Operating Systems Practical Assignment - Profile Your Processes

Sander van Loo and Erik van Zijst
August 19th, 2004

Vrije Universiteit,
Faculty of Sciences, Department of Mathematics and Computer Science,
Amsterdam, The Netherlands

{sander,erik}@marketxs.com

Introduction

This document describes our work for the first practical assignment of the operating systems course. In this assignment we were to add profiling support to the minix 2.0.0 kernel that allows users to inspect running code. Profiling is often a valuable tool when optimizing programs for performance. To this end it measures how much time was spent in which part of the code, isolating program "hot spots", which are often good starting points in performance improvement.
Our work consists of three components: a user-level utility program that aids users in doing profiling experiments, modifications to the minix kernel, and a set of testing programs and scripts that were used to test our work for correctness. We will describe each in a separate chapter.

User Program

The utility program that accompanies our modified minix kernel is used to conveniently start any given program and transparently collect profiling statistics while it runs. When the program is finished, our "wrapper program" retrieves the profiling data from the operating system's kernel and presents it as a simple histogram.
Internally, the program is structured as follows. First it parses the arguments that were passed to it via the shell command. The program understands two optional switches, the name of the program to execute and that program's arguments. The switches are used to tell the profile utility which part of the target program to inspect (`-r startaddr-endaddr`) and to influence the program's profiling output (`-n number_of_rows`). After the switches that may be omitted, or given in any random order, the name of the target program is expected. Obviously, this may not be omitted. Any trailing arguments are always passed to the target program.
Next, the utility forks and concurrently executes the target program and instructs the kernel to start profiling the pid of the new child process. It is crucial that the profiling is not started until the target program actually runs. Without synchronization between the parent and the child, it is possible the parent calls the profile system call after the child was forked, but before the child used `execvp()` to replace the memory image. When that happens, the profilers memory boundary checks would check against the image of the parent process (the `profile` utility), rather than the image of the target program. To let the parent wait until the child successfully finished `execvp()`, we use the `ptrace ()` system call that allows the parent to trace the execution of the child. In the child we call `ptrace()` to enable inspection by the parent, while the parent enters `wait()` after forking to be notified when the child is ready to start. After this notification, the profiler is started and the child instructed to continue execution. Note that since the minix ptrace implementation, or at least its documentation seems somewhat limited and appears to lack the equivalent of `PTRACE_DETACH` on Linux, the parent continues to be notified of the child's signals, but ignores them and lets the child continue until the child actually terminates.
When the target finally finishes, the profiling data is first fetched from the kernel. Then the profiling is turned off. It is essential to first retrieve the statistics before disabling profiling. Otherwise, there is a chance another user already started a new profiling session before we could collect our own statistics. As long as the profiling is still enabled, our statistics are safe. The statistics are returned as an array of counters or bins, each counter representing the time spent in its associated instruction address range. Depending on the maximum number of bars that may be displayed (specified by the `-n` argument), the counters are distributed over the bars and printed with a length

relatively proportional to the largest bar.
Note that since first the instruction address range is divided over a certain number of bins by the kernel, after which these counters are divided over a possibly different number of bars, there is a considerable chance the resulting bars will not represent an exactly equally sized code segment. Often the last bar only represents the time spent in a smaller segment of the address range. This must be taken into account when interpreting profiling histograms. Of course, the exact address ranges covered by each bar are explicitly included in the histogram.

Kernel Modifications

The only part of the system that can collect profiling statistics is the operating system's kernel. Hence, the user-level utility program passes instructions for profiling on to the kernel where we do the actual work. The assignment states that two system calls must be added for this purpose. One to switch profiling on and off and one to retrieve the collected statistics. The system calls must be added in the memory manager.

