DUNIX: DISTRIBUTED OPERATING SYSTEMS EDUCATION VIA EXPERIMENTATION

O. Frieder
Bellcore
445 South St.
Morristown, NJ 07960
ophir@bellcore.com

A. Litman
Bellcore
445 South St.
Morristown, NJ 07960
ami@bellcore.com

M. E. Segal
EECS. Dept.
University of Michigan
Ann Arbor, MI 48109
ms@citi.umich.edu

Numerous distributed operating systems have been proposed in the literature, some of which have developed into commercial systems. Unfortunately, the educational arena has not kept up-to-date with these developments. Currently, little educational emphasis has focussed on experimental distributed operating systems. Furthermore, only a few systems have been developed as a tool to *teach* distributed systems, and those that were, are basically *skeleton systems*, and not complete. DUNIX is a fully operational, complete, distributed operating system. The DUNIX kernel is intentionally small, modular, and simple, thus is easily understood and modified. A powerful interactive kernel debugger available in DUNIX, enables students to easily observe and modify the system. Thus, *experimentation* with an actual system is possible. A sample session that illustrates the salient features of DUNIX, in terms of laboratory experimentation is presented.

1. INTRODUCTION

The study of operating systems has been part of the core computer science curriculum for many years. A typical undergraduate operating systems course consists of classroom lectures on operating systems principles as well as laboratory assignments to reinforce the lecture material. An operating systems course can also be taught by using a real operating system to illustrate important concepts and to provide a basis for laboratory assignments. Not only does the real operating system allow the student to see how individual concepts are combined to form a cohesive unit, but this approach also gives the student a hands-on learning tool. Unfortunately, most production-quality operating systems are far too large and complex to be understood by a student in a one-term course. To address this problem, two different approaches were taken. In the first approach [13], a book describing a complete operating system (UNIX™ Version 6) was written. Thus, the students could better understand the operating system source code. However, as no kernel experimentation tools were developed, debugging the code developed by the student was very difficult. The second approach lead to the development of a number of smaller operating systems, whose main purpose is to illustrate operating systems principles to students [3, 5, 20]. The primary disadvantage of these "toy" operating systems is that many issues that are encountered in the design of real operating systems are often ignored.

Teaching a contemporary operating systems course is even more challenging as the field continues to evolve. In particular, as the uniprocessor timesharing systems of yesterday are replaced with today's distributed operating systems, finding a sample distributed operating system to use in conjunction with a course may present certain problems. As with their non-distributed counterparts, a distributed operating system used in a course must avoid overwhelming the student with details, but should have sufficient functionality to illustrate important operating systems concepts and their implementation.

In this paper, we describe how an existing distributed operating system, DUNIX, can be used as a hands-on teaching tool for operating systems courses. DUNIX contains most of the functionality found in typical distributed operating systems, but is organized in a highly modular fashion to hide implementation details and make the source code more understandable. DUNIX also has many features which can be exploited to aid in teaching an operating systems course with a significant laboratory component.

The remainder of this paper is organized as follows. In Section 2, an overview of the DUNIX environment and the DUNIX kernel structure is presented. The structure of a DUNIX-based distributed operating system laboratory is described in Section 3. Section 4 contains a sample DUNIX session that illustrates how DUNIX might be used in a classroom setting. Concluding remarks are presented in Section 5.

TM UNIX is a trademark of AT&T Bell Laboratories

2. THE DUNIX OPERATING SYSTEM

The DUNIX operating system [11, 12], a UNIX derivative, is a fully operational, complete distributed operating system in production use at Bellcore. DUNIX provides users with the illusion that only a single machine exists, when in reality numerous machines comprise the system. DUNIX provides the features of a "standard" non-distributed operating system while masking the computer boundaries.

A typical DUNIX environment is shown in Figure 1. As shown, multiple machines are interconnected via a packet switched network. Each machine is fully configured and consist of its own local peripherals. To provide high availability, disks can be shared[1] between the individual machines.

The DUNIX system is by no means unique. Many other distributed operating systems have been developed, e.g., Amoeba [14, 15], the V System [4], DEMOS [2], Crystal [7], Aegis [9, 10], the CHORUS system [1], the Cambridge Distributed Computing System [16], Accent [17], Mach [18], Clouds [6], and LOCUS [21]. All of these are/were used daily in research environments; in some cases, they are

[1] Although the disks are dual-ported, a given disk is physically connected to a single computer at a given time. If that computer breaks, the disk may be switched to another machine by the system operator.

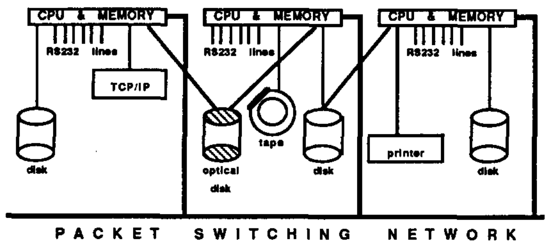


Figure 1. The DUNIX Hardware Base

also available as commercial products. For an excellent survey of distributed operating systems, the reader is referred to [19]. Although these systems are widely publicized, there have been no published accounts of using these systems as educational tools.

2.1 Global Structure of the DUNIX System

In DUNIX, each computer has its own copy of the kernel. All local kernels cooperate to create the illusion of a single UNIX machine. Each computer is fully autonomous and every kernel is equivalent and operates in the same manner. Note many distributed systems, like those employing the client/server model, do not follow this principle. Furthermore, in DUNIX a set of cooperating kernels resemble a single independent kernel in that no distinction is made regarding the locality of an operation. Supporting such a structure simplifies the debugging of the system since debugging a distributed systems requires only the debugging an individual kernel on a single machine.

The DUNIX kernel is small, modular, and relatively low in complexity as compared to other complete operating systems such as Berkeley 4.x UNIX. The kernel is divided into modules that only interact with other modules through their calling interface. This allows a programmer (or a student) to concentrate on one aspect of the system without worrying about side effects between modules or implementation details of modules not being studied.

The structure of the kernel follows Dijkstra's concept of software levels [8]. The system is composed of levels of abstractions where any level can only depend on lower levels. Figure 2 shows this structure. The kernel is composed of three main components: *the lower kernel, the upper kernel,* and *the switch.* Each of these components is *active* only when a process is running within that component. The same process may run in any component.

All objects (files, devices, etc.) reside at exactly a single computer. There is no remote caching of objects' states, and read ahead and write behind of files are confined within the computer having the file. The system is procedure call oriented. When a process wishes to perform an operation on an object, it does not send a message to a server, but instead expands to the computer where the object is located, and performs the operation itself.

The *lower kernel* maintains local objects. It provides abstract operations on these objects, and is responsible for the integrity of the objects. The *upper kernel* maintains the context of the processes, i.e., the user ID, the file creation mask, the binding of open file-descriptors to lower level names, etc.

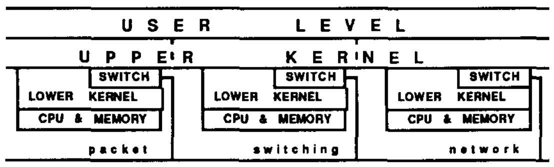


Figure 2. The structure of the DUNIX system

The upper and lower kernels are not concerned with networking and communications protocol issues; only *the switch* is cognizant of the peculiarities of the network. The switch transfers the activity of a process from the upper kernel of one computer to the lower kernel of another computer (which could be the same computer).

The upper and lower kernels do not distinguish a remote operation from a local one, while the switch does not know the semantics of the activity it is transferring, e.g., it does not distinguish between killing a process and creating a file. Most services are provided to the upper kernel via the switch, except for CPU cycles and address-space management. These services are directly provided by the the local lower kernel.

2.2 Naming of System-wide Objects

A system-wide object is an object residing in one computer and of potential interest to processes in other computers. In DUNIX, system-wide objects have *universal names.* While a process is in the upper kernel, it may hold or store only the universal names of the object(s) of interest. It may access the objects only via these names. Universal names are not reused. A universal name has the following attributes:

- Fixed size bit-string, the size depending on the type of the object.
- Location independence (utilized in process migration).
- Context independence, namely, if two processes each have a universal name and these names are bitwise equal, then both names refer to the same object. This attribute is important when forking one process to two.

The DUNIX universal device names are equivalent to standard UNIX device numbers. These names are 16-bits wide and encode, among other details, the ID of the computer having the device. Naming of files is more elaborate, since unlike devices, their lifespan is relatively brief. Universal file names refer only to active files, i.e., files whose representations reside in the primary memory file table. The 64-bit name contains the following details:

- Unique identifier (the same unique identifier is recorded in the file representation)
- ID of the computer having the file
- Index into the primary memory file table, and a unique identifier.

2.3 Software Statistics

The size of a DUNIX kernel depends on the assortment of I/O devices the kernel drives. Consider a modest DUNIX kernel with a minimal set of device drivers. Such a kernel has device drivers for disks and RS232 lines, but does not support TCP/IP. This kernel would contain approximately 17,400[2] lines of source code. Include files which are inserted within several other source files by the compiler (xxxx.h files) are counted only once. Comment and blank lines compose roughly 30% of the total count. The 17,400 lines of the are partitioned as follows:

- An Ethernet-based switch (1,800 lines)
- The *upper kernel* (2,800 lines)
- The *lower kernel* (12,800 lines)

Because the lower kernel is responsible for accessing the various devices, the disparity in size between the upper and lower kernel becomes even larger in a richly-configured system containing a large assortment of I/O devices.

3. A DISTRIBUTED OPERATING SYSTEM EDUCATION LABORATORY

3.1. Why Use DUNIX?

Two earlier operating systems have been used as teaching tools, namely, MINIX [20] and XINU [5]. Both systems have an advantage over DUNIX in that they are accompanied by a textbook and execute on a wide variety of machines. Since there is currently no text accompanying, DUNIX can only be used as a tool in an advanced distributed operating systems *laboratory* course.

For laboratory courses, DUNIX has numerous advantages over these two systems. The first is that DUNIX is truly a distributed system. Second, unlike XINU, it is a production-quality operating system that contains copy-on-write memory management, process migration, device drivers, and most notably, a powerful debugger which enables the user to view the behavior of the internals of the kernel. A DUNIX system also has extensive user-level software including software development tools, TCP/IP, a mail system, a text formatter, print spooler, and the X Windows System™.

Third, DUNIX supports system partitioning, i.e., the physical machines on the same network can be configured into independent disjoint systems. Due to the single machine illusion and the symmetry in the mode of operation, each disjoint system, even if consisting of only a single machine, resembles the global system. Thus, the debugging of the system is simplified.

Fourth, in DUNIX a subset of the computers are used for software development and other non-experimental programming tasks. This subset of computers is designated as the *production system.* The remaining computers are designated *crash computers.* A crash computer is used to test new kernels or other utilities. Because no production work is done on a crash computer, normal users are not affected during testing.

By connecting the RS232 console ports of each crash computer to the production system, a crash computer may be halted and restarted, new kernels may be downloaded into it, etc., all without any physical access to either machine. Thus, a crash computer may be manipulated from a user logged into the production system in the same manner that it could if a user was physically located at the computer's console. The console line is also used by the debugger to communicate with the user.

Finally, a DUNIX system can be configured such that the critical segments of the filesystem are write protected via hardware and/or software. As a minimal file system is relatively large (~20 Mbytes), reloading it whenever a computer crashes is a lengthy and bothersome process. Since any experimentation requires a controlled environment, it is vital that for every newly configured kernel, the file system will always start out in the same state. Since experimental kernels crash very often and can easily corrupt the filesystem, supporting read-only filesystems is essential.

3.2. Laboratory Organization

In an ideal systems laboratory, a DUNIX system would be configured in the production system/crash computers relationship described above. Students are provided with a set of "bare" slave machines which are used as crash computers. With these machines, students can experiment with all aspects of operating systems including CPU scheduling, executing privileged instructions and writing device drivers that directly access devices connected to them.

For example, by modifying the CPU and disk scheduling routines, a student can alter the performance of the system, possibly introducing new scheduling priorities. New process migration policies can be studied. Varying the swap space, cache strategies, and file system organization are all experiments that can be performed. The appropriate subset of experiments are left to the discretion of the instructor.

By giving every student in the class an account on the production (master) DUNIX system and each team of (2–3) students a time allotment on a subset of the crash computers, a team can build kernels on the production system and test them on a crash system. As shown in Figure 3, students situated at any terminal can develop their software on the production system (which would normally be more a powerful system than the crash computers) and then download the software onto a crash computer. In the figure, the six physical computers are partitioned into 4 logical systems (systems 0, 1, 2, 3): 2 computers used as a single production system (sysid=0), 2 computers configured as a single crash system (sysid=1), and two additional crash computers (sysid=2, sysid=3). Reconfiguring the logical systems can be done by a suitably privileged user at a terminal; no hardware reconfiguration is necessary.

4. EXPERIMENTATION WITH DUNIX

The DUNIX kernel debugger both traces specified event as well as provides an interactive environment to analyze a running system. In the interactive mode, the kernel is halted and the state of kernel can be examined. In the trace mode, the system continues to execute, however, events selected by the user are traced. Thus, a user can witness the various activities associated with a particular system call. Consequently, the DUNIX kernel debugger can be used not only as an aid to kernel debugging, but also as a visualization tool to illustrate the behavior of a kernel. By observing the system trace and the performance counters, a user can readily appreciate how changes in algorithms or kernel parameters affects system behavior.

[2]In comparison, an equivalent Berkeley 4.2 UNIX kernel consists of approximately 45,000 lines.
™ X Windows is a trademark of MIT.

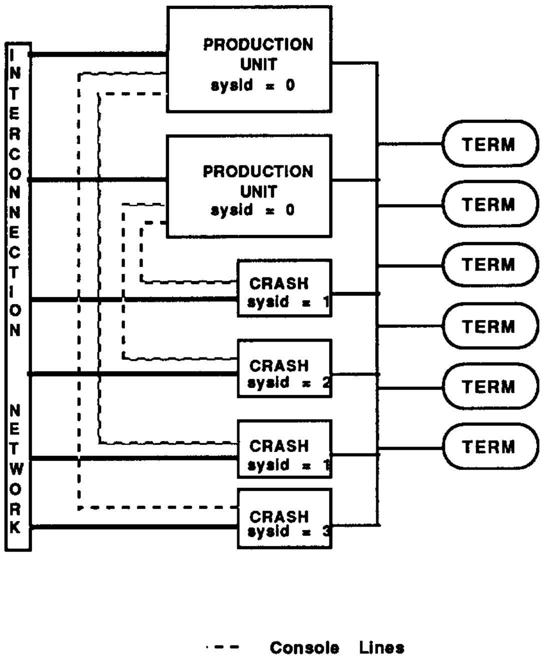


Figure 3. Laboratory Configuration

As some bugs are time dependent, trace output may hide some of these errors. However, without the tracing, detecting and correcting the errors is difficult. To remedy this problem, the DUNIX debugger can log the output in an internal circular buffer for display at a later time. Furthermore, as writing to a user terminal dramatically increases the processