

Operating Systems Projects -- Minix Revisited¹

James Howatt

Department of Computer Science
Southeastern Louisiana University
Hammond, LA 70402
<jhowatt@selu.edu>

1. Introduction.

I first used Minix [10] in 1992, for reasons elaborated on in [9]. In short, my previous school had a dedicated operating systems lab, and I preferred to use a real operating system instead of a simulator like OSP [7] or NACHOS [5], but not one as complex as Linux [1] (although Nutt [8] may help students navigate that complexity).

My current department has no dedicated OS lab, and most of our students will not partition their hard drives to install Minix. So, for the past four years, I assigned projects that had them use Ben-Ari's Concurrency Interpreter [4], Solaris' semaphores and shared memory utilities, as explained in [6], system calls to manipulate directories, C's address-of operator for determining the layout of a program in memory, etc. Although students learned much from the assignments, it was not the same as having them work inside of an actual OS. But recently, Kees Bot, the Minix system administrator, produced a version [3] that allows a Minix file system to exist as a single Windows '9X file, eliminating the need for hard drive partitioning. Students now need to reserve only 60MB of hard disk space for only one semester to install Minix. (Students using Windows 2000 or later may need to run Minix in a simulator such as Bochs[2] or VMware[11]).

With Minix, students get to work with a complete, operational software engineering artifact (albeit one of educational, rather than industrial, strength). Minix is small enough that students can grasp the "chunks" they need for specific projects, but large enough that they can see that small changes made in one function can detrimentally affect code in a far distant function. Students also learn that desk-checking and walk-throughs are a necessity; when their modified versions of Minix do not boot, there's no debugger to fall back on. (However, they can reboot from an always-saved, stable version to find and correct their problems.)

2. Minix Projects.

For each of the projects described below, students work in groups of three. They can divide the project workload in any way they choose as long as the tasks are reasonably balanced. Any member of a group, however, can be called upon to describe the entire project. Project deliverables include a project report, a project log (a diary of team activities), and a diskette with a Minix file system that contains a bootable image containing their modifications and a `/usr` file system with all of Minix's usual directories, but containing only the files that were modified or added for the project.

When teams write system calls, I supply them with Unix-like "man" pages that describe the syntax and semantics of the calls. The user-level programs that test the system calls must follow the specifications given in the man pages; one team's test program must be able to run on another team's system.

To help students understand Minix's system call structure, I trace for them one execution of the `times(2)` system call, describing the code in each relevant source code file (`.c` and `.h`). Not surprisingly, many of their system calls closely mimic `times(2)`.

2.1 Installation.

In the first week, the teams download and install Minix, modify `/usr/include/minix/config.h` to configure Minix for their platforms, modify `/usr/src/kernel/tty.c` to add their names to the boot banner, and modify `keyboard.c` in the same directory to print a message of their choosing whenever function key six is pressed. This exercise allows them to become familiar with the Minix file system and with recompiling and rebooting the system.

¹ Submitted for publication to SIGCSE Bulletin.

2.2 A Simple System Call.

In the second week each team writes a simple system call to fetch from the memory manager (MM) the running process' process id (PID) and its parent's PID, and writes a user-level program to test the system call. The prototype, from the man page, is:

```
#include <prog2.h> // The team-supplied header file
int GetPIDs (pid_t *Me, pid_t *Mom);
```

To implement this simple system call, the students must: (1) modify one system header file, `minix/callnr.h`, (2) add the header file `prog2.h` to `/usr/include`, (3) add their source code to `/usr/src/lib/other`, (4) put their object files in `/usr/lib/i386/libs.a`, (5) modify `/usr/src/mm/{proto.h,table.c,Makefile}`, and (6) add a source-code file to `/usr/src/mm`.

Their test programs must fork a binary tree of processes at least four deep. The teams are told to use `getpid(2)` and `getppid(2)` to validate their results.

The students find with this assignment that they write relatively little code, but the code that they do write, as shown above, is scattered throughout Minix. They quickly discover the benefits of advanced planning, careful code reading, and `grep(1)` and `find(1)`.