The changes are reflected in the following files:

- `usr/include/minix/callnr.h`: defines for the new system call numbers
- `usr/src/mm/table.c`: maps the new system call numbers to their implementation
- `usr/src/fs/table.c`: adds null-mappings to the file system for our system calls that are already implemented in the memory manager
- `usr/src/mm/proto.h`: prototypes of our new system calls
- `usr/src/mm/param.h`: here we added aliases for the messages passed between user programs and the memory manager
- `usr/src/mm/profile.c`: implementation of the new system calls
- `usr/include/minix/com.h`: here we defined field names for the messages sent between the memory manager and the kernel.
- `usr/include/profstub.h`: header file for usr/src/mm/profile.c that contains the profiling data structures
- `usr/src/kernel/clock.c`: implementation of our profiler
- `usr/src/extra/`: contains our tests and the user space profile utility

Note that the current versions of these files were not our first attempt. Several different approaches were tried that all led to different fatal problems.

During our first, somewhat naive attempt, we tried to add all necessary profiling code to the memory manager. This failed for various reasons. First of all the memory manager has no program loop that is executed at regular intervals, but rather on demand, and secondly, since the memory manager is a separate process, it doesn't contain the real process table that is needed to retrieve the target program's instruction pointer (or program counter) at regular intervals.

In our second attempt we descended further into the kernel and chose the system task as the place to add our profiling routines. From within `kernel/system.c` we could reach the real process table, but unfortunately the main loop in the system task is again not run at regular intervals either, but on demand similar to the memory manager. Adding the code to the system's main loop would lead to a profiler that samples the target program every time one of its system calls is invoked. Not only is this unpredictable and depended on the system's load, it can also be influenced by the the target program's execution. Worse, when no system call is made, the program won't be sampled unless some other, concurrently running user program makes one. This also means that when the target program makes lots of system calls, the program will always be sampled in the middle of its system call. Leading to the perception that the program does nothing but making system calls. For profiling to make sense, data must be collected either at entirely periodic intervals, or at entirely random intervals. As soon as the sampling rate becomes related to the activity of programs or external events, the result is no longer guaranteed to show the relative expense of a program's code.
To avoid this problem, we chose to use alarms. We added our profiling code to `kernel/system.c`. When the procedure to enable profiling was called, we would set an alarm to make sure our sampling routine would be called after a fixed interval. Since alarms are not persistent, the alarm would need to be reset every time the sampling routine was invoked. In order to do this, we sent a message to the clock at the end of the sampling routine that would set a new alarm. Unfortunately, when we tried this approach, the whole system froze as a result of a deadlock in the kernel. When the alarm went off the first time and we tried to schedule a new alarm at the end of our sampling routine, it is the clock that is at that point still waiting for the alarm handler to return. Only after it returns it will be able to receive another message. When we tried to re-schedule our alarm, we sent a message to the

clock and then waited for it to be accepted. Unfortunately this leads to a deadlock where the clock is effectively sending a message to itself, while it is not ready to receive one because it is sending a message.

During our third and final attempt, we moved the profiling code to the clock. Inside the clock (`kernel/clock.c`) is a routine `clock_handler()` that is run at every clock tick. In this routine we check whether profiling is currently enabled and if it is, the sampling routing is invoked. This saves us from re-scheduling alarms and gives us a periodic sampling rate at fixed intervals which is not related to the activity of the system. The profiling logic in this file is implemented in routines `do_setprofile()`, `do_getprofile()`, `prof_configure()`, `prof_count()` and `prof_findproc()`. One thing that may be inconvenient is that the kernel's clock works on a fixed frequency of 60Hz (as defined in `include/minix/const.h`). As a result, our profiler can only sample a program 60 times per second. For programs that do not run very long, this interval may be (too) short. Some programs even execute entirely within one tick. This makes it impossible to profile them. A typical real-world example is the `ls` command that often finishes so fast it is completely missed by our profiler. At the same time though, it is important to realize that profiling is typically used as an aid for performance enhancements that make programs run faster. Programs that execute entirely within a single clock tick will usually not be very high on the list for performance improvements. Also, in some cases it may be possible to modify a program in a way it executes many redundant times in succession to make it run long enough for reliable profiling.

Experiments

With our kernel modifications in place and the user-level profile utility accompanying it, we needed tests to verify the correctness of the new functionality. All test applications are located in the `src/extra` directory. We ran 4 different kind of experiments, represented by 4 test programs:

1. `t_empty.c`: here we test to see if the profiling system can handle programs that run so fast, they cannot be profiled.
2. `t_hotspots.c`: this program contains clearly identifiable hotspots for verifying the histogram.
3. `t_syscall.c`: this program focusses on system calls to see how they are profiled.
4. `t_blocking.c`: our last test program does blocking system calls.

By default, we compiled every program with separate text and data segments. However, we compiled the second program twice, once with separate text and data segments and once with combined segments to verify whether the profiler understands the difference and still profiles both correctly.
Following is a detailed description of each experiment.

Empty Sets
We first wrote an application that does nothing but return immediately after startup. This gives us the ability to see if our profile utility handles empty profile data correctly. This test program is `src/extra/t_empty.c`. We expect it to yield an entirely empty histogram with no peaks, reading 0% for every address range.

Locating Hotspots
Our next test involves the program `src/extra/t_hotspots.c` which has two simple hotspots in two routines: `bogusa()` and `bogusb()`. From its main routine it calls the routines that each contain a loop with a very high number of iterations. Both routines contain a large, time-wasting loop that increments a counter several million times. In the histogram we expect to see two large peaks in two different address ranges. When comparing these ranges with the addresses of the routines obtained by the `nm` command, we expect the addresses of the peaks to fall inside the ranges of the hotspot routines. As another verification, we made the first loop `bogusa()` to make exactly twice as much iterations as the second loop in `bogusb()`. As a result, we expect the peak representing the loop in `bogusb()` to be exactly twice as large the the peak representing `bogusa()`.

System Calls
In our third test we focus on profiling system calls. Our `t_syscall.c` program repeatedly invokes the system call `time()` to retrieve the current time. Since this is not a blocking call (and hence must be tracked by our profiler) it should leave a significant peak in the profiling report. Since the profiler registers the instruction pointed to by the program counter of the profiled user process and this pointer will never point outside its own text segment, it is expected that our profiler continuously registers the code that is used to send a message to the kernel. While the kernel is processing the `time()` system call, it is expected that the user process is sitting in the system call's stub code and waiting for the `sendrec()` to receive the result from the kernel.

Blocking Calls
To verify that the kernel does indeed ignore blocking operations of profiled applications, we wrote a third test that spends a couple of seconds inside a useless, time-wasting loop and then spends the same time in the system's `sleep()` routine. When profiling the application, we expect to see only a single hot spot. That of the time-wasting loop. If the kernel did indeed ignore the time spent in the `sleep()` call, it should not show up as a bar in the profiling histogram. However, if the report shows two almost equally sized peaks, we can conclude the kernel does not handle the blocking state correctly. Note that since we measure the time that was wasted with an accuracy of one second, the actual time spend in the sleep method could differ from the wasted time by a whole second. However, if enough time was wasted, this shouldn't be an issue for our test.

Test Evaluation

In this section we shall discuss the results of our proposed tests to see if they match our expectations.

When the first test was run, with the kernel set to profiling all of the applications address range, the following histogram appeared:

```
## ./profile ./t_empty
./t_empty
                  0%.........................................................0%
0000-0019 (00%) :
001a-0033 (00%) :
0034-004d (00%) :
004e-0067 (00%) :
0068-0081 (00%) :
0082-009b (00%) :
009c-00b5 (00%) :
00b6-00cf (00%) :
00d0-00e9 (00%) :
00ea-0103 (00%) :
0104-011d (00%) :
011e-0137 (00%) :
0138-0151 (00%) :
0152-016b (00%) :
016c-0185 (00%) :
0186-019f (00%) :
01a0-01b9 (00%) :
01ba-01d3 (00%) :
01d4-01ed (00%) :
01ee-01ff (00%) :
#
```

Since the test program terminated so quickly, even before the kernel took the first snapshot, the histogram only shows empty bars. Not a single bin was incremented.

