CHEMNITZ UNIVERSITY
OF TECHNOLOGY

DEPARTMENT OF COMPUTER SCIENCE
CHAIR FOR REAL TIME SYSTEMS

Profiling Linux System Call Activity

Student Research Project

Author: Sascha Dienelt
Supervisor: Dr. Robert Baumgartl
Date of Submission: 10.12.2006

Contents

1. Introduction

1.1 VisioOS – Visual Simulation of Operating Systems

VisioOS [1] is a project simulating an operating system and its hardware for educational purposes. It is divided into the simulation core and the user interface. The core consists of five components:

- Process management
- Basic memory management
- High level memory management (stack, heap and shared memory)
- File management
- External storage management

Every component is configurable by the user interface.

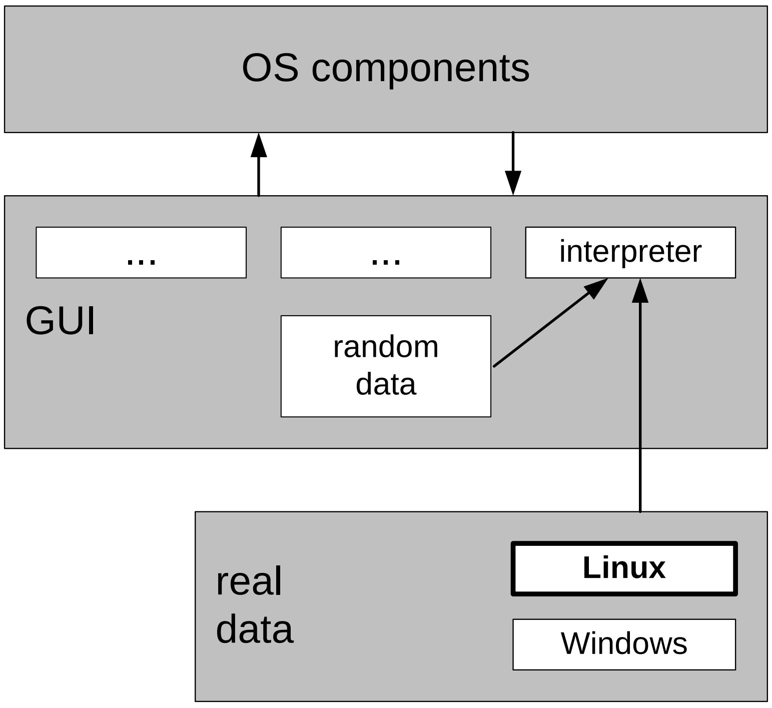


Figure 1.1 placement

The interpreter is able to send a number of commands to the OS components, so several processes can be simulated. These commands can be user defined by a C file, randomly generated or created by the run of an application on a real operating system.

1.2 Simulating real applications

To simulate the behavior of real applications in another operating system than Linux or Windows, it's necessary to record all system calls in the original system and convert the received data into information the interpreter of the simulation is able to work with.
The goal of this project is to develop a tool providing the possibility to log all system calls a process and its sons execute and also measure the number of clock cycles every process compute in user mode between the system calls. The received data should be suitable input data for the interpreter of VisioOS.

1.3 Linux system calls

A process is able to use services provided by the kernel by making a system call [2]. Every system call has a specific number. This number is written into the EAX register. There can be up to 6 parameters, which are set into the registers EBX, ECX, EDX, EDI, ESI and EBP. If a system call needs more than 6 values to pass, it has to use pointers to structures, where the additional parameters are stored.
To switch to kernel mode the software interrupt 0x80 is called and the system call handler calls the system call services routine according to the system call number found in the EAX register. The address of every system call service routine is stored in the system call table. The system call's return value is stored in the EAX register.

2. State of the art

2.1 Methods

There are different ways to gain the system calls of a process.
The first idea is to catch them in kernel mode and log the received data into a file. To act in kernel mode it's necessary to develop a driver [3]. This driver can change the system calls in a way that own functions were used instead of the original ones. This procedure is called "hooking".
An easier way is to modify the system call service routines itself and integrate some additional services into the kernel allowing to control and perform the logging process. In this method a driver must also be developed to control which process have to be logged.
The most user-friendly way would be a method just acting in user mode. Strace is such an application, which provides the possibility to log all system calls of a given process and its sons. But it doesn't offer all needed information.
Strace uses the system call Ptrace for its procedures. So this could be a good method to gain all required data.

2.2 Hooking

Hooking describes the method injecting own code in an active kernel. There are various possibilities to do this. The easiest way is to modify in kernel mode the system call table, where all addresses of the system call service routines are stored.
We have to develop a driver (because they act in kernel mode) which saves the original address and stores the address of his own function which calls the original function, makes the logging stuff and returns the value the original function returned.
The following example shows how to hook the fork system call. Unfortunately this method does work for all system calls. Mostly such a proceeding causes a kernel panic.

```
[...]

extern void *sys_call_table[]; //The system call table
int (*old_sys_fork)(struct pt_regs); //Place to store the original address

[...]

asmlinkage int hook_sys_fork(struct pt_regs regs) {
      int newpid;

      //Call the original function
      newpid=(*old_sys_fork)(regs);

      //Do logging stuff
      [...]

      //Return the value given by the original function
      return newpid;
}
static int __init mod_init(void) {

      [...]

      //Save original address
      old_sys_fork = (void *) sys_call_table[__NR_fork];
      //Store new address
      sys_call_table[__NR_fork] = hook_sys_fork;

      return 0;
}
static void __exit mod_exit(void) {

      [...]
      //Restore original address
      sys_call_table[__NR_fork] = old_sys_fork;
}
module_init( mod_init );
module_exit( mod_exit );
```

Figure 2.1 example for hooking sys_fork with sys_call_table modification

There is another method that hooks the system call service routines without changing the system call table [4]. It overwrites the first assembler instructions of the service routine with a jump to its own one. When the service routine is called the own instruction can now be executed. After that we execute the overwritten instructions and jump back to the service routine. This method is very difficult, because every service routine has its own start instructions and must be handled separately. Also the return value of the system call is unknown.

```
[...]

extern void *sys_call_table[]; //The system call table

//New code
unsigned char ab_jmpcode1[7] = "\xb8\x67\x45\x23\x01"  /* mov $address, %eax */
                                "\xff\xe0";            /* jmp *%eax */

//Old code
unsigned char ab_bcode[20]  =
"\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90" /* 13 nops */
                                "\xb8\x67\x45\x23\x01" /* mov $address, %eax */
                                "\xff\xe0";            /* jmp *%eax */

void hook_sys_execve( int  a)
{
      //Do logging stuff
      [...]

      asm volatile (
                        "mov    %ebp, %esp;" /* restore stack */
                        "popl   %ebp;"
                        "jmp    ab_bcode"
                  );
}
static int __init mod_init(void) {
      int i;
      char *ptr;
      unsigned int addr = (unsigned int) &hook_sys_execve;

      [...]

      ptr = (char *) &addr;                /* get from_jump address */
      for(i=0;i<4;i++)
            ab_jmpcode1[1+i]=ptr[i];
      ptr = sys_call_table[__NR_execve];
      for(i=0;i<7;i++) {
            ab_bcode[i]=ptr[i];            /* backup overwritten bytes */
            ptr[i]=ab_jmpcode1[i];         /* hook */
      }
      addr = (unsigned int) ptr+7;         /* get to_jump address */
      ptr = (char *) &addr;
      for(i=0;i<4;i++)
            ab_bcode[14+i]=ptr[i];

      return 0;
}
```

```
static void __exit mod_exit(void) {
      int i;
      char *ptr;

      [...]

      ptr = sys_call_table[__NR_execve];
      for(i=0;i<7;i++)
      ptr[i] = ab_bcode[i];           /* restore */
}
module_init( mod_init );
module_exit( mod_exit );
```

Figure 2.2 example for hooking sys_execve without sys_call_table modification

A big problem of hooking is that the position of the system call table must be known. Until Kernel 2.4 this address is given, but with Kernel 2.6 it isn’t exported any longer. So it has to be searched in kernel memory by looking for some patterns.

2.3 Kernel modification

Much easier is to modify the kernel code. The logging instructions are inserted directly in the system call service routines and the kernel has to be recompiled. This logging method is very fast and stable.
The kernel is extended with additional functions to:

- Add process ids to the table of logged processes
- Check if a process should be logged
- Start / stop logging
- Check if logging is enabled
- Make a log entry

To control the logging there is also a driver required.

2.4 Strace

Strace [5] is an application which writes all system calls with arguments and returned value of a process into an output stream.
But we need additional information to calculate the time the process spent in user mode since the last system call and Strace can’t be extended easily.
Strace uses Ptrace to gain its data, maybe we can use it a similar way.

2.5 Ptrace

Ptrace [6] is a system call which allows processes to observe and influence other processes if they have the right to do this. It is often used by debuggers to do a step by step execution of an application.
With this method all logging operations can be done in user mode.

2.6 Conclusion

	Kernel 2.4	**Kernel 2.6**
Hooking with system call table modification	⊞ kernel independent ⊟ causes problems with many system calls	⊞ kernel independent (restricted) ⊟ causes problems with many system calls ⊟ position of the system call table unknown
Hooking with system call service routine machine-code modification	⊞ can be used with all system calls ⊞ no modification of the system call table necessary ⊟ very complex	⊞ can be used with all system calls ⊞ no modification of the system call table necessary ⊟ very complex ⊟ position of the system call table unknown
Modification of the Kernel code	⊞ can be done with all kernel versions ⊞ causes no problems ⊟ measure system must get a new kernel	
Ptrace	⊞ can be done with all kernel versions ⊞ runs in user mode without kernel modification	

Because it’s very difficult to gain the address of the system call table in Kernel 2.6 the hooking methods are not realizable there.
A modified kernel causes the lowest effort to develop but the highest to install at the measure system.
Ptrace seems to be a very user friendly method to log all required information on every system without any super user rights needed.