2.3 Semaphore Implementation

Minix does not provide user-accessible semaphores, so the students' third project is to implement ten semaphores in MM (where tasks can be easily blocked and restarted), to write system calls for the semaphore operations, and to write user-level solutions to the producer-consumer problem, using a file as the bounded buffer. The specifications for the system calls are:

```
#include <minix/semaphore.h>
int SemDown (int SemaphoreNumber);
int SemUp   (int SemaphoreNumber);
int SemInit (int SemaphoreNumber, int Value);
int SemStatus (int SemaphoreNumber, int* Value, int* NumBlocked);
```

`SemDown` and `SemUp` perform the P and V operations, resp., on the specified semaphore; `SemInit` sets the specified semaphore to the given value; and `SemStatus` sends back the identified semaphore's current value and the number of processes in its wait queue. No synchronization mechanism is needed to protect the semaphore structures because no other user processes will run while MM is running.

The producer-consumer program will use the semaphores to synchronize accesses to the bounded-buffer (implemented as a file) and necessary counters (also a file). The program will open and initialize the files, initialize the semaphores, and then fork a specified number of producer and consumer processes which produce or consume a specified number of items. In future classes, I plan to have half the teams implement the semaphores and the other half implement a form of shared memory (to be used as the buffer). Then pairs of teams will use each other's system calls in their producer-consumer solution.

2.4 Process Scheduling

For their fourth project, the students add a level to Minix's three-level queuing. Instead of strict round-robin scheduling, the scheduler will split user processes between I/O-bound and CPU-bound process queues, with the I/O-bound processes receiving higher priority. The students must design and write user programs to test their modifications. As a part of the test, the students need to determine what will make a process actually block on I/O, rather than doing a non-blocking transfer to cache. I encourage the students to add necessary fields to the kernel's process table and to implement any system calls they might need to fetch process values as a part of their tests (they write the man page(s) this time). Except for those system calls, the students should have to modify only the scheduling routines and the process table, `struct proc`, in `/usr/src/kernel/proc.c` and `proc.h`.

2.5 Memory and Thrashing

In project 5, instead of modifying or instrumenting Minix's memory manager, which now implements swapping, the teams will combine code reading with testing to determine at what point Minix will start thrashing—spending more time swapping than processing. After they determine the thrashing point, they are to recommend a policy that could be implemented to prevent thrashing. Students may find this project to be easier if they run Minix on older, low-memory machines, or inside of a simulator like Bochs or VMware, where the memory size can be specified.

2.6 File System Attributes

Depending on the amount of time remaining in the semester, I assign the students one or more of the following file system (FS) projects. All are written as user commands.

`fcontig(1)` -- when given a file name, this command will report the degree of contiguity of a file—the degree to which its data blocks fall in consecutively numbered blocks. This gives the students the opportunity to work with i-nodes, and to determine how to write data files in consecutive and non-consecutive blocks. They find that working with a file system on a 1.44 MB floppy greatly simplifies testing.

`fsfrag(1)` – when given a Minix file system, this command reports the degree of external fragmentation. Since Minix uses a bit-map block-management scheme, the students will need to traverse the bit maps for the i-nodes, directories and data blocks. They gain a familiarity with super blocks, bit strings, and C's bit manipulation operators.

`fsifrag(1)` – for a Minix file system, this command reports the least, greatest and average amount of internal fragmentation (unused space in the last data block) in all of the data files. The project gives students the opportunity to work with directories and i-nodes (they are not allowed to use `readdir(3)` except for testing), and with recursion, which lends itself nicely to traversing the directory tree. We assume that the files are written sequentially. If a user opens a new file, seeks to the last byte in the thousandth block, and writes just one byte of data, then no fragmentation would be reported for that file. Additionally, the reported fragmentation does not include unused space in i-node index blocks.

2.7 Resource Usage.

For their final project, the students will implement a variant of the `getrusage(2)` system call, which has a prototype of:

```
#include <sys/time.h>
#include <sys/resource.h>
#include <unistd.h>

int getrusage (int who, struct rusage *usage);

struct rusage {
    struct timeval ru_utime; /* user time used */
    struct timeval ru_stime; /* system time used */
    long ru_maxrss;         /* maximum resident set size */
    long ru_ixrss;          /* integral shared memory size */
    long ru_idrss;          /* integral unshared data size */
    long ru_isrss;          /* integral unshared stack size */
    long ru_minflt;         /* page reclaims */
    long ru_majflt;         /* page faults */
    long ru_nswap;          /* swaps */
    long ru_inblock;        /* block input operations */
    long ru_oublock;        /* block output operations */
    long ru_msgsnd;         /* messages sent */
    long ru_msgrcv;         /* messages received */
    long ru_nsignals;       /* signals received */
    long ru_nvcsw;          /* voluntary context switches */
    long ru_nivcsw;         /* involuntary context switches */
};
```

