Understand Linux OS via Dynamic Tracing Tool (Systemtap)

Abdullah Al Raqibul Islam
Department of Computer Science, College of Computing and Informatics
University of North Carolina
Charlotte, USA
aislam6@uncc.edu

***Abstract*—Operating system (OS) is a piece of software that helps users to operate the hardware. To understand the operating system, it is very important to know it's internal logic and performance characteristics. The purpose of this project is to understand and observe the internal behaviors and performance characteristics of Linux operating system. In exploring the relations among some of these quantities, I have used a dynamic tracing tool namely Systemtap.**

Keywords—operating system, dynamic tracing, systemtap

I. Introduction

As I mentioned earlier, operating system is a system software that manages computer hardware, software resources, and provides common services for computer program [1]. It has several components to make sure the various parts of a computer system works together. All userlevel software has to undergo the operating system to get the utilization of the hardware as well as the other system resources. Among the core components, in this course. we have been tought about the kernel, process management, task scheduler, memory management, filesystem management, etc.

Kernel is the heart of the operating system. It is responsible for the core operating system tasks i.e. protection of resources, isolation of applications and users etc. Kernel has the full access to all the machine hardware. Kernel is considered as the only trusted component and all other softwares are obliged to perform previliged instructions through it. Among the other components, scheduler is responsible in scheduling the proesses to the processor. Memory management is responsible to make sure any program does not conflicts with the memory that is own by some other program. File sytem is another abstruction of operating system to provide persistent, named data.

To find software bugs due to incorrect system setup, system administrator perform instrumentation in finding those bugs, which includes, performance statistics collection and their analysis, debug or system audit. One of the common approaches to instrumentation is "sampling" by installing probes at specified places of software to collect state of the system: values of some variables, stacks of threads, etc. Sampling is very helpful when you do not know where issue happens.

For example, some function, say foo() that processes lists of elements, consumes 80% of the time, but doesn't say why. It could be because the list is too long or list is an inappropriate data structure for foo(). With tracing we can install a probe to that function, gather information on lists (i.e. their length) and collect cumulative execution of function foo(), and then cross-reference them, searching for a pattern in lists whose processing costs too much CPU time.

To gain deep understanding about the various parts of the Linux Operating System, I have conducted the following experiments using SystemTap:

1. Understand OS kernel: Addresses and contents of user frame and kernel frame of a user process.
2. Understand system-calls:
   - Within specific time duration, observe all the system-calls issued by a process, and, in the whole system.
   - Within specific time duration, trace time spent in all system calls in a per process way
3. Understand Processes
   - Trace parent-child tree of a process
   - Detailed current process information (including executable files, command line arguments, environment variables, uid, gid, cpuid, etc.)
   - Trace the time duration a process takes before the process actually starts to execute.
4. Understand Virtual File System
   - Trace major and minor page faults
5. Understand Virtual Memory System
   - Trace file open/close/read/write/mmap operations done in the system.
   - Trace the above mentioned operations in a per-process way.

II. Systemtap

I have spent quality time for the background study which includes learning to write systemtap script to trace, study, and

monitor the activities of the Linux operating system [5], [6], [7], [8].

SystemTap is both scripting language and tool which can be used for profiling program on Linux kernel-based operating systems in runtime. SystemTap scripts are generally focuses on tracking events. It is designed to balance between several key requirements including easy of use, performance, tranparency, flexibility and sefty to use in production system etc. The structures and components of systemtap is presented in Figure 1.

One of the systemtap's wellknown keyword is prob point, which indicate the a particular point in kernel/userspace code, or a specific event. The systemtap input consists of a script which makes the assiciation of hendeler routines to a specific prob point. When a prob point hits, the associated handler routine will be executed.

