Educational operating system experiments

Sara Salahi and Stephen Tarzia

March 16, 2007

1 Introduction

Teaching the inner-workings of an operating system (OS) is a daunting endeavor. Most operating systems are composed of large, complex code that has taken hundreds of programmer-years to write. OS concepts include the concepts associated with various subsystems within the OS as well as how those subsystems are interconnected. The complexity of subsystem interconnections within an OS makes it difficult for instructors to choose a starting point. Many times, instructors are left with a choice between two unsatisfactory options: 1) teaching one small, but complex subsystem of a real OS or 2) aiming to give their students a more thorough understanding by having them study a "toy" OS. The most common choice is to use detailed programming assignments to reinforce the general principles covered in lectures and reading material. However, kernel programming can be an inefficient learning method because it involves many uninteresting implementation details.

We are developing OS experiments covering several topics, using instrumentation tools and prepared scripts and C applications. Our goal is to build intuition in the students through observation of real OS behavior. Our emphasis is on the aspects of operating systems most relevant to application programmers (rather than OS designers). A gray-box Information and Control Layer (ICL) [1] is a software interface that lets an external user control and gather information about the state of a closed system. The idea is to allow students to approach the OS as gray-box researchers: discovering its properties through careful manipulation and observation. We call this an *experimental* approach to OS education, and we believe that it greatly improve students' understanding of OS principles. Our approach is applicable to all OS subsystems, but we begin with the virtual memory system.

Our hypothesis is that OS experimentation is an engaging and effective way to improve students' understanding of OS principles. The skills developed in our experiments are also useful in debugging systems applications; thus, a secondary goal is to expose students to useful software development practices, both in performance optimization and debugging. Our final experiments will be tested on current undergraduate students taking an introductory course in operating systems. So far, we have found that instrumentation is handled admirably

by Sun's DTrace and that the OS state can be effectively controlled by user applications.

2 Instrumentation tools

2.1 DTrace

We rely heavily on Sun's DTrace [2]. This kernel instrumentation tool allows users to insert probes that monitor and reveal the function calls that take place within a real open source OS. We use DTrace primarily for kernel function tracing and for collecting of system call and I/O frequency and timing statistics.

In the Opensolaris Student Guide [7], the authors have made available a script that uses DTrace to reveal all of the function calls that take place after a page fault until the page fault handler returns. Our second experiment is based on this script; the output is made more intuitive and informative through annotation of obscure kernel function names and omission of unimportant details.

2.2 `prstat` and `top`

In addition to DTrace, Solaris offers several high-level system monitoring tools. `prstat` lists active processes in a fashion similar to the Unix `top` utility. This real-time data is useful in scheduling and virtual memory experiments.

3 Control mechanisms

3.1 User applications

In one of our experiments the students observe the behavior of two simple C programs. These are meant to demonstrate certain memory access patterns. In particular, the superiority of row-wise matrix access to column-wise is shown. In another experiment we opt to observe the behavior of the complex GNU `emacs` text editor.

3.2 Projects and `rcapd`

Projects are a Solaris feature that allows one to set runtime parameters for a group of processes [4]. We use it's resource limiting feature to create a restricted memory environment in which to run the paging experiments. These resource limits can be assigned on a per-user basis or at process load-time. Limits are not enforced by the kernel. Instead, the `rcapd` daemon periodically enforces the resource caps.

3.3 Kernel parameters

SunOS allows certain kernel parameters to be modified at runtime. Figure 1 shows those parameters which we consider useful for future experiments. So far

parameter	description
physmem	Reduce amount of system memory to test paging effects.
lotsfree	Serves as the initial trigger for system paging to begin.
desfree	Specifies the preferred amount of memory to be free at all times on the system.
minfree	Specifies the minimum acceptable memory level. When memory drops below this number, system biases allocations toward allocations necessary to successfully complete pageout operations or to swap processes completely out of memory. Either allocation denies or blocks other allocation requests.
throttlefree	Specifies the memory level at which locking memory allocation requests are put to sleep, even if the memory is sufficient to satisfy the request.
handspreadpages	The distance between the first hand and the second hand (as part of the clock algorithm used to reclaim pages when memory is low).
rechoose_interval	Specifies the number of clock ticks before a process is deemed to have lost all affinity for the last CPU it ran on.