3. The Concept

3.1 Kernel modification

The new functions for the logging service are declared in “kernel.h”, which is included by nearly every kernel source file.
The functions are implemented in “printk.c”.
In every relevant system call service routine the logging function is called with all necessary arguments.
This logging function has to check if logging is enabled. If this isn’t the case the system call resumes immediately.
Otherwise the matching entry in the system call counter table is increased. This table is just for additional information and contains the number of system calls executed during the logging process at all. It is written into the kernel log (/var/log/messages) when logging is finished.
After that the function checks if the actual process has to be logged. If the processes table doesn’t contain the process id the system call resumes.
Otherwise the logging is disabled to prevent the logging of system calls used by the logging function.
Thereafter the system call command is converted into a CSV entry.
Now the log file (/tmp/log.log) is opened, the new line is written into it and the file is closed.
The logging is enabled and the system call can be resumed.

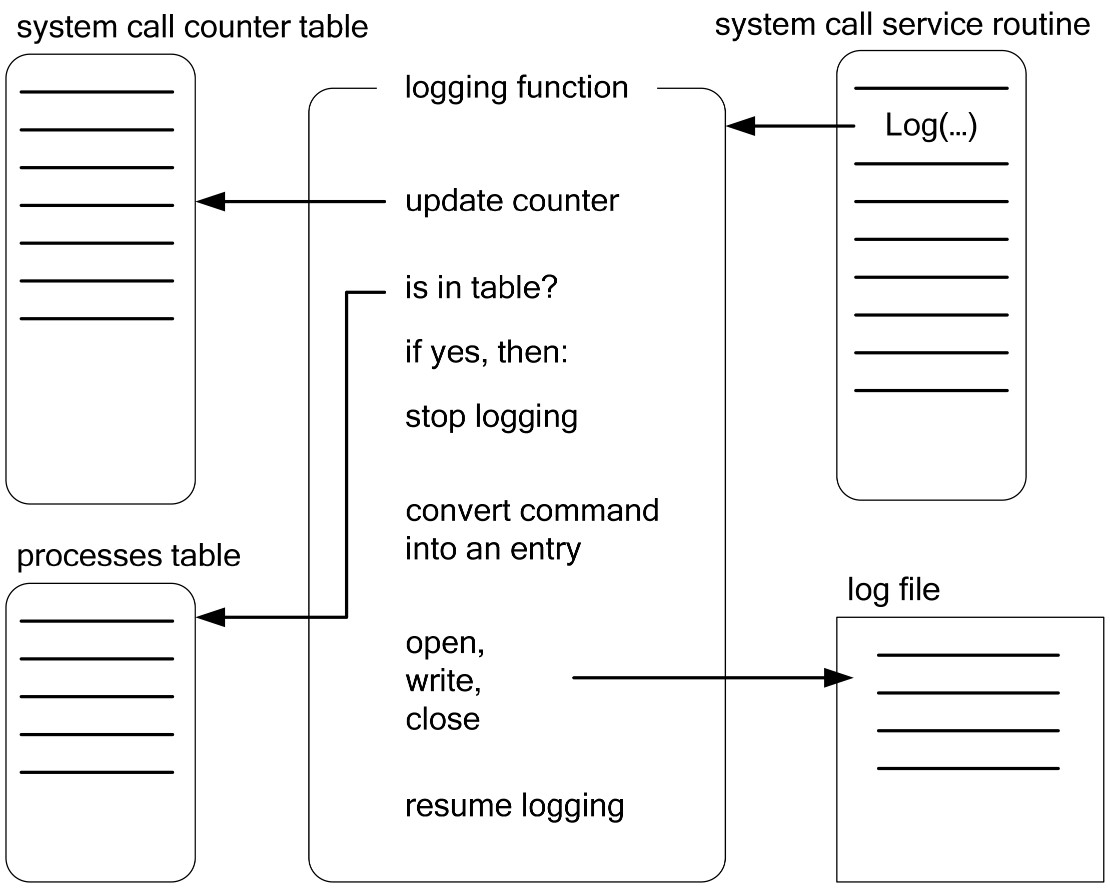


Figure 3.1 logging process

After including “kernel.h” the following logging services can be used:

- Management of the logged processes
- Start and stop of the logging
- Add a logging entry
- Management of the system call counter

To communicate with the services of the kernel (e.g. start the logging of a process) there has to be developed an interface.

This interface is a driver, a Linux kernel module (LKM). Loading the module causes the log file to be flushed, maybe hook some system calls and the kernel can start logging. Because the processes table is still empty no process is logged until an id is inserted into it.
The module can be used as a character based device. The process ids of the processes to be logged are written on the device.
When the module is unloaded, the hooked system calls must be unhooked by restoring the old entries in the system call table. The logging finishes now.

3.2 Ptrace

The system call Ptrace can be used to give a process a property that indicates that this process is traced. Mostly Ptrace is used this way [7]:

```
#include <sys/ptrace.h>
#include <sys/types.h>
#include <sys/wait.h>
#include <unistd.h>
#include <linux/user.h>

int main() {
      pid_t child;
      struct user_regs_struct regs;

      child = fork();
      if(child == 0) {
            ptrace(PTRACE_TRACEME, 0, NULL, NULL);
            execl("/bin/ls", "ls", NULL);
      } else {
            wait(NULL);
            ptrace(PTRACE_GETREGS, child, NULL, &regs);
            printf("The child made a system call %ld\n", regs.orig_eax);
            ptrace(PTRACE_CONT, child, NULL, NULL);
      }
      return 0;
}
```

Figure 3.2 a simple Ptrace example

When the traced process is attached by using Ptrace with “PTRACE_TRACEME” itself or by another process using Ptrace with “PTRACE_ATTACH”, there will be sent a signal to the tracing process every time the traced process enters or leaves a system call. So the tracing process stops waiting and can read and write all data of the traced one. When he finishes his work he calls Ptrace with “PTRACE_CONT” and the traced process continues.