This is a much more ambitious project than the rest. To collect the needed information, the students must: (1) change FS to count input and output data blocks and count messages sent and received; (2) change MM to count

swaps, messages sent and received, and signals received; and (3) change the kernel to count voluntary and involuntary context switches. Minix does not implement paging, so `ru_minflt` and `ru_maxflt` will be set to zero. The teams also need to implement system calls to retrieve those counts, and test programs to make the system calls. The students can implement a single user-level system call, to MM say, that, in turn, calls the kernel and FS to retrieve needed information, construct the structure (`struct rusage`), and copy it back to the user process. Or they can implement multiple system calls, to MM, FS and the kernel, to fill the structure at the user level.

3. Observations and Conclusions

Although Minix is an outstanding platform for OS projects, students do not work with paged memory management and do no programming with threads. But, paging code tends to be quite complicated (see Linux's for example), and the combination of Minix swapping and lectures on paging suffices. And, because the students not only use a synchronization mechanism, but also get to implement it, they get most of the benefits of a threads project.

The biggest initial hurdle that students face is the Unix environment, and doing everything via the command line. Using C is not an issue, because our students come into the class knowing C++. I do, however, stress the differences between parameter passing and memory allocation in C and C++. Writing the first few system calls challenges the students, but in later projects, it becomes almost second nature to them.

Portability is a big plus for the students. Both DOSMINIX and Bochs, each with a 40MB Minix file system, fit easily on a 100MB zip[®] disk; students can work on their projects in our on-campus labs and then take them home to work with on their own PCs.

I allow the teams one or two weeks for each project. I used to give them more time on fewer projects, but they wouldn't start working until the week the projects were due, anyway. If time becomes a problem, the last project can be split among two or three teams. As is usual, the better teams finish all parts of all of the projects. For weaker teams, I sometimes supply detailed hints or even partial solutions, but at a reduction in their scores. Also, as is usual, the students complain about the amount of work the assignments require, but are pleased with what they have accomplished, and about how much more they understand, after the course is over.

My web page for this class can be found at <http://cs.selu.edu/~jhowatt/431>. Steven Hartley also has Minix assignments that can be found at <http://king.mcs.drexel.edu/~shartley/MCS370>.

References.

- [1] Beck, M., Bohme, H., Dziadzka, M., Kunitz, U., Magnus, R., and Verworner, D. *Linux Kernel Internals, 2nd Edition*. Addison Wesley Longman, Essex, England, 1998.
- [2] <http://bochs.sourceforge.net>
- [3] Bot, K.J. How to Get and Install Minix 2.0.3. <http://www.cs.vu.nl/pub/minix/2.0.3/>
- [4] Bynum, B. and Camp, T. After You Alfonse: A Mutual Exclusion Toolkit. Proc., 27th SIGCSE Technical Symposium on Computer Science Education, February, 1996.
- [5] Christopher, W.A., Procter, S.J. and Anderson, T.E. The Nachos Instructional Operating System. <http://http.cs.berkeley.edu/~tea/nachos/nachos.ps>.
- [6] Gray, J.S. *Interprocess Communications in UNIX: The Nooks and Crannies, 2nd Ed.* Prentice-Hall, Englewood Cliffs, 1998.
- [7] Kifer, M. and Smolka, S. *OSP: An Environment for Operating Systems Projects*. Addison-Wesley, Reading, MA, 1991.
- [8] Nutt, G. *Kernel Projects for Linux*. Addison Wesley Longman, Boston, MA, 2001.
- [9] Saeed, M., Schoenberger, A. and Howatt, J. Building an Operating Systems Laboratory Using an NSF ILI Grant: Our Experience. *Proc. 1994 Small College Computing Symposium*, Winona, MN, pp. 106-111.
- [10] Tanenbaum, A.S. *Operating Systems – Design and Implementation*. Prentice-Hall, Englewood Cliffs, NJ. 1987.
- [11] <http://www.vmware.com>