Teaching Operating Systems Concepts with SystemTap

Darragh O'Brien
School of Computing
Dublin City University
Glasnevin
Dublin 9, Ireland
dobrien@computing.dcu.ie

ABSTRACT

The study of operating systems is a fundamental component of all undergraduate computer science degree programmes. Making operating system concepts concrete typically entails large programming projects. Such projects traditionally involve enhancing an existing module in a real-world operating system or extending a pedagogical operating system. The latter programming projects represent the gold standard in the teaching of operating systems and their value is undoubted. However, there is room in introductory operating systems courses for supplementary approaches and tools that support the demonstration of operating system concepts in the context of a live, real-world operating system. This paper describes an approach where the Linux monitoring tool SystemTap is used to capture kernel-level events in order to illustrate, with concrete examples, operating system concepts in the areas of scheduling, file system implementation and memory management. For instructors and students (where often for the latter seeing is believing) this approach offers an additional simple and valuable resource for solidifying understanding of concepts that might otherwise remain purely theoretical.

KEYWORDS

SystemTap, operating system, scheduling, file system, memory management

ACM Reference format:
Darragh O'Brien. 2017. Teaching Operating Systems Concepts with SystemTap. In *Proceedings of ITiCSE '17, Bologna, Italy, July 3-5, 2017,* 6 pages.
DOI: http://dx.doi.org/10.1145/3059009.3059045

1 INTRODUCTION

The study of operating systems plays a key role in undergraduate computer science degree programmes. Essential learning outcomes are delivered by operating systems related projects. The latter traditionally involve programming extensions in the areas of process management, file system support and memory management to a pedagogical operating system [2, 7]. Such programming projects represent the gold standard in the teaching of operating systems and their value is undoubted. However, there is room in introductory operating systems courses for supplementary approaches and tools that support the demonstration of operating system concepts in the context of a live, real-world operating system.

This paper describes how SystemTap [3, 4] can be applied to both demonstrate and explore low-level behaviour across a range of system modules in the context of a real-world operating system. SystemTap scripts allow the straightforward interception of kernel-level events thereby providing instructor and students alike with concrete examples of operating system concepts that might otherwise remain theoretical. The simplicity of such scripts makes them suitable for inclusion in lectures and live demonstrations in introductory operating systems courses.

This paper is structured as follows: In Section 2 SystemTap and its capabilities are briefly introduced. In Section 3 we apply SystemTap to capture scheduling-related events and to demonstrate context switching and multitasking. In Section 4 we employ SystemTap to intercept and record I/O events across a selection of file system implementations. An explanation of such events necessitates an understanding of the underlying file systems and the differences between them. In Section 5 we use SystemTap to explore page table allocation under Linux's memory manager. Early informal evaluation results are reported in Section 6 and Section 7 concludes the paper.

2 SYSTEMTAP

SystemTap allows the dynamic monitoring of Linux kernel events. A SystemTap script is compiled to a module that is loaded into the kernel where it gathers data that is fed back to the user. Scripts typically consist of a set of handlers that fire when kernel events of interest occur. For example, the script presented in Listing 1 monitors the return status of each invocation of the open system call and, for each successfully opened file, displays both the name of the file and the name of executable that opened it.

```
# Listing 1
probe syscall.open.return {
  if ($return == -1) next
  printf("%s opened %s\n", execname(),
    kernel_string($filename))
}
```

The combination of an event plus handler is known as a *probe*. Several event types and filtering options exist to allow a script to be tailored to a fine-grained set of events of specific interest to the user. SystemTap comes with an extensive collection of examples as well as reusable components (known as *tapsets*) for inclusion in user-defined scripts.