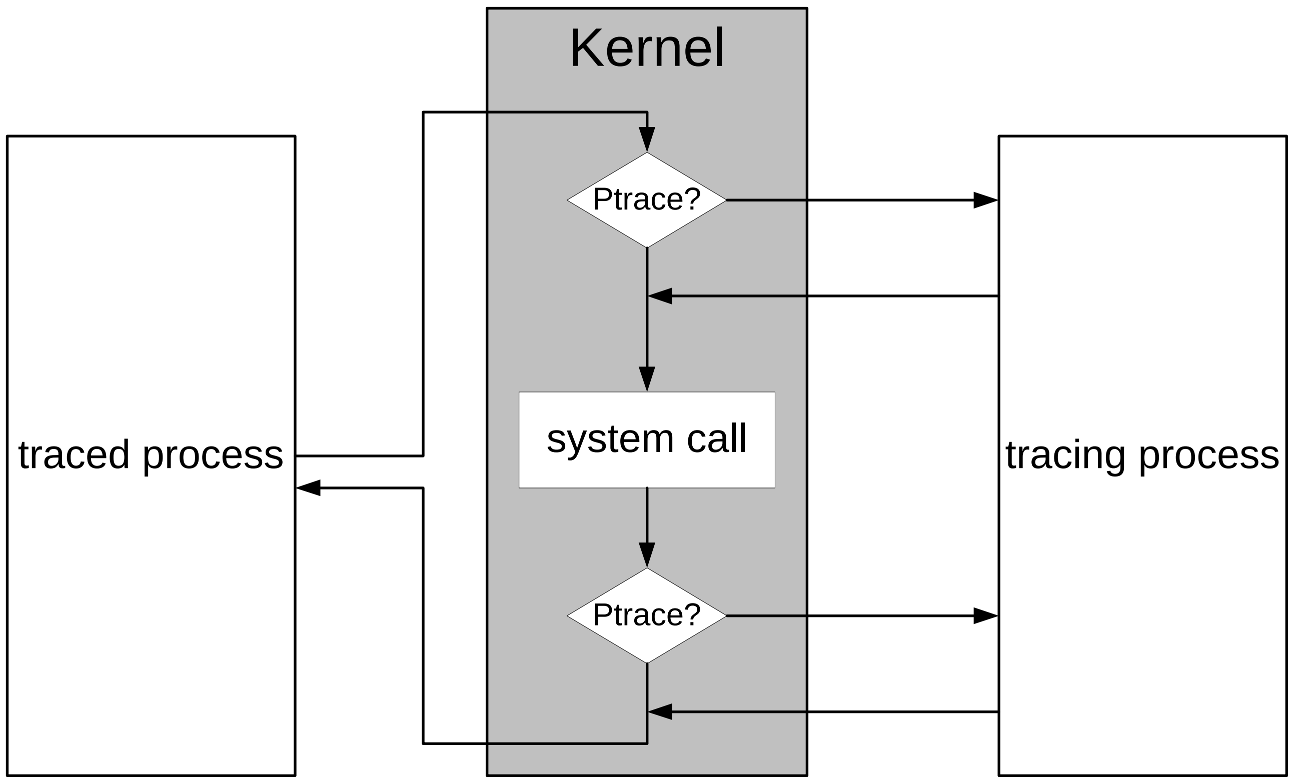


Figure 3.3 Ptrace

Ptrace just traces one process and doesn’t handle process families, so we have to do this when the traced process calls a fork.

3.3 The converter

Because the recorded data is in CSV format and the interpreter expects a C-like code, the data needs to be converted. The converter must read every line of the CVS file, assign the system call and handle the parameters and the return value. The return value must be stored in a variable and when the value is reused by another system call in a similar context as parameter, the value must replaced by the variable.

4. Details of Implementation

4.1 Kernel modification

In every relevant system call service routine the following function is called.

```
void sys_call_log2(const char *function, ...);
```

The function expects as first argument the name of system call followed by the return value and the parameters of it.
Additional in the fork routine the new process id is added to the processes table, so the complete process family can be logged.
To control the processes to log, a module must be loaded. It offers an interface to add process ids to the processes table.

The module can be loaded with the following shell command:

```
insmod log.o
```

Normally “insmod” can only be executed by the super user.
After the command the module should be loaded successfully.
If the device "/dev/klog" does not yet exist., it must be created with the following command:

```
mknod /dev/klog c 240 0
```

The command “mknod” creates the device. "/dev/klog" is the position of the new device, “c” means that it’s a character based one, “240” is the major and “0” the minor number of the module.
To make the device accessible to all users the rights must be set:

```
chmod 666 /dev/klog
```

Now we can use this device in our user mode applications to communicate with the kernel logging service.