When we place a systemtap script, it will be passed through a translator where it will be parsed first, and then been elaborated. Elaboration is a script processing phase where the necessary symbolic references to the kernel or user program is been resolved. After that, the script goes through the translation phase and been translated to a quantity of C code. From the figure, we can see the output file of the translator as prob.c

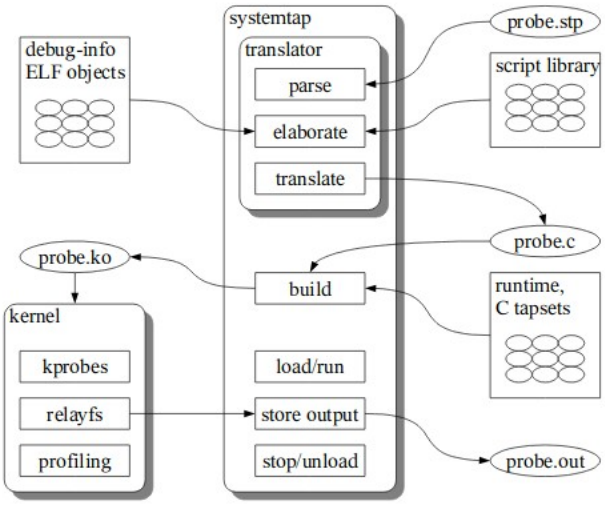

Figure 1: Systemtap processing steps [2]

To run the probs, the systemtap driver program loads the kernel module by using insmod. Insmod is a simple program to insert a module into the Linux Kernel. The module will initialize and register itself and let the probs so their tasks.

III. Understand Systemcall

System call is a way by which computer program can make request for service to the kernel of a operating system. Userlevel software makes a system call when it makes a request to to operating system kernel. Understanding system call is very important to understand the operating system properly.

I have prepared several scripts to understand the behavior of system calls. Here I am giving the brief descriptions of the scripts

1. syscall_count.stp: This script is intended to collect system-call data for the whole system. It will show the top 20 <process, syscall> pair in every 10 seconds. The sample output is shown in Figure 2.

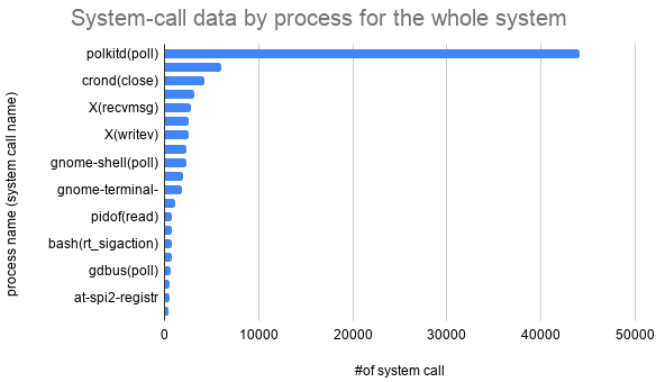

Figure 2: Systemtap script output for tracing system-call called by processes in a 10 seconds interval (within the whole system)

2. syscall_count_by_pid/proc/name.stp: This group of scripts are intended to collect system-call data by pid, process and syscall name respectively for the whole system. It will show the top 20 syscalls by the corresponding process in every 10 seconds. A sample result showing system call called in a 10 seconds interval has added in Figure 3.

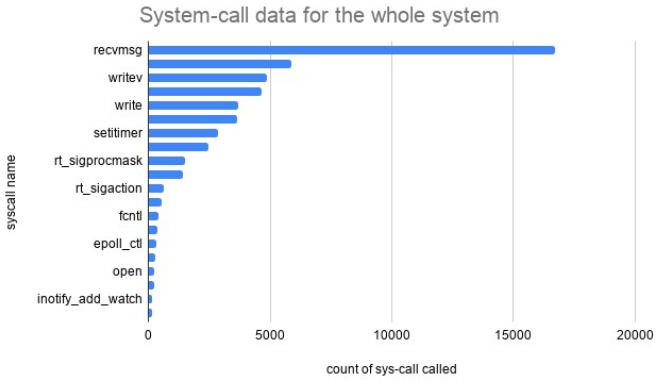

Figure 3: Systemtap script output for tracing system-call called in a 10 seconds interval (within the whole system)

3. syscall_timetrace_by_proc.stp: This script is intended to trace time spent in all system calls in a per-process way. It will show the top 20 results in every 10 seconds. Sample output shown in Figure 4.