Figure 1: Solaris kernel parameters

we have not adjusted any of these.

4 Opensolaris source browser

Sun recently released the source code for most of it's SunOS kernel, the core of the Solaris operating system. A very good code browser is available online [6]. We have found this resource helpful in our own investigation of SunOS's virtual memory functions. In keeping with the gray-box approach, we believe that the basic principles of OS operation can be understood without parsing the source code. However, interesting details in the code should be pointed out to students, and the curious student should be able to browse the source code if desired.

5 Experimental setup

Students will have SSH or local access to a Solaris machine and a repository of scripts and precompiled binaries to run in each experiment. The students have a straightforward procedure to follow. Generally they will run experiment scripts, record the output, vary the state of the system, repeat the experiment scripts, and answer some questions or provide a general writeup about what

they've seen. The procedure and results for two virtual memory experiments are included in the appendices.

As a "pre-lab" portion to our specific page fault experiments, students should have been exposed to an introductory level of paging in a classroom setting. They will be told how much RAM is available and how large the pages are, and with this knowledge they should be able to calculate the matrix size necessary for optimizing the performance of the program.

6 Future Work

Additional experiments on scheduling, synchronization, the filesystem, etc. will be created.

References

[1] A. C. Arpaci-Dusseau , R. H. Arpaci-Dusseau, Information and control in gray-box systems, In *Proc. 18th ACM Symposium on Operating Systems Principles*, October 21-24, 2001.

[2] B. M. Cantrill, M. W. Shapiro, and A. H. Leventhal. Dynamic instrumentation of production systems. In *Proc. USENIX Annual Technical Conference*, pages 15-28, June 2004.

[3] Solaris Dynamic Tracing Guide. Sun Microsystems, Inc. 2005. `http://docs.sun.com/app/docs/doc/817-6223`. Part No: 817–6223–11

[4] System Administration Guide: Advanced Administration. Sun Microsystems, Inc. 2006. `http://docs.sun.com/app/docs/doc/817-0403`. Part No: 817–0403–13

[5] Solaris Tuneable Parameters Reference Manual. Sun Microsystems, Inc. 2006. `http://docs.sun.com/app/docs/doc/817-0404`. Part No: 817–0404–13.

[6] Opensolaris source code browser. Sun Microsystems, Inc. 2005. `http://src.opensolaris.org/source/xref/onnv/onnv-gate/usr/src/uts/`

[7] Introduction to Operating Systems: A Hands-On Approach Using the OpenSolaris Project. Sun Microsystems, Inc. 2006. `http://opensolaris.org/os/community/edu/curriculum_development/`. Part No: 819–5580–10

A Experiment 1: Slow matrix access

In this experiment, students discover a matrix access performance bug.

A.1 Student instructions

In this experiment you will use Solaris's resource capping daemon, `rcapd`, to soft-limit the resident size of your programs to 128 MB. Every five seconds, `rcapd` checks process memory usage and reclaims pages from processes exceeding their limit. We call this a *soft* resource limit because it is not enforced by the kernel on allocation; a process may temporarily hold more of the resource than allowed before the capping daemon notices and enforces the policy.

In Solaris, a *project* defines a resource limitation policy applied collectively to a set of processes. We have defined the `sandbox` project with a resident size cap of 128 MB. A process may be started under the `sandbox` project like so:
`$ newtask -p sandbox` *`[command goes here]`*

Another useful tool is the `time` command. This prints the real, user, and system time for a process after it terminates: `$ time` *`[command goes here]`*

In this experiment you will examine two program that do *scalar multiplication* on a matrix; i.e. it applies a constant factor to every element of the matrix.

A.1.1 Procedure

1. Start the system monitor to view real-time process statistics. Observe the status and memory usage of running processes. In particular, the `SIZE` and `RES` indicate the virtual memory size and resident size (the amount actually in physical memory), respectively. At top, the size of physical memory is given:
   `$ top`

2. Run the first scalar multiply implementation, `scalar1`, with a size 10000 by 1000 matrix:
   `$ time newtask -p sandbox ./scalar1 100000 1000`

3. Run the second implementation, `scalar2`:
   `$ time newtask -p sandbox ./scalar2 100000 1000`
   Let it run for three minutes, then interrupt it with `CTRL-C`.

   The second implementation contains a *performance bug*. It is not incorrect, but because the programmer did not have a good understanding of the underlying OS, it runs very inefficiently.

4. To better understand `scalar2`'s behavior, it is a good idea to run some monitoring tools. Specifically, run `top` in a second terminal while you repeat the previous step in the first terminal. You can tap `spacebar` to update `top`'s statistics more frequently.

You should have noticed something unusual about `scalar2`; it remains asleep during most of its execution. You can also see that its virtual memory size is 384 MB, much larger than the 128 MB cap being enforced. You should see the process's resident memory size always increasing and being periodically truncated down to 128 MB by `rcapd`. These facts indicate that excessive pagefaults may be the source of slowdown.

5. You can use the `pfault_timeline.d` DTrace script to trace the occurrence of pagefaults during `scalar1` and `scalar2`'s execution.
   `$ ./pfault_timeline.d`
   Interrupt it with `CTRL-C` to stop tracing and display the results. You may want to start tracing after the initial matrix allocation is complete: these allocation pagefaults are uninteresting.

   This should verify our hypothesis that pagefaults were the cause of slowdown.

6. Run both `scalar1` and `scalar2` on a square 10000 by 10000 matrix.
   ```
   $ time newtask -p sandbox ./scalar1 10000 10000
   $ time newtask -p sandbox ./scalar2 10000 10000
   ```

   You may be surprised to find that both implementations run relatively efficiently on this matrix.

7. Compare `scalar1.c` and `scalar2.c` to find the performance bug in `scalar2.c`.

A.1.2 Discussion

Assuming 128 MB physical memory, 4 kB pages, and a 10000 by 1000 matrix initially allocated but not in physical memory:

1. Was the system idle or busy during your run? How did this affect your results?

2. What is the fewest number of pagefaults needed to perform scalar multiplication?

3. Assuming no competing processes, how many pagefaults will result from an LRU page replacement policy if column-wise access is used? row-wise?

4. Why does `scalar2` experience so many fewer pagefaults for a 10000 by 10000 matrix than for a 100000 by 1000 matrix?
   ANSWER: Column-wise access in 10000 by 10000 matrix is fine because all 10000 pages = 40 MB can fit in physical memory. A 100000 height column requires 100000 pages = 400 MB and thus the top of the column will be paged out when approaching the bottom of the column .

5. Even if we have an abundance of free physical memory and paging never occurs, `scalar1` will be faster than `scalar2`. Why is this?
   ANSWER: There will be more cache hits in the CPU.

Figure 2: Sample output for `scalar1` and `scalar2`

```
-bash-3.00$ time newtask -p sandbox ./scalar1 100000 1000
Allocating 100000 by 1000 float matrix...
Allocation complete.  Sleeping 10 seconds...
Starting scalar multiplication...
0% ---------------- progress ---------------- 100%
@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@
Multiplication complete.

real    1m22.460s
user    0m8.509s
sys     0m3.506s
-bash-3.00$ time newtask -p sandbox ./scalar2 100000 1000
Allocating 100000 by 1000 float matrix...
Allocation complete.  Sleeping 10 seconds...
Starting scalar multiplication...
0% ---------------- progress ---------------- 100%
@^C

real    1m35.970s
user    0m7.562s
sys     0m4.585s
```