Opening the device:

```
handle=open("/dev/klog", O_RDWR);
```

Writing on the device:

```
write(handle, "1234", 4);
```

Closing the device:

```
close(handle);
```

These commands would log all system calls of the process with the id “1234” until the device is closed.
The logged data are located in the file “/tmp/log.log” in CSV format.

To log an application the tool “runner” can be used. It expects the file to execute and log as argument and handles the communication with the module.

To unload the module the super user has to use the following shell command:

```
rmmod log
```

Now the module should be unload successfully and the kernel log “/var/log/messages” contains some additional information about the total number of all system calls.

4.2 Ptrace

The Ptrace based logging tool expects the destination of the log file and the application to execute and log. The tool just creates a new process, which calls Ptrace with “PTRACE_TRACEME” and executes the application. The parent process waits for every system call and writes the system call number, all register values and relevant time data into the log file.
To log the whole process family the tracing process must create a new one, when the traced process makes a fork. The new traced process must be attached to the created one by the tracing process. This happens with Ptrace and “PTRACE_ATTACH”. So the new tracing process becomes the parent of the process to trace. This may cause problems with some inter-process communication issues (e.g. semaphores).
Another problem is that the new process may start calling system calls before the parent leaves the fork system call and knows the new process id.
When system calls uses pointer to structures in there arguments, required data must be read with Ptrace and “PTRACE_PEEKDATA”.

```
[...]

if (regs.orig_eax == 162) {
      seconds=ptrace(PTRACE_PEEKDATA, pid, regs.ebx+28, NULL);

      [...]

}

[...]
```

Figure 4.1 handling sys_nanosleep

In the above example Ptrace is been used to read parts of the “timespec” structure to get the number of seconds the process wants to sleep.

To get the number of cycles the traced process computes in user mode, we need “utime”. It can be accessed via the virtual file system “/proc/PID/stat”. This value, the actual timestamp and the number of clock cycles are written besides the register values and the process id into the log file.

```
./runner <log file> <executable> [<arguments>]
```

The Ptrace logging tool expects two arguments. The first is the destination of the CSV log file and the second is the file to be executed with its parameters.

4.3 The converter

The interpreter of VisioOS needs a C-like code and not a CSV file. So it’s necessary to develop a converter which creates a C file for every single process and handles the values returned by a system call. These values have to be replaced by a variable, because the returned value by e.g. “open” can be reused by “read”.

The converter reads the CSV file line by line and every command is assigned to a process by its id. This process id is stored in a list if it is not already there.
The arguments were saved in the variables list. It’s important that there exists no variable with same function and same value.
Finally the system call is stored as C-code with the fitting variables in the system call list.