ITiCSE '17, July 3-5, 2017, *Bologna, Italy*
 ISBN 978-1-4503-4704-4/17/07…$15.00.
DOI: http://dx.doi.org/10.1145/3059009.3059045

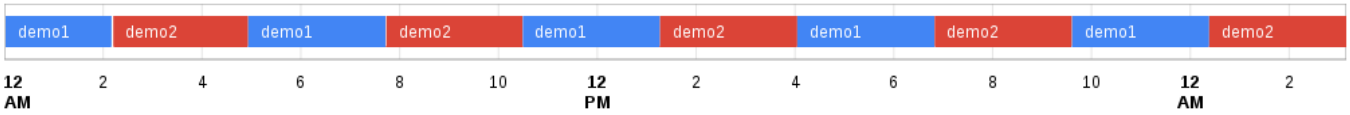

Figure 1: CPU bound processes

3 SCHEDULING

Every introductory operating systems module touches on the areas of process management, scheduling, multitasking, multithreading, etc. With SystemTap it is a straightforward task to monitor CPU on/off events i.e. the reassignment of the CPU from one schedulable entity (process or thread) to another. For example, the script excerpt presented in Listing 2 intercepts and timestamps (in microseconds) every CPU reassignment involving a process of interest (defined by the user in `target_names`). After a predefined number of recordings (`MAX_SAMPLES`) the script exits.

```
# Listing 2
probe scheduler.cpu_on
{
 prev_task_name = task_execname(task_prev)
 next_task_name = task_execname(task_current())

 # Only record target processes
 if (!(prev_task_name in target_names ||
       next_task_name in target_names)) {
         next
 }

 # Record process coming off CPU
 if (prev_task_name == currently_on) {
   timestamps[num_samples, prev_task_name] =
     gettimeofday_us()
   num_samples++
 }

 # Record process coming on CPU
 if (next_task_name in target_names) {
   timestamps[num_samples, next_task_name] =
     gettimeofday_us()
   currently_on = next_task_name
   num_samples++
 }

 # Collect MAX_SAMPLES samples
 if (num_samples >= MAX_SAMPLES) {
   exit()
 }
}
```

3.1 CPU bound processes

In a first experiment the above script is executed while concurrently running two instances (*demo1* and *demo2*) of a CPU bound process (each process simply spins in an infinite loop). An excerpt of the script output (simply a dump of the contents of the `timestamps` associative array) is presented below (the number on the right is the normalised time (in microseconds) at which the corresponding process came on/off the CPU):

```
demo1, 0
demo1, 7809
demo2, 7937
demo2, 17764
demo1, 17767
demo1, 27766
demo2, 27828
demo2, 37765
```

Translating the above data into a format suitable for Google Charts (with microseconds translated to seconds) yields the timeline presented in Figure 1. As is evident from the timeline the two CPU bound processes dominate CPU usage and are allocated roughly equal time on the CPU. (Those periods in the timeline when neither process is executing are too short to be visible in the figure.)

Although a simple exercise the above example illustrates graphically and *with real data* the concept of preemptive multitasking (preemptive because each process must be forcefully evicted from the CPU since it is *always* in a runnable state). In this case the script was executed on a virtual machine with a single CPU. This simple script could be extended to produce scheduling timelines for a (virtual) machine with multiple CPUs thereby demonstrating the concept of *parallel* execution.

3.2 I/O bound processes

In a second experiment a slight variation on the script presented in Listing 2 is executed where scheduling data on *all* processes is collected (rather than only on a predefined set). An excerpt of sample output is presented below:

```
systemd-journal, 0
systemd-journal, 256
rcu_sched, 257
rcu_sched, 260
ksoftirqd/0, 261
ksoftirqd/0, 264
vlc, 264
vlc, 312
```

Translating the above data into a format suitable for Google Charts (again with microseconds translated to seconds) yields the timeline presented in Figure 2. As is evident from the timeline the period an I/O bound process spends on the CPU before blocking is shorter than that of a CPU bound process. We also see the `swapper` process make an appearance. This is Linux's *idle task* i.e. a process that is executed when there are no other runnable processes (as may happen on a lightly loaded machine). Although not arising in this example, occasionally Linux *appears* to take the CPU from a process only to immediately reassign it to the same process. This is rather the successive scheduling of *threads* within the same process.