In the second test program we actually traced hotspots in the `t_hotspots.c` program. The following histogram appeared:

```
# ./profile -r 0060-00b6 ./t_hotspots
./t_hotspots
                  0%.........................................................36%
0060-0064 (00%) :
0065-0069 (00%) :
006a-006e (00%) :
006f-0073 (00%) :
0074-0078 (18%) : *******************************
0079-007d (00%) :
007e-0082 (16%) : ****************************
0083-0087 (00%) :
0088-008c (00%) :
008d-0091 (00%) :
0092-0096 (00%) :
0097-009b (36%) : ***************************************************************
009c-00a0 (00%) :
00a1-00a5 (30%) : ****************************************************
00a6-00aa (00%) :
00ab-00af (00%) :
00b0-00b4 (00%) :
00b5-00b6 (00%) :
#
```

Note that we have zoomed in on only a limited address range. This is to isolate the part that contains the bogus loops. If the entire address space is profiled, the bars would represent ranges that are so big that both bogus functions are represented by a single bar. We obtained the range `0060-00b6` using the `nm` utility and looking for the `bogusa()` and `bogusb()` functions. A suitable range can also be obtained by first profiling everything and then zooming in on the bars that contain the peaks[1]. In the histogram we actually see 4 peaks, rather than two. When comparing their address with the `nm` output, we can see that each bogus loop really contains 2 hotspots. Although we have not analyzed this further, we expect this to correspond with the individual operations of the loops. Inside every iteration a condition is evaluated and a variable incremented. It is interesting to see that the bogusa loop used 18 + 16 = 34% of the total program execution, while the bogusb loop used 36 + 30 = 66%. This ratio is exactly according to our expectations since the second loop makes twice as many iterations.

The next test involves system calls. Our `t_syscall.c` program contains a loop that invokes the non-blocking system call `time()` millions of times. When the program is run, it yields the following report:

```
# ./profile ./t_syscall
./t_syscall
                  0%.........................................................97%
0000-0019 (00%) :
001a-0033 (00%) :
0034-004d (00%) :
004e-0067 (00%) :
0068-0081 (00%) :
0082-009b (00%) :
009c-00b5 (00%) :
00b6-00cf (00%) :
00d0-00e9 (00%) :
00ea-0103 (01%) :
0104-011d (00%) :
011e-0137 (01%) :
0138-0151 (00%) :
0152-016b (00%) :
016c-0185 (00%) :
0186-019f (97%) : ***************************************************************
01a0-01b9 (00%) :
01ba-01d3 (00%) :
01d4-01ed (00%) :
```

1 Instead of zooming in iteratively, or calculating a convenient address range, one can also instruct the profile utility to display a separate bar for each individual address. However, in most cases this yields impractically big histograms.

```
01ee-01ff (00%) :
#
```

There is one obvious peak in the address range `0186-019f`. A quick glance over `nm`'s output tells us that that is indeed the code that sends the message to the kernel and waits for the result to return.

In our last test we run the application `t_blocking.c` of which the total execution time is evenly divided between real work (a useless loop) and a blocking system call. Since the `save_time()` function executes the `sleep()` system call, the possible hotspots are not necessarily with the address range of the `waste_time()` and `save_time()` routines. If the kernel incorrectly counts our blocking call as time spent, we would expect a large peak in the system call's stub (the code that sends the message for sleep to the kernel, similar to our findings in the previous test). As such, we chose to first do a profiling run over the entire address space and then zoom in on the range that contains the peak(s). The initial run gives us the following histogram:

```
## ./profile ./t_blocking
Wasting time...
Wasted time for 10 seconds, now sleeping 10 seconds...
./t_blocking
                  0%........................................................100%
0000-0137 (100%): **************************************************************
0138-026f (00%) :
0270-03a7 (00%) :
03a8-04df (00%) :
04e0-0617 (00%) :
0618-074f (00%) :
0750-0887 (00%) :
0888-09bf (00%) :
09c0-0af7 (00%) :
0af8-0c2f (00%) :
0c30-0d67 (00%) :
0d68-0e9f (00%) :
0ea0-0fd7 (00%) :
0fd8-110f (00%) :
1110-1247 (00%) :
1248-137f (00%) :
1380-14b7 (00%) :
14b8-15ef (00%) :
15f0-1727 (00%) :
1728-17ff (00%) :
#
```