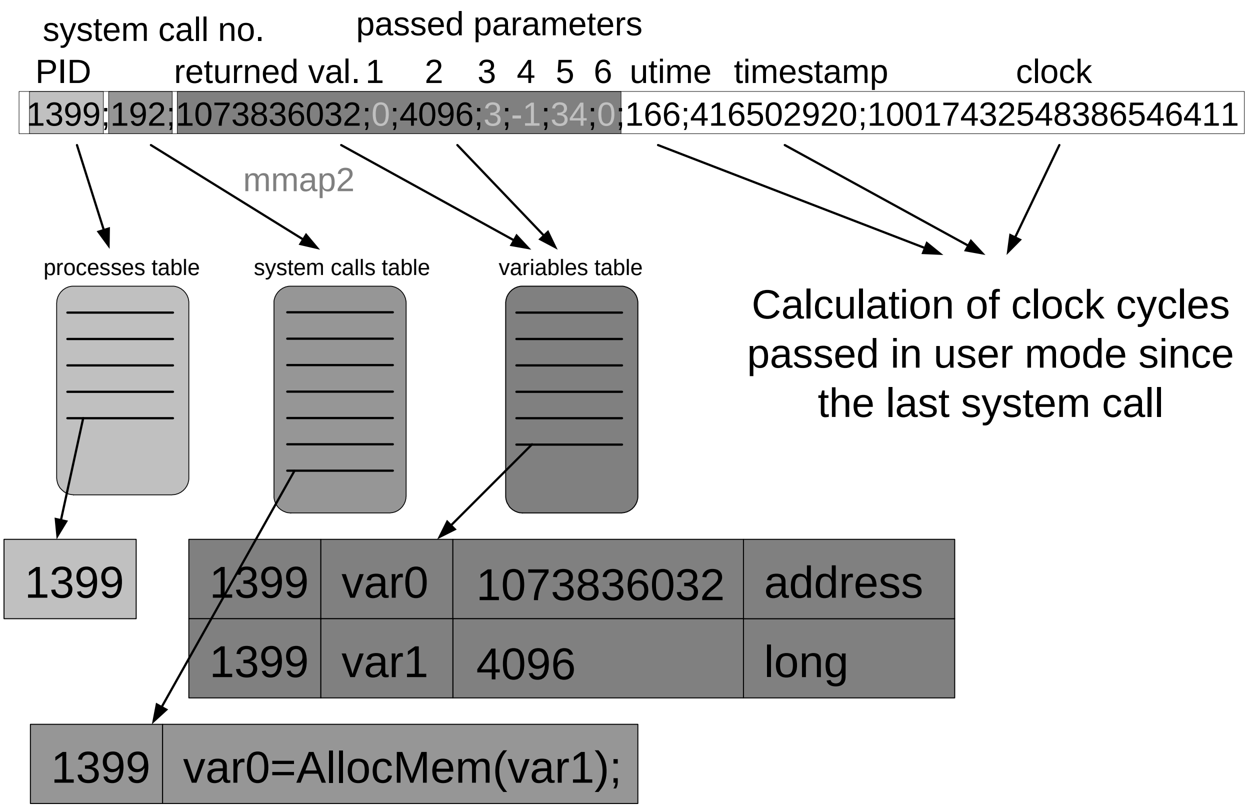


Figure 4.2 converter

The above figure shows how the converter handles a line of the CSV file. Firstly it checks if the process already exists in the processes table. Then it refers the system call number to the system call and takes all relevant parameters and the returned value. In this case 192 is the number of the system call “mmap2”. The analog command for the interpreter is “AllocMem”. It requires the size of the memory to be allocated and returns the address.
The timestamp and the clock are used to calculate the clock frequency. If “utime” has been increased since the last system call “WorkingForTicks” is added as command with the number of clock cycles as argument.

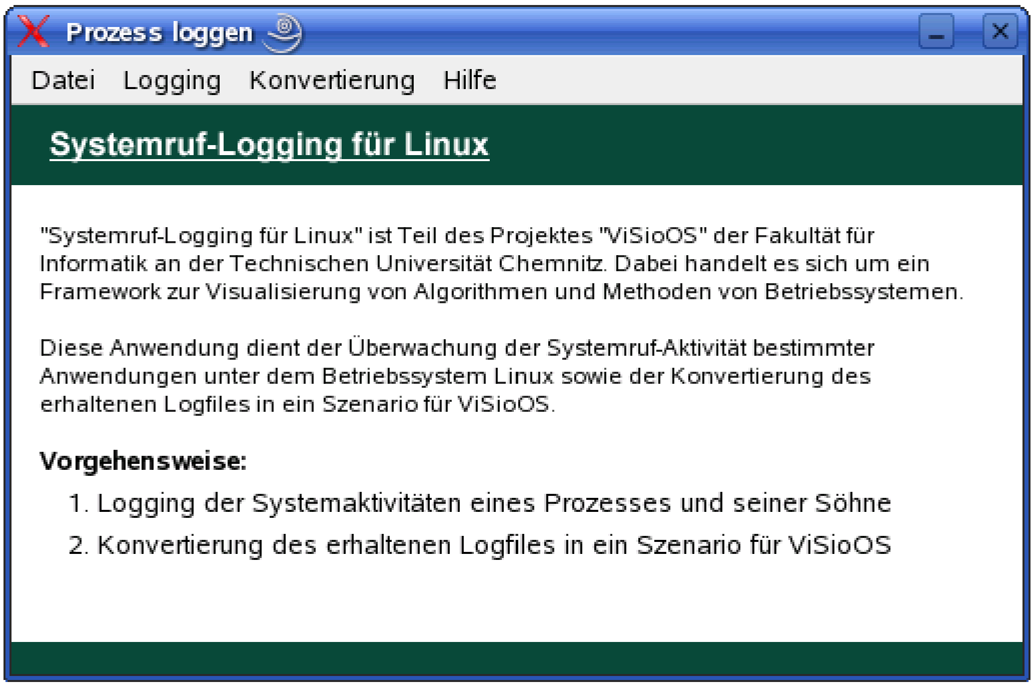
After the complete processing of the CSV file for every single process a C file is generated, which can be read by the interpreter of VisioOS.

```
./log2c <log file> <c file destination>
```

The converter expects two arguments. The first is the position of the CSV log file and the second is the path where the INS and C files should be stored.

4.4 The user interface

For an easier handling there is a graphical user interface. It is written in Java and so it requires the Java runtime environment.
It is divided into three pages:
On the start page there are some introducing words about the project and the proceeding of the application.



Systemruf-Logging für Linux

"Systemruf-Logging für Linux" ist Teil des Projektes "ViSioOS" der Fakultät für Informatik an der Technischen Universität Chemnitz. Dabei handelt es sich um ein Framework zur Visualisierung von Algorithmen und Methoden von Betriebssystemen.