Figure 3: Sample `pfault_timeline.d` output for `scalar1`

```
scalar1                                             11920
        value  ------------ Distribution ------------ count
        < 1000 |                                      0
         1000 |@@                                     4497
         2000 |@@                                     4250
         3000 |                                       1
         4000 |@@                                     5151
         5000 |@@                                     5842
         6000 |@@                                     6121
         7000 |@@                                     4801
         8000 |                                       0
         9000 |                                       0
        10000 |                                       0
        11000 |                                       0
        12000 |                                       2
        13000 |@@                                     5375
        14000 |@@@                                    6599
        15000 |@@                                     5671
        16000 |@@@                                    6306
        17000 |@@                                     4461
        18000 |@@                                     3852
        19000 |@@@                                    7007
        20000 |@@                                     5024
        21000 |@@@                                    6143
        22000 |@@                                     5006
        23000 |                                       1
        24000 |                                       0
        25000 |                                       0
        26000 |                                       0
        27000 |                                       0
        28000 |                                       0
        29000 |                                       0
        30000 |                                       0
        31000 |                                       0
        32000 |                                       0
        33000 |                                       681
        34000 |@@                                     5592
        35000 |@@                                     5631
        36000 |                                       0
```

Figure 4: Sample `pfault_timeline.d` output for `scalar1`

```
scalar2                                             11923
        value  ------------ Distribution ------------ count
         3000 |                                         0
         4000 |@                                        2371
         5000 |                                         0
         6000 |                                         53
         7000 |@@                                       6985
         8000 |@@                                       7321
         9000 |@                                        3450
        10000 |                                         0
        11000 |                                         0
        12000 |                                         238
        13000 |@@                                       6135
        14000 |@                                        3740
        15000 |                                         0
        16000 |                                         929
        17000 |@@                                       6429
        18000 |@@                                       7284
        19000 |@                                        4151
        20000 |                                         0
        21000 |@                                        2701
        22000 |@@                                       6580
        23000 |@                                        4565
        24000 |@                                        2938
        25000 |                                         0
        26000 |                                         0
        27000 |                                         0
        28000 |                                         0
        29000 |                                         0
        30000 |                                         0
        31000 |                                         0
        32000 |@                                        3965
        33000 |@@                                       6204
        34000 |@                                        3666
        35000 |                                         0
        36000 |                                         1744
        37000 |@@                                       7170
        38000 |@@                                       7161
        39000 |@                                        3854
        40000 |                                         0
        41000 |                                         0
        42000 |                                         0
        43000 |                                         0
        44000 |                                         1282
        45000 |@@                                       6988
        46000 |@@                                       6960
        47000 |@@                                       7266
        48000 |@@                                       6927
        49000 |@                                        4077
        50000 |                                         0
        51000 |                                         0
        52000 |                                         0
        53000 |                                         0
        54000 |                                         0
        55000 |                                         0
        56000 |                                         0
        57000 |                                         985
        58000 |@@                                       6784
        59000 |@@                                       6201
        60000 |                                         0
```

A.2 `rcapd` issues

In order to make page faulting more intuitive for students, we generated a sample program that – when run with DTrace – displays page fault trends. We hoped that this sample program would show the students the performance difference (in terms of page faulting) when accessing a matrix column-wise versus accessing the same matrix row-wise. In reality, our test machine had too much RAM (3 GB) to show any instructive page faulting with a single simple program. Therefore, we ran `rcapd` while running this program to cap the amount of RAM available for the matrices at 128 MB. What we found from these trial runs was that `rcapd` did not provide a solid run-time cap on the amount of RAM available resulting in our program acquiring more than 128 MB of RAM and then the `rcapd` daemon reclaiming the memory and then our program acquiring more RAM again and so on. We originally had hoped to model a smaller machine with `rcapd`; instead it was more like modelling a competing process. Nonetheless, the results are instructive.

Also, in a real class laboratory environment, several *projects* should be defined so each concurrently working student has his/her own 128 MB chunk of RAM.

B Experiment 2: Pagefault call tree

In this experiment we show the call trace for a several page faults. We make three types of annotations to the call tree: long comments, short comments, and branch suppression. We aim to instill a high-level understanding by revealing the low level system functions using DTrace with added comments/explanations.

B.1 Student instructions

B.1.1 Procedure

1. Run the verbose tracing script indicating as a parameter the application `emacs` as the target:
   `$ ./pfault emacs`

2. In another terminal, open the `emacs` text editor application:
   `$ emacs -nw`
   The tracing script will print a call tree for the first pagefault occurring in `emacs`. You can see that there is a lot going on behind the scenes when a pagefault occurs. This is why we try to avoid them!

3. Now, repeat the above with the concise, annotated script:
   `$ ./pfault_annotated.d emacs`
   Observe the output from DTrace as `emacs`'s memory is allocated.

4. Again, run the experiment script on `emacs`:
   `$ ./pfault_annotated.d emacs`

5. Now run the spell checker in `emacs`:
   `<alt-x>ispell<enter>`

6. Observe the pagefault that occurs when `emacs` loads the spell-checking code.

B.1.2 Discussion

1. How and why are the two pagefaults different?

2. What kinds of variations in each trace can you see in the similar situations, and what causes these variations?

3. When would you expect the page the be found on the *free list*?

Figure 5: `pfault_annotated.d` output for allocation pagefault

```
CPU FUNCTION
  0  -> pagefault                          <== fault occurred on address = feffb338
  0    -> as_fault                         <== as_fault() handles faults on a given address space.
  0      -> as_segat                       <== as_segat() walks an AVL tree of segment structures looking for a
                                             segment containing the faulting address. If no such segment is found,
                                             the process is sent a SIGSEGV (segmentation violation) signal.
  0      <- as_segat
  0      -> segvn_fault                    <== Segment specific fault handler.  First task is to find the page:
  0        -> anonmap_alloc                <== Allocate an anonymous page: i.e. a non-filesystem pages
  0          -> kmem_zalloc                 <== Kernel allocates zero-filled memory for itself.
  0          <- kmem_zalloc
  0          -> kmem_zalloc                 <== Kernel allocates zero-filled memory for itself.
  0          <- kmem_zalloc
  0        <- anonmap_alloc
  0        -> segvn_faultpage              <== Now that we have the necessary page attributes, search for each
                                             faulting page.
  0          -> page_lookup                <== search for page in page cache
  0          <- page_lookup
  0          -> hat_memload                <== Hardware Address Translation layer:
                                             load the page table entry (PTE) for the page.
  0            -> x86pte_set                <== Create a page table entry.
  0            <- x86pte_set
  0          <- hat_memload
  0        <- segvn_faultpage
  0      <- segvn_fault
  0    <- as_fault                         <== Now virtual address feffb338 is mapped to a valid physical page.
                                             The instruction causing the page fault will be retried and
                                             should now complete successfully.
  0  <- pagefault
```

Figure 6: `pfault_annotated.d` output for regular disk-access pagefault

```
CPU FUNCTION
  1  -> pagefault                          <== fault occurred on address = f92b000
  1    -> as_fault                         <== as_fault() handles faults on a given address space.
  1      -> as_segat                       <== as_segat() walks an AVL tree of segment structures looking for a
                                             segment containing the faulting address. If no such segment is found,
                                             the process is sent a SIGSEGV (segmentation violation) signal.
  1      <- as_segat
  1      -> segvn_fault                    <== Segment specific fault handler.  First task is to find the page:
  1        -> segvn_faultpage              <== Now that we have the necessary page attributes, search for each
                                             faulting page.
  1          -> page_lookup                <== search for page in page cache
  1          <- page_lookup
  1          -> page_lookup                <== search for page in page cache
  1          <- page_lookup
  1          -> kmem_zalloc                 <== Kernel allocates zero-filled memory for itself.
  1          <- kmem_zalloc
  1          -> sdstrategy                 <== The device driver strategy routine is called.
  1          <- sdstrategy
  1          -> swtch                      <== While the page is being read, the thread causing the page fault
                                             blocks via a call to swtch().
  1            -> savectx                  <== Save process context, because another process is going to run.
  1            <- savectx
  1  -> restorectx                         <== Restore process context, because this process is going to run again.
  1  <- restorectx
  1  <- swtch
  1  -> hat_memload                        <== Hardware Address Translation layer:
                                             load the page table entry (PTE) for the page.
  1    -> x86pte_set                        <== Create a page table entry.
  1    <- x86pte_set
  1  <- hat_memload
  1  <- segvn_faultpage
  1  <- segvn_fault
  1  <- as_fault                           <== Now virtual address f92b000 is mapped to a valid physical page.
                                             The instruction causing the page fault will be retried and
                                             should now complete successfully.
```