It is clear that all registered time lies within the range `0000-0137`, so we zoom in on that range which gives us:

```
# ./profile -r 0000-0137 ./t_blocking
Wasting time...
Wasted time for 11 seconds, now sleeping 11 seconds...
./t_blocking
                  0%........................................................69%
0000-000f (00%) :
0010-001f (00%) :
0020-002f (00%) :
0030-003f (00%) :
0040-004f (00%) :
0050-005f (00%) :
0060-006f (00%) :
0070-007f (00%) :
0080-008f (00%) :
0090-009f (00%) :
00a0-00af (69%) : **************************************************************
00b0-00bf (31%) : ****************************
00c0-00cf (00%) :
00d0-00df (00%) :
00e0-00ef (00%) :
00f0-00ff (00%) :
0100-010f (00%) :
0110-011f (00%) :
0120-012f (00%) :
0130-0137 (00%) :
#
```

After another zoom on the addresses `0050-00d0` (which contains our functions, as well as the program's main routine), we get:

```
# ./profile -r 0050-00d0 ./t_blocking
Wasting time...
Wasted time for 10 seconds, now sleeping 10 seconds...
./t_blocking
                  0%........................................................71%
0050-0056 (00%) :
0057-005d (00%) :
005e-0064 (00%) :
0065-006b (00%) :
006c-0072 (00%) :
0073-0079 (00%) :
007a-0080 (00%) :
0081-0087 (00%) :
0088-008e (00%) :
008f-0095 (00%) :
0096-009c (00%) :
009d-00a3 (00%) :
00a4-00aa (71%) : **************************************************************
00ab-00b1 (00%) :
00b2-00b8 (29%) : *************************
00b9-00bf (00%) :
00c0-00c6 (00%) :
00c7-00cd (00%) :
00ce-00d0 (00%) :
#
```

We now find two peaks, however we can easily tell by `nm`'s output that both peaks are caused by our `waste_time()` routine. There is no measurable time spent in `save_time()`, which tells us that the kernel did indeed ignore all time spent in the blocking system call.

So far, all profiled programs have been compiled with separate text and data segments, however our `Makefile` compiled the hotspots test twice. Once with separate segments (`t_hotspots`) and once with combined segments (`t_hotspots-c`). When we profile the combined binary, we see that its profiled address range has indeed increased from `01ff` (separate) to `02ff` (combined). Nevertheless, when we zoom in on both histograms, we find all peaks to be at the same addresses, which corresponds with the output of `nm` that shows that both binaries have the bogus loops at exactly the same location:

```
# nm t_hotspots | sort | grep bogus[ab]
0000006a T _bogusa
0000006a t bogusa:F
0000008c T _bogusb
0000008c t bogusb:F
# nm t_hotspots-c | sort | grep bogus[ab]
0000006a T _bogusa
0000006a t bogusa:F
0000008c T _bogusb
0000008c t bogusb:F
#
```

Aside from our test programs, we also added two simple utilities (`src/extra/t_profile.c` and `src/extra/t_getprof.c`) that can be used to manually invoke our system calls without the need for the `profile` utility. Especially the `t_profile.c` program can be useful to make the kernel stop profiling when profiling was started but the `profile` application was killed before it switched kernel profiling off.

Conclusion

In this text we discussed our profiling additions to the minix 2.0.0 operating system. We explained how we came to our decision to add it to the kernel's clock, with the system calls in the memory manager. We also presented a number of experiments to test various profiling scenario's. Our profiler proved capable of correctly dealing with blocking calls and differently compiled binaries. In all cases the profiler's output was verified using tools such as `nm` and equaled our expectations.