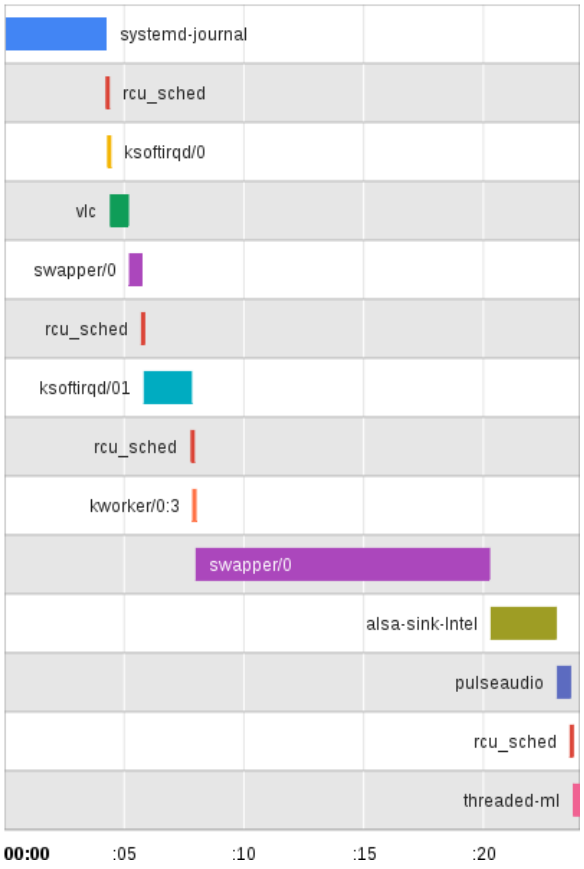

Figure 2: I/O bound processes

Extending the script to differentiate between threads of the same process would serve to demonstrate the concept of *multithreading*.

4 FILE SYSTEMS

In Section 4.1 the experimental setup is described. Subsequent sections present SystemTap scripts and example experiments for demonstrating file system concepts. In Section 4.2 prefetching is demonstrated. In Sections 4.3 and 4.4 two widely deployed file systems are compared: Ext2 [1] and its journalled successor Ext3 [9]. In Section 4.5 the functioning of log-structured file system NilFS [5] is briefly examined.

4.1 Setup

SystemTap monitors system-wide Linux kernel events. To facilitate straightforward filtering on events of interest the file systems under investigation, namely Ext2, Ext3 and NilFS, are installed to dedicated spare partitions. The physical layout (i.e. block offsets from the start of the partition to each of its logical regions) of each of the Ext2/3 partitions is extracted using `dumpe2fs`. Ext2/3 layouts are respectively depicted in Figures 3 and 4. As can be seen, the essential difference between Ext2 and Ext3 is the journal maintained by the latter in order to efficiently restore file system consistency in the event of a crash.

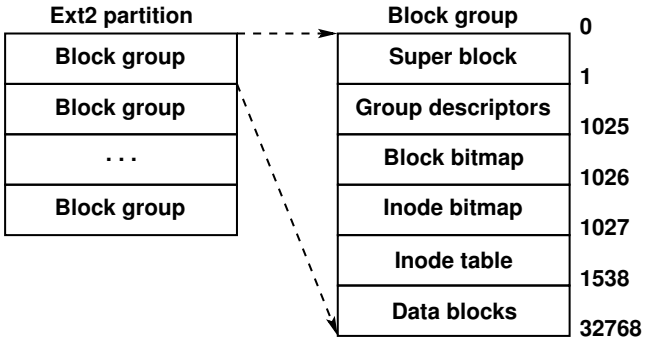

Figure 3: Ext2 partition

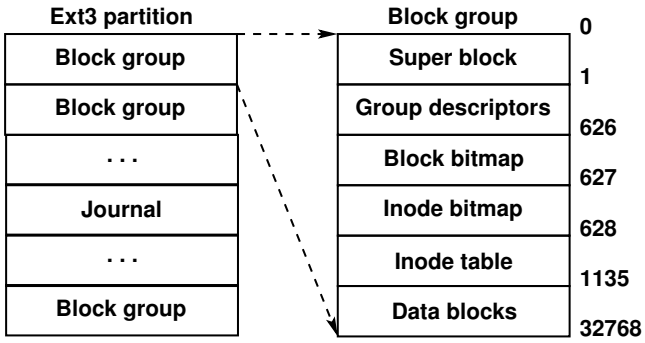

Figure 4: Ext3 partition

In the experiments described below we are interested in observing file system reads and writes. The SystemTap script presented in Listing 3 uses two probes to intercept all I/O request creation and completion events. Sectors are converted to blocks in order to allow straightforward mapping of the script's output to the partition structures presented in Figures 3 and 4. The file systems to be monitored have been installed in partitions `sda6-8`.

4.2 Prefetching

Prefetching [6] is a common I/O technique for improving sequential file access performance. The idea is a simple one: while fetching block *N* from the disk, prefetch blocks *N+R*, where *R* is a read-ahead value specified in blocks. The assumption is that prefetched blocks will be requested in the near future. Those requests can be satisfied from the block cache and costly I/O operations are thus reduced.

While the SystemTap script presented in Section 4.1 is running, a Python program is executed that uses the following code excerpt to issue one hundred read requests for each of one hundred contiguous blocks in a file on the Ext3 file system:

```
with open('/ext3/4mb.img', 'r') as f:
    for i in range (0, 100):
        f.read(4096)
```

In response the SystemTap script reports the following eight I/O operations where each has been annotated with the total number of blocks read after its completion:

```
C : 3360 : 1  : R : 1
```

```
# Listing 3
probe ioblock.request {
  if (!size) next
  if (devname != "sda6" &&
      devname != "sda7" &&
      devname != "sda8") next

  # Convert from sectors to blocks
  sblock = sector / 8
  eblock = (sector + size / 512 - 1) / 8
  blocks = eblock - sblock + 1

  # Note request for matching with completion
  sector += lbas[devname] + size / 512
  queued[sector] = 1
  line = sprintf("%u : %u : %s",
    sblock, blocks, bio_rw_str(rw))
  bios[sector] = line
}

probe ioblock.end {
  if (!queued[sector]) next

  # Report completion
  printf("C : %s\n", bios[sector])
  delete queued[sector]
  delete bios[sector]
}
```

```
C : 3361 : 4  : R : 5
C : 3365 : 8  : R : 13
C : 3373 : 16 : R : 29
C : 3389 : 32 : R : 61
C : 3421 : 32 : R : 93
C : 3453 : 32 : R : 125
C : 3485 : 32 : R : 157
```

Several file system concepts are demonstrated by this simple exercise:

- One hundred read requests, each for a single block of size 4096 bytes, thanks to prefetching translates to far fewer I/O operations, eight in this case.
- As the Linux I/O subsystem grows increasingly confident that a file is being processed sequentially it steadily increases the read-ahead value from 1 to a maximum value of 32 blocks.
- Linux implements asynchronous read-ahead i.e. blocks 126-157 are prefetched despite the fact that block 126 is never requested by the program.
- Running the same Python program immediately after completion of the first run results in zero reported I/O operations: all reads are satisfied from the block cache.

4.3 Ext2

While the SystemTap script presented in Section 4.1 is running, the following commands are executed (where `sync` forces changed blocks to be written to disk):

```
$ echo "Hello world!" > /ext2/hello.txt ; sync
```

In response the SystemTap script reports the following six I/O operations where each has been annotated with an explanation (note that the `debugfs` utility can be used to map blocks 1538 and 6152 to the files to which they belong):

```
C :    1 : 1 : W : update group descriptor
C : 1025 : 1 : W : update block bitmap
C : 1026 : 1 : W : update inode bitmap
C : 1027 : 1 : W : update inode table
C : 1538 : 1 : W : add new dirent to root dir
C : 6152 : 1 : W : write "Hello world!"
```

Several file system concepts are demonstrated by this simple exercise:

- Creating and writing to a file under Ext2 translates to several distinct low-level I/O operations across various locations in the file system as both new data and metadata must be written to disk.
- A system crash during the execution of the required I/O operations will leave the file system in an inconsistent state.
- Omitting the call to `sync` results in an observable delay before SystemTap reports I/O activity. This demonstrates Linux's write-back block cache strategy: changed blocks are periodically rather than synchronously written to disk.

4.4 Ext3

A key advantage of a journalling file system is its ability to efficiently restore a file system to a consistent state after a system crash. To this end, before committing file system changes to disk they are written as a contiguous transaction to a journal. During reboot any intact journal transactions that were not successfully committed to disk are replayed. Any half-written journal transactions are simply ignored.

Ext3 supports several journalling strategies. The default (and the one studied below) is metadata-only journalling (known as *ordered mode* journalling) in which data blocks are written to disk before metadata changes are journalled.

While the SystemTap script presented in Section 4.1 is running, the following commands are executed:

```
$ echo "Hello world!" > /ext3/hello.txt ; sync
```

In response the SystemTap script reports the following 13 I/O operations where each has been annotated with an explanation (note again that the `debugfs` utility can be used to map blocks 1539, 1245783-1245789 and 1135 to the files to which they belong):

```
C :    1539 : 1 : W : write "Hello world!"
C : 1245783 : 1 : W : update journal
C : 1245784 : 1 : W : update journal
C : 1245785 : 1 : W : update journal
C : 1245786 : 1 : W : update journal
C : 1245787 : 1 : W : update journal
C : 1245788 : 1 : W : update journal
C : 1245789 : 1 : W : update journal
C :       1 : 1 : W : update group descriptor
C :     626 : 1 : W : update block bitmap
C :     627 : 1 : W : update inode bitmap
```

```
C :     628 : 1 : W : update inode table
C :    1135 : 1 : W : add new dirent to root dir
```

Several file system concepts are demonstrated by this simple exercise:

- Creating and writing to a file under Ext3 translates to several distinct low-level I/O operations across various locations in the file system as new data, journal entries and metadata must be written to disk.
- Although writes to a journal are contiguous, maintaining a journal incurs an observable overhead.
- Ext3's default journalling strategy journals only metadata updates.
- A system crash during the execution of journal updates means metadata changes are never committed to disk as incomplete journal transactions will be ignored at boot time and, although some data is lost, the file system remains consistent.
- A system crash during the execution of metadata updates is recoverable: at boot time the journal entries are replayed and the file system is restored to a consistent state.

4.5 NilFS

In a final experiment a third partition was installed with log-structured file system NilFS [5]. A log-structured file system aims to optimise file system writes [8]. To this end, all writes, whether data or metadata, are simply appended to a log that expands across the disk. The assumption is that as system memory capacities increase most reads will in future be satisfied from the block cache and thus file systems should target performance-hurting writes.

While the SystemTap script presented in Section 4.1 is running, the following commands are executed:

```
$ echo "Hello world!" > /nilfs/hello.txt ; sync
```

In response the SystemTap script reports the following single I/O operation:

```
C : 269566 : 15 : W : append data and metadata to log
```

Creating and writing to a file under NilFS translates to a single, multi-block contiguous write to the head of the log on the disk.

5 MEMORY MANAGEMENT

The Linux process address space structure is depicted in Figure 5. From the bottom up: The text section holds machine executable code, global data resides in the data section, runtime memory requirements are served from the heap and the stack supports procedure call and return. Assuming a 32-bit architecture the size of the address space is 4GB (2^{32} bytes) with the top quarter (1GB) reserved for the kernel (not shown in Figure 5). Under virtual memory the size of the process address space is independent of the size of the physical address space (i.e. the quantity of physical RAM installed on the machine). Virtual addresses are 32 bits wide and run from `0x00000000` to `0xFFFFFFFF`.