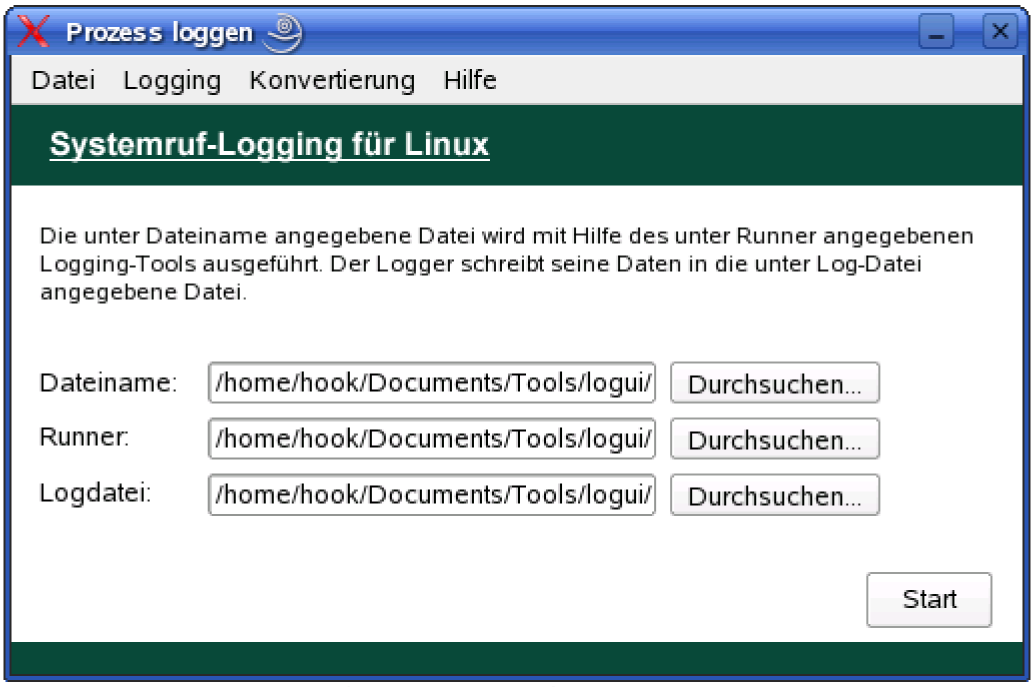
Diese Anwendung dient der Überwachung der Systemruf-Aktivität bestimmter Anwendungen unter dem Betriebssystem Linux sowie der Konvertierung des erhaltenen Logfiles in ein Szenario für ViSioOS.

Vorgehensweise:

1. Logging der Systemaktivitäten eines Prozesses und seiner Söhne
2. Konvertierung des erhaltenen Logfiles in ein Szenario für ViSioOS

Figure 4.3 screenshot: start page

The log page can be reached by the menu. Here the file to execute and to log, the tracing application and the log file destination can be chosen.



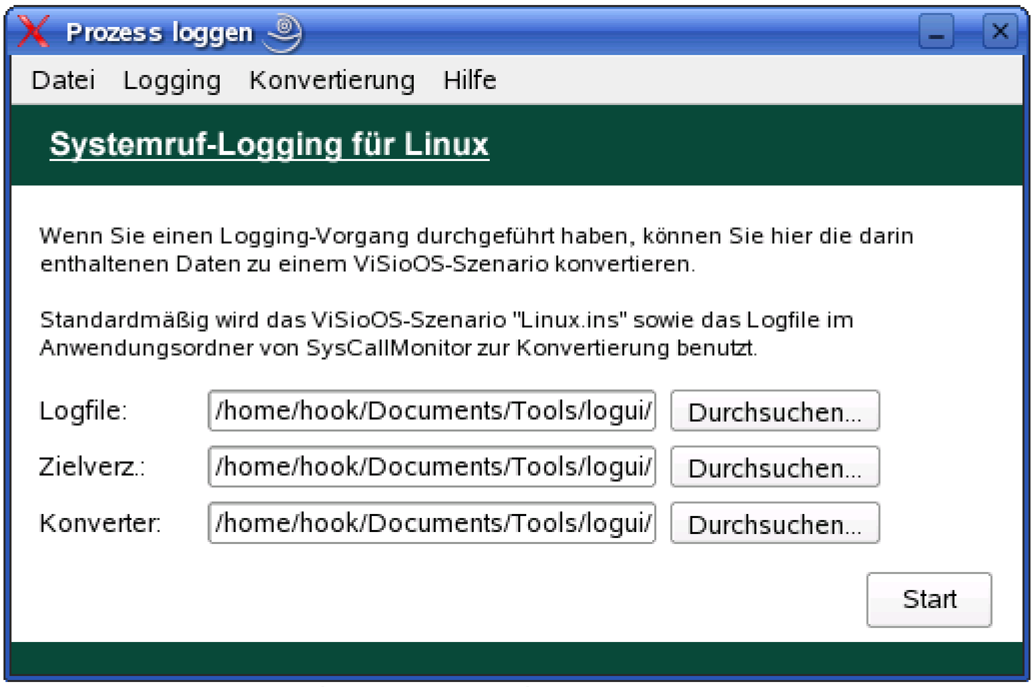
Systemruf-Logging für Linux

Die unter Dateiname angegebene Datei wird mit Hilfe des unter Runner angegebenen Logging-Tools ausgeführt. Der Logger schreibt seine Daten in die unter Log-Datei angegebene Datei.

Dateiname: /home/hook/Documents/Tools/logui/
Runner: /home/hook/Documents/Tools/logui/
Logdatei: /home/hook/Documents/Tools/logui/

Figure 4.4 screenshot: log page

The convert page can be reached by the menu. Here the log file position, the output file destination path and the converting application can be chosen.



Systemruf-Logging für Linux

Wenn Sie einen Logging-Vorgang durchgeführt haben, können Sie hier die darin enthaltenen Daten zu einem ViSioOS-Szenario konvertieren.

Standardmäßig wird das ViSioOS-Szenario "Linux.ins" sowie das Logfile im Anwendungsordner von SysCallMonitor zur Konvertierung benutzt.

