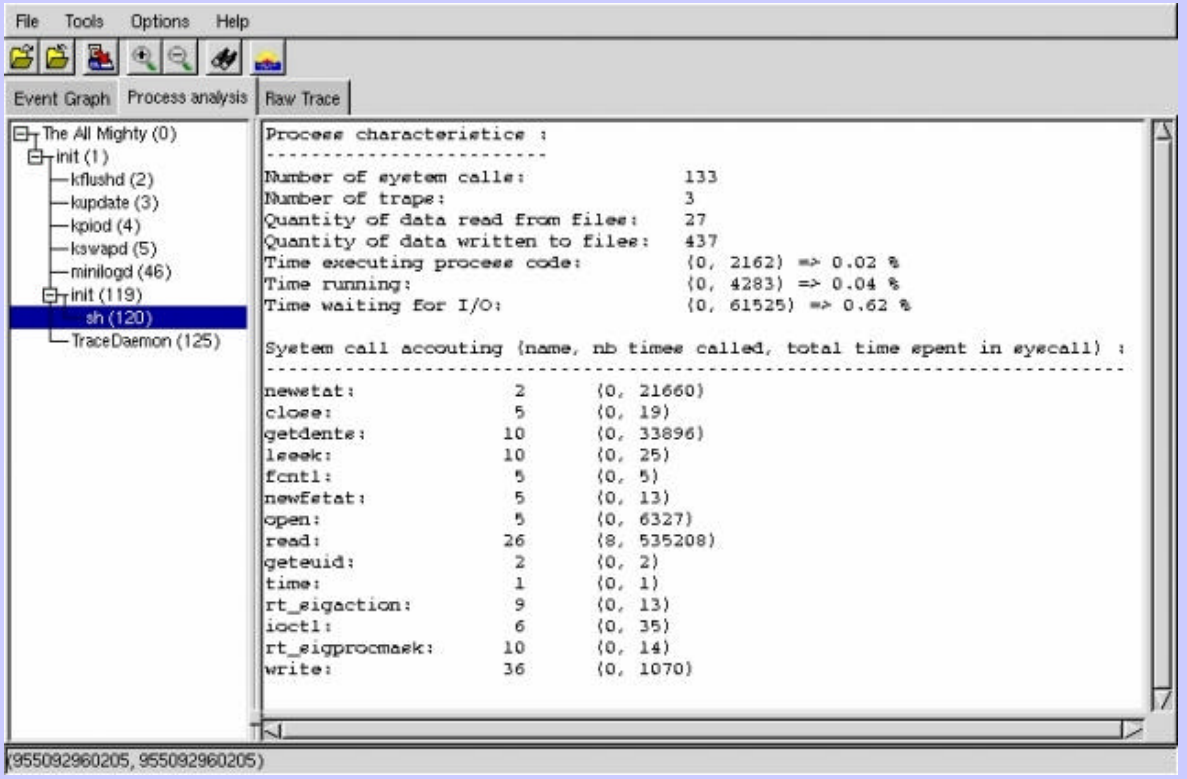
- In order to facilitate the usage of the trace daemon, some scripts are provided. The following command line is an example usage of such a script:

  trace 30 out

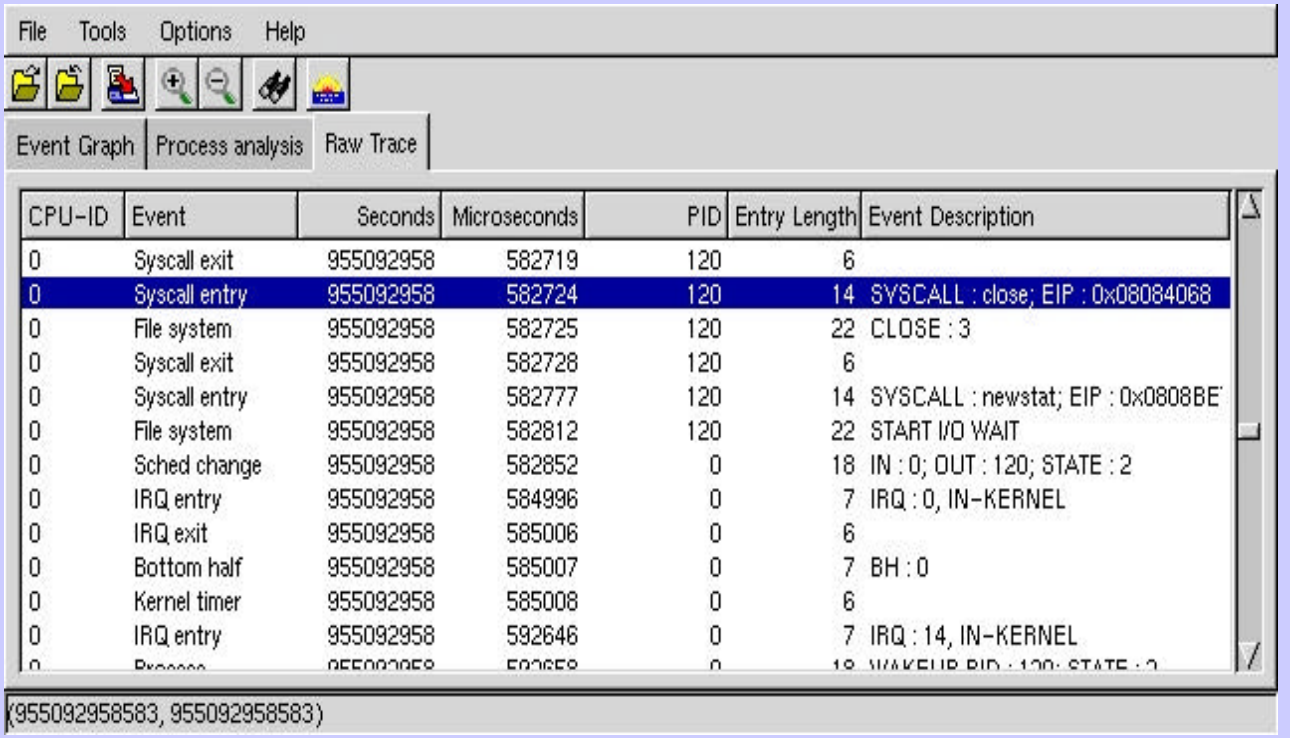
Viewing Traces



Viewing Traces



Viewing Traces



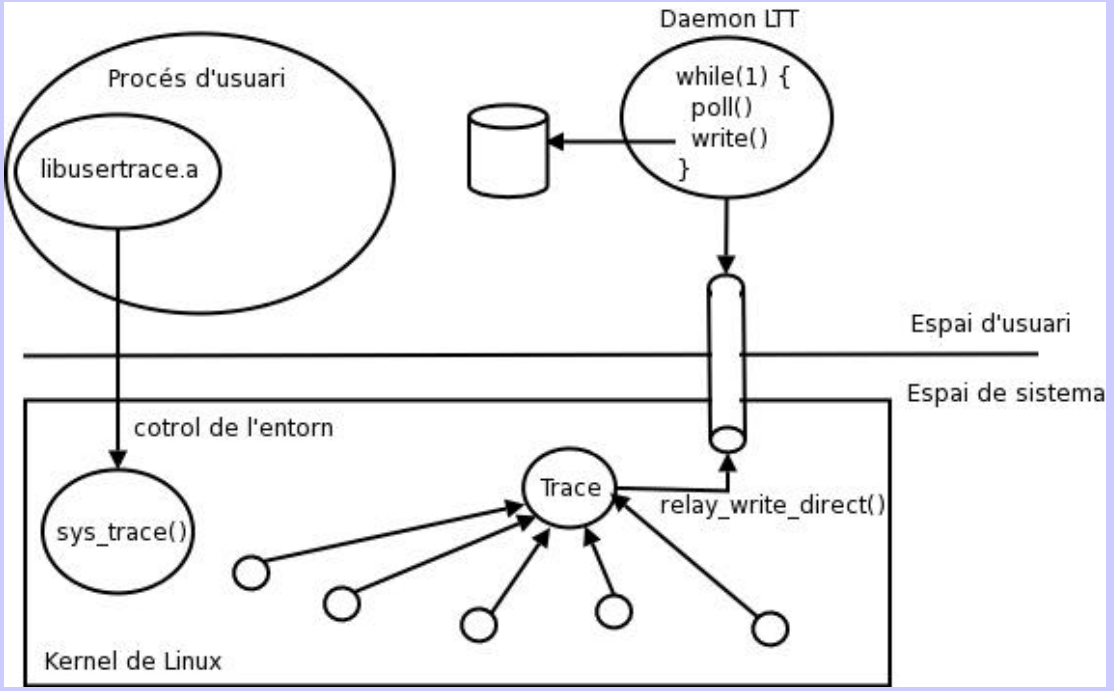
CPU-ID	Event	Seconds	Microseconds	PID	Entry Length	Event Description
0	Syscall exit	955092958	582719	120	6	
0	Syscall entry	955092958	582724	120	14	SYSCALL : close; EIP : 0x08084068
0	File system	955092958	582725	120	22	CLOSE : 3
0	Syscall exit	955092958	582728	120	6	
0	Syscall entry	955092958	582777	120	14	SYSCALL : newstat; EIP : 0x0808BE
0	File system	955092958	582812	120	22	START I/O WAIT
0	Sched change	955092958	582852	0	18	IN : 0; OUT : 120; STATE : 2
0	IRQ entry	955092958	584996	0	7	IRQ : 0, IN-KERNEL
0	IRQ exit	955092958	585006	0	6	
0	Bottom half	955092958	585007	0	7	BH : 0
0	Kernel timer	955092958	585008	0	6	
0	IRQ entry	955092958	592646	0	7	IRQ : 14, IN-KERNEL