The Linux memory manager is responsible for translating virtual to physical addresses. In the simplest case this translation is managed by a hierarchical structure consisting of a single page directory which points to a set of page tables with each of the latter pointing to physical page frames. (A hierarchical structure that

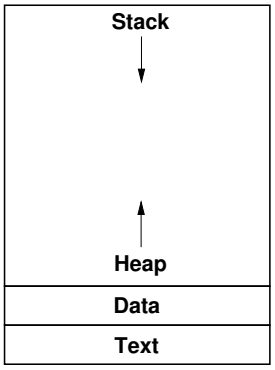

Figure 5: Process address space

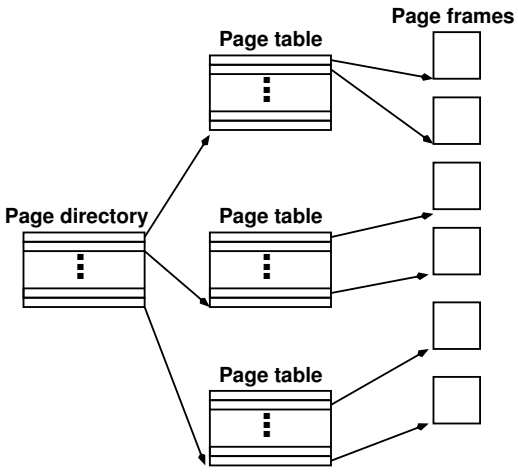

Figure 6: Page directory, page table, page frame hierarchy

expands in response to demand is more memory-efficient than a single flat structure.) This structure is depicted in Figure 6.

There are 1024 entries in a page directory and in each page table. Each page frame is 4096 bytes in size. Thus the described hierarchical structure is capable of mapping the full 4GB ($1024 * 1024 * 4096 = 4GB$) address space. Each page table is capable of mapping 4MB ($1024 * 4096 = 4MB$) of the process address space.

For a "small" process the low end of the address space (text, data and heap sections) can be expected to be mapped by a single page table. Similarly, the high end of the address space (stack) is assumed to be mapped by another page table. Thus we would expect the entire process address space of a "small" process to be mapped by a total of two page tables. Below we use a SystemTap script to confirm that this is indeed the case.

The script presented in Listing 5 hooks the return from the `__pte_alloc()` function in order to intercept all page tables created in the context of the *hello* process whose source code is presented in Listing 4.

```
# Listing 4
int main() {
 printf("Hello world!\n");
}
```

```
# Listing 5
probe kernel.function("__pte_alloc").return
{
 if (execname() != "hello") next

 printf("Faulting address: 0x%08x\n", $address)
 task = task_current()
 mm = task->mm
 nr_ptes = atomic_long_read(&mm->nr_ptes)
 printf("Page tables now: %d\n", nr_ptes)
}
```

```
# stap -g alloc.stp -c "./hello"
Hello world!
Faulting address: 0x080f4f88
Page tables now: 2
```

The above output confirms that two page tables are indeed sufficient to map the process address space of the *hello* process. Interestingly we see that although two page tables are required, only a single call to `__pte_alloc` is intercepted. The first call to `__pte_alloc` occurs in the context of the parent process (it is the parent process that initialises a child process's stack) and is thus missed by our script. In a second experiment, we add a 4MB global array to our process address space's data section as follows:

```
# Listing 6
char x[1024*4096];
int main() {
  printf("Hello world!\n");
}
```

Running the SystemTap script in the context of the "larger" process confirms that one additional page table is now required to map the enlarged address space:

```
# stap -g alloc.stp -c "./hello"
Hello world!
Faulting address: 0x080f4f88
Page tables now: 2
Faulting address: 0x084f5c54
Page tables now: 3
```

6 EVALUATION

The effectiveness of the proposed SystemTap-enhanced approach to demonstrating operating systems concepts has yet to be formally and quantitatively compared to other approaches. However, feedback from annual anonymous student surveys is consistently positive. Student comments indicate the module rates among the most popular of the degree programme. Of particular note is the high level of student engagement with the module. In class presentation of material and demonstrations are described as "interactive", "fun", "engaging" and "interesting". Students commented positively on the "beneath the bonnet" treatment of the subject. Students also indicated the module motivated them to carry out further research in the area of operating systems in their own time.