Logfile: /home/hook/Documents/Tools/logui/

Zielverz.: /home/hook/Documents/Tools/logui/

Konverter: /home/hook/Documents/Tools/logui/

Figure 4.5 screenshot: convert page

5. Measurement

5.1 A small test program

To test the overhead of the logging extensions, a simple application is required which just calls some system calls.

```
#include "stdio.h"
#include <unistd.h>
#include <stdlib.h>
#include <fcntl.h>
#include <string.h>


int main(void)
{
      int i;
      char puffer[100];
      int *mem;
      FILE *handle;

      for (i=0;i<10;i++) {
            strcpy(puffer,"This text is in \"test.txt\"\n");
            handle = fopen("test.txt", "w");
            fputs(puffer, handle);
            fgets(puffer, 80, handle);
            fclose(handle);
            mem = malloc(1024*sizeof(int));
            free(mem);
      }
      return 0;
}
```

Figure 5.1 source of the test program

The above code just calls 100 times the system calls “open”, “write”, “read”, “close”, “malloc” and “free”. To measure the execution time an additional program executes the logging application with the test program as parameter.

5.2 Results

The execution time of the test program was measured 10 times without any logging tool, with kernel logging and with Ptrace logging.

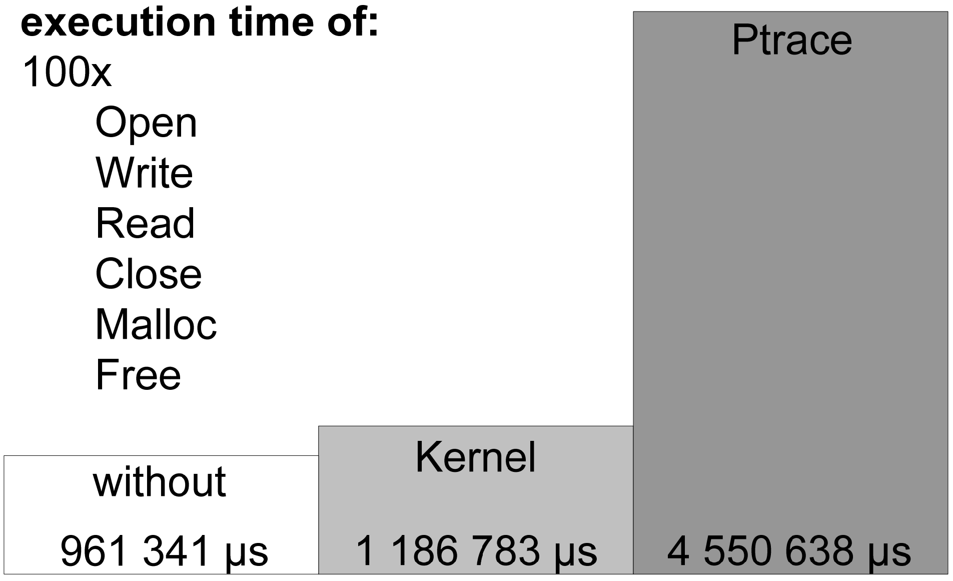


Figure 5.2 performance

Kernel logging just causes a small overhead but the execution time of the test program with Ptrace logging is nearly 5 times higher than without logging.
So Ptrace logging can cause performance problems with big and complex processes to log.

6. Summary and discussion

6.1 Performance

The user mode logging has a much worse performance than the kernel method. This may be caused by the system calls the tracing process has to do. So every system call by the traced process produces at least 8 additional calls.
The kernel version has to do no system calls and can use the system call service routines directly. So the overhead is much lower.
But for processes having not a long execution time the performance issue isn't much important.

6.2 Usability

When user wants to use the kernel logging method on his system, he has to path his kernel, recompile and install it. After that he has to load the module and create the device. For all these actions he has to have super user rights. This is not very user friendly and for experts only.
The Ptrace method is easy to use and needs no preparation and no super user rights.

6.3 Possible improvements

To improve the performance it's necessary to avoid too many system calls made by the tracing process. This may be done by writing the logged data not directly into the file, but in a shared memory buffer. Also it's not necessary to log all system calls and all registers.
Maybe also other logging methods can be used e.g. library hooking.

Appendix A. Bibliography

[1] Visual Simulation of Operating Systems
http://osg.informatik.tu-chemnitz.de/de/animationen-und-simulationen/visual-simulation-of-operating-systems.html

[2] Daniel P. Bovet, Marco Cesati
Understanding the Linux Kernel, 3rd Edition

[3] Eva-Katharina Kunst, Jürgen Quade
Linux Treiber entwickeln
http://ezs.kr.hsnr.de/TreiberBuch/

[4] Yet another article about stealth modules in Linux
http://packetstormsecurity.org/9908-exploits/linux_stealth_module.txt

[5] Das Linux Anwenderhandbuch - Strace
http://www.linux-ag.de/linux/LHB/node105.html

[6] Ptrace man page
http://unixhelp.ed.ac.uk/CGI/man-cgi?ptrace+2

[7] Linux Journal - Playing with ptrace, Part I
http://www.linuxjournal.com/article/6100

Appendix B. Figures