RelayFS

- As the Linux kernel matures, there is an ever increasing number of facilities and tools that need to **relay large amounts of data from kernel space to user space**

- The main idea behind the relayfs is that every data flow is put into a separate "**channel**" and each channel is a file. In practice, **each channel is a separate memory buffer** allocated from within kernel space upon channel instantiation

Details: overview



Details: the event structure

```
int event_struct_size[TRACE_EV_MAX + 1] =
{
        sizeof(trace_start),
        sizeof(trace_syscall_entry),
        0,                      /* TRACE_SYSCALL_EXIT */
        sizeof(trace_trap_entry),
        0,                      /* TRACE_TRAP_EXIT */
        sizeof(trace_irq_entry),
        0,                      /* TRACE_IRQ_EXIT */
        sizeof(trace_schedchange),
        0,                      /* TRACE_KERNEL_TIMER */
        sizeof(trace_soft_irq),
        sizeof(trace_process),
        sizeof(trace_file_system),
        sizeof(trace_timer),
        sizeof(trace_memory),
        sizeof(trace_socket),
        sizeof(trace_ipc),
        sizeof(trace_network),
        sizeof(trace_buffer_start),
        sizeof(trace_buffer_end),
        sizeof(trace_new_event),
        sizeof(trace_custom),
        sizeof(trace_change_mask),
        0                       /* TRACE_HEARTBEAT */
};
```

Details: Starting

```
for(i = 0; i < num_cpus; i++) {
        sprintf(relay_file_name, "%s/cpu%d", dirname, i);
        trace_channel_handle(i) = relay_open(relay_file_name,
                                             buffer_size,
                                             n_buffers,
                                             flags,
                                             &trace_callbacks,
                                             start_reserve,
                                             end_reserve,
                                             trace_start_reserve,
                                             0,
                                             0,
                                             0);
        if(trace_channel_handle(i) < 0)
                return -ENOMEM;
    }
```

Details: preparing an event

- Choosing the channel

```
channel_handle = trace_channel_handle(cpu_id);
```

- Choosing the event:

```
if (event_id == TRACE_EV_SCHEDCHANGE) {
   incoming_process = (struct task_struct *) (((trace_schedchange *)
     event_struct)->in);
   (((trace_schedchange *) event_struct)->in) = incoming_process->pid;
}
```

Details: Writing an event

- Acquiring the channel:

```
relay_lock_channel(rchan, flags); /* nop for lockless */
reserved = relay_reserve(rchan, data_size, &time_stamp,
                         &time_delta, &reserve_code, &interrupting);
```

- Writing data:

```
relay_write_direct(reserved,  &cpu_id,  sizeof(cpu_id));
relay_write_direct(reserved, &event_id, sizeof(event_id));
relay_write_direct(reserved,  &time_delta,  sizeof(time_delta));
relay_write_direct(reserved,  event_struct,  event_struct_size[event_id]);
```

- Committing data and releasing channel:

```
relay_commit(rchan, reserved, bytes_written, reserve_code, interrupting);
relay_unlock_channel(rchan, flags);
```

Details: Controlling the module

- New system call: trace

```
if(trace(gTracHandle, TRACER_START, 0, 0) < 0)
    {
    /* Don't go any further */
    printf("TraceDaemon: Unable to start tracing \n");
    exit(1);
    }
 }
```

Details: reading from relayFS

- The daemon controls the buffers:

```
while (1) {
   if (gWaitingForThreads) // no more polling needed
     pause(); // we're now waiting on the finishing timer signal

   for (i = 0; i < gNumCPUs; i++)  {
      pollfds[i].fd = gTracRelayFile[i];
      pollfds[i].events = POLLIN;
    }

   ready = poll(pollfds, gNumCPUs, -1);
   if (ready < 0)
      continue;

   for (i = 0; i < gNumCPUs; i++)  {
      if (pollfds[i].revents & POLLIN)
        {
          read_all_ready();
          break;
        }
    }
 }
```

Details: the trace syscall

- Definition:

```
/* Definition for trace system call */
  #define __NR_trace 274
  _syscall4(int, trace, \
            unsigned int, tracer_handle, \
            unsigned int, tracer_command, \
            unsigned long, command_arg1, \
            unsigned long, command_arg2);
```

Future directions

- Dynamically insertable trace statements (DProbes)
- Dynamic event creation
- Support for multiple platforms
- Support for other operating systems
- Remote tracing
- Raw logging
- etc...