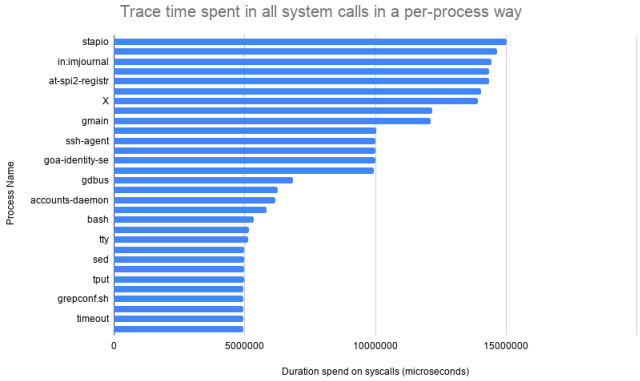

Figure 4: Systemtap script output for tracing time spent by processes for all system calls within a 10 seconds interval

IV. Understand Process And Scheduler

According to Andrew Tanenbaum's book "Modern Operating Systems", “All the runnable software on the computer, sometimes including the operating system, is organized into a number of sequential processes, or just processes for short. A process is just an instance of an executing program, including the current values of the program counter, registers, and variables.". A process life time in terms of scheduler has shown in Figure 5.

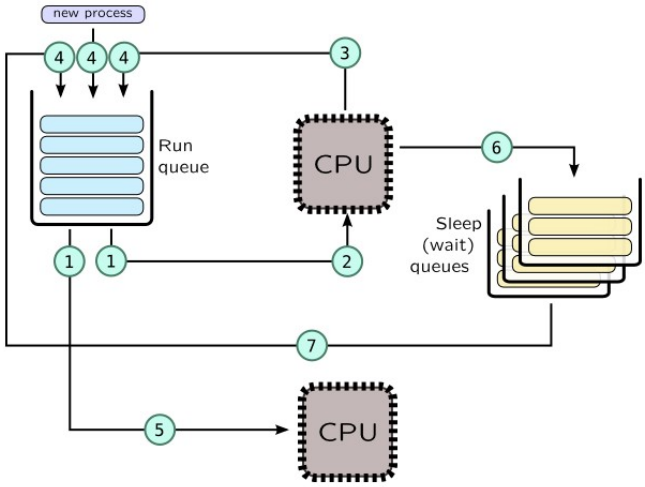

Figure 5: Process lifetime in terms of the scheduler [3]

Each process has it’s own address space. Processess are scheduled in the processor by context switch. In linux, process and threads are implemented by a structure named task_struct. It is listed and described in a header file named sched.h in linux source code directory, path: linux/include/linux/sched.h

Lifetime of a process and corresponding probes are shown in Figure 6. Unix process is started by a two stage way,

- Parent process calls fork() system call
  a) Kernel creates exact copy of a parent process including address space
  b) If fork() is successful, it will return in the context of two processes (parent and child).
- Child process calls execve() system call
  a) to replace address space of a process with a new one based on binary which is passed to execve() call

In completion of a process,

- When child process finishes its job, it will call exit() system call.
- If parent wants to wait until child process finishes, it will call wait() system call, which will stop parent from executing until child exits.

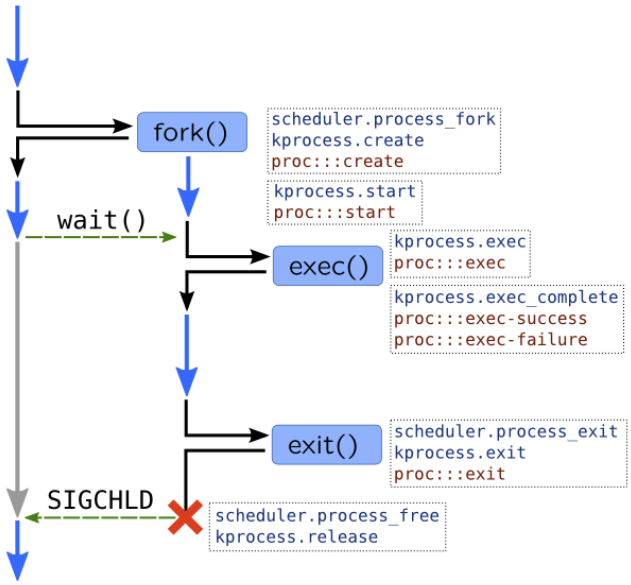

Figure 6: Lifetime of a process and corresponding probes [3]

Processes may be traced with kprocess and scheduler tapsets in SystemTap. Here are some useful probes:

Action	SystemTap Probes
Process creation	- kprocess.create - scheduler.process_fork
Forked process begins its execution	- kprocess.start: called in a new process context - Scheduler.wakeup_new: process has been dispatched onto CPU first time
execve()	- Kprocess.exec: entering execve() - Kprocess.exec_complete: execve() has been completed, success variable has true-value if completed successfully,

	errno variable has error number in case of error
Process finished	- kprocess.exit - scheduler.process_exit
Process structures deallocated due to wait()/SIGCHLD	- kprocess.release - scheduler.process_free

Table 1: Systemtap probs for tracing processes

As part of this porject, I have prepared a script to gather in-depth knowledge about process. It includes log for tracking parent-child tree of a process. Detailed current process information (including executable files, command line arguments, environment variables, uid, gid, cpuid, etc.) also been included. Besides these I have also traced the time duration a process takes before the process actually starts to execute. The output of tracing time consumption by process for being dispatched onto CPU for the first time has shown in the Figure 7.

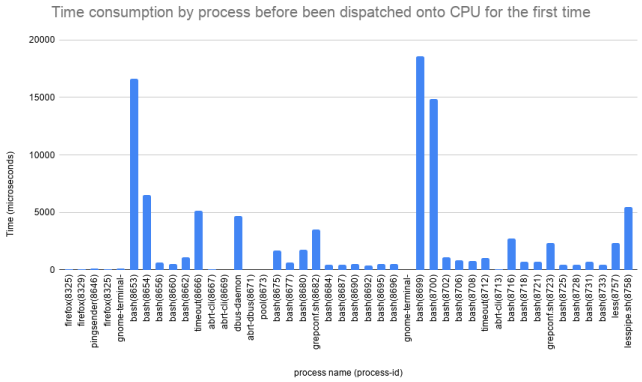

Figure 7: Systemtap script output for tracing time consumption by process for being dispatched onto CPU for the first time

V. Understand Virtual File System

One of the defining principal of Linux is “Everything is a file”. Files are organized by a operating system component called File System. In the class, we have learned several file system algorithms, their features, usages, etc. Different filesystems have different properties and lies in different complexity range. Despite the design complexity, they all use the same list of APIs.

When a process opens a file, it calls open() system call, which returns a file descriptor. Following that, the process may call several other system call i.e. read() to read from a file descriptor or lseek() to reposition read/write file offset etc.

```
intervals for read -- min:0us avg:3us max:86us count:888
value |-------------------------------------------------- count
    0 |                                                      7
    1 |@@@@@@@@@@                                           97
    2 |@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@ 456
    4 |@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@ 258
    8 |@@@@@@                                               66
   16 |                                                      2
   32 |                                                      1
   64 |                                                      1
  128 |                                                      0
  256 |                                                      0
```

Table 2: Histogram showing intervals for read()

I found it very interesting to trace system wide usage of different file system related systemcalls like, open(), close(), read(), write(), mmap(), etc. I have prepared systemtap script for this. The output in Table 2 and Table 3 shows the histogram of read() and close() respectively. I have used Systemtap’s @hist_log(v) to represent a base-2 logarithmic histogram.

```
intervals for close -- min:0us avg:1us max:3us count:10
value |-------------------------------------------------- count
    0 |@                                                    1
    1 |@@@@@@@@                                             7
    2 |@@                                                   2
    4 |                                                     0
    8 |                                                     0
```

Table 3: Histogram showing intervals for close()

VI. Understand Virtual Memory System

Virtual memory is a memory management technique that provides a necessary abstruction of the storage resources by using a data structure namely page table. Page table actually acts like a bridge between physical memory and virtual memory. Memory Management Unit (MMU) is a hardware unit having all the memory refferences and performing the translation of virtual memory addresses to physical memory addresses.

When program accesses memory, Memory Management Unit (MMU) do the following process:

takes address -> finds an entry in a page table -> gets physical address

Table 4: Workflow of Memory Management Unit

That entry, however, may not exist in the page table. In that case CPU will raise an exception called a **page fault**. There are three types of page faults that may happen:

1. **Minor page fault**: occurs when page table entry should exist, but corresponding page wasn't allocated or page table entry wasn't created. For example, Linux do not allocate mmapped pages immediately, but wait until first page access which causes minor page faults.
2. **Major page fault**: occurs when it is required to read from disk. It may be caused by accessing memory-mapped file or when process memory was paged-out onto disk swap.
3. **Invalid page fault**: occur when application access memory at invalid address or when segment permissions forbid such access. For example writing into text segment, which is usually disallowed. In this case OS may raise SIGSEGV signal.
    - A special case of invalid page faults is copy-on-write fault which happens when forked process tries to write to a parent's memory. In this case, OS copies page and sets up a new mapping for forked process.

I have prepared a systemtap script to trace the major and minor pagefaults generated in the system. The result is shown in the Figure 8.

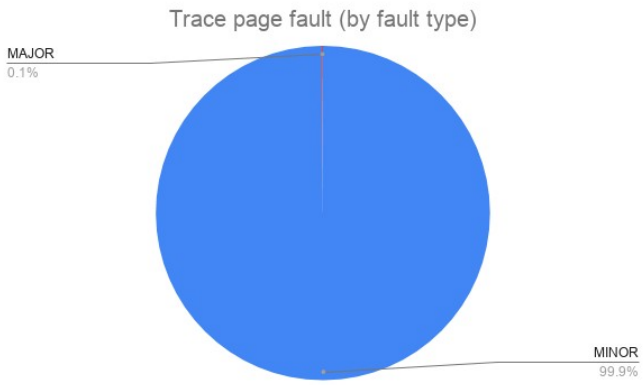

Figure 8: Systemtap script output for tracing different types of page fault in the system

While tracing the different types of page faults, I have learned about the different fault_flags. I have prepared another systemtap script to trace the different pagefault flags generated in the system. The result is shown in the Figure 9. Also the list of fault flag with the flag value and the reson is attached in Table 5.

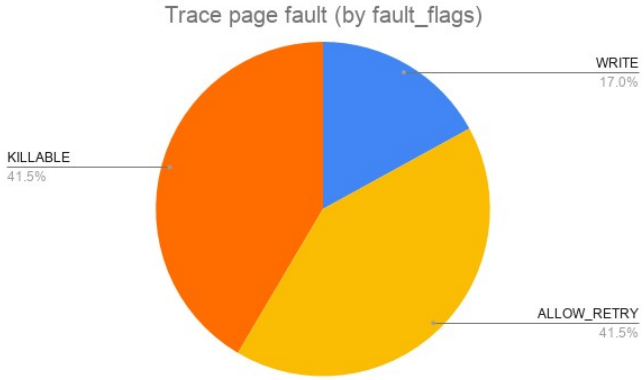

Figure 9: Systemtap script output for tracing page faults with different types of fault_flags in the system

FAULT_FLAG	Value	Reason
FAULT_FLAG_WRITE	0x01	Fault was a write access
FAULT_FLAG_MKWRITE	0x02	Fault was mkwrite of existing pte
FAULT_FLAG_ALLOW_RETRY	0x04	Retry fault if blocking
FAULT_FLAG_RETRY_NOWAIT	0x08	Don't drop mmap_sem and wait when retrying
FAULT_FLAG_KILLABLE	0x10	The fault task is in SIGKILL killable region
FAULT_FLAG_TRIED	0x20	Second try
FAULT_FLAG_USER	0x40	The fault originated in userspace
FAULT_FLAG_REMOTE	0x80	faulting for non current tsk/mm
FAULT_FLAG_INSTRUCTION	0x100	The fault was during an instruction fetch

Table 5: FAULT_FLAG list [4]

ACKNOWLEDGMENT

This project is done as part of the completion of course “ITCS 6144/8144: Operating System Design”, course instructor Professor Dr. Dong Dai. I have taken this course in Fall 2019 semester, as part of my PhD degree at UNC Charlotte.

REFERENCES

[1] *Wiki Page: https://en.wikipedia.org/wiki/Operating_system*

[2] Architecture of systemtap: a Linux trace/probe tool: https://sourceware.org/systemtap/archpaper.pdf

[3] Dynamic Tracing with DTrace & SystemTap: https://myaut.github.io/dtrace-stap-book/

[4] Linux Source Code, mm.h file: https://github.com/torvalds/linux/blob/master/include/linux/mm.h

[5] SystemTap Tapset Reference Manual: https://sourceware.org/systemtap/tapsets/

[6] Brendan's blog on "Using SystemTap": dtrace.org/blogs/brendan/2011/10/15/using-systemtap/

[7] Systemtap Examples: https://sourceware.org/systemtap/examples/index.html#process/strace.stp

[8] SystemTap Beginners Guide: https://lrita.github.io/images/posts/systemtap/SystemTap_Beginners_Guide.pdf