7 CONCLUSIONS

The study of operating systems plays a critical role in undergraduate computer science degree programmes. Operating systems concepts covered in lectures are traditionally made concrete through the undertaking of significant programming projects. While such programming projects are invaluable and remain the gold standard in delivering the desired operating system learning outcomes, there is room for supplementary approaches and tools that support the classroom-/lab-based demonstration of operating system concepts in the context of a live, real-world operating system.

SystemTap is a Linux monitoring tool. This paper has demonstrated how simple SystemTap scripts can be used to intercept kernel events which can then be mapped to operating system concepts which might otherwise have remained purely theoretical. Examples in the paper covered topics traditionally covered in a module on operating systems: scheduling, file system implementation, and memory management.

To date the SystemTap scripts described in this paper have been used primarily for demonstration purposes only. Possible extensions were suggested to a selection of the SystemTap scripts presented. It would be interesting to create and evaluate the effectiveness of a series of SystemTap programming exercises for students. Requiring students to write SystemTap scripts and explain their output would yield valuable learning outcomes:

- Selecting appropriate events to hook demonstrates an understanding of the Linux kernel,
- explaining SystemTap output demonstrates an understanding of operating system concepts,
- as a general purpose Linux monitoring tool with a vast array of applications beyond those considered in this paper, SystemTap programming skills are valuable in their own right.

For instructors and students (and particularly for the latter group for whom, often, seeing is believing) SystemTap demonstrations, whether in a classroom or laboratory setting, have a valuable role to play in solidifying understanding of operating system concepts.

REFERENCES

[1] Remy Card, Theodore Ts'o, and Stephen Tweedie. 1994. Design and Implementation of the Second Extended Filesystem. In *Proceedings of the First Dutch International Symposium on Linux*. State University of Groningen, Netherlands, 1–6.

[2] Wayne A. Christopher, Steven J. Procter, and Thomas E. Anderson. 1993. The Nachos Instructional Operating System. In *Proceedings of the USENIX Winter 1993 Conference Proceedings on USENIX Winter 1993 Conference Proceedings (USENIX'93)*. USENIX Association, Berkeley, CA, USA, 4–4. http://dl.acm.org/citation.cfm?id=1267303.1267307

[3] Frank Ch. Eigler. 2006. Problem Solving with SystemTap. In *Proceedings of the Ottawa Linux Symposium*. 261–268.

[4] Frank Ch. Eigler. 2015. SystemTap Tutorial. (2015). https://sourceware.org/systemtap/tutorial.pdf

[5] Ryusuke Konishi, Yoshiji Amagai, Koji Sato, Hisashi Hifumi, Seiji Kihara, and Satoshi Moriai. 2006. The Linux Implementation of a Log-Structured File System. *ACM SIGOPS Operating Systems Review* 40, 3 (2006), 102–107.

[6] R. H. Patterson, G. A. Gibson, E. Ginting, D. Stodolsky, and J. Zelenka. 1995. Informed Prefetching and Caching. *SIGOPS Oper. Syst. Rev.* 29, 5 (Dec. 1995), 79–95. DOI:http://dx.doi.org/10.1145/224057.224064

[7] Ben Pfaff, Anthony Romano, and Godmar Back. 2009. The Pintos Instructional Operating System Kernel. *SIGCSE Bull.* 41, 1 (March 2009), 453–457. DOI: http://dx.doi.org/10.1145/1539024.1509023

[8] Mendel Rosenblum and John K Ousterhout. 1992. The design and implementation of a log-structured file system. *ACM Transactions on Computer Systems (TOCS)* 10, 1 (1992), 26–52.

[9] Theodore Y. Ts'o and Stephen Tweedie. 2002. Planned Extensions to the Linux Ext2/Ext3 Filesystem. In *Proceedings of the FREENIX Track: 2002 USENIX Annual Technical Conference*. USENIX Association, Berkeley, CA, USA, 235–243. http://dl.acm.org/citation.cfm?id=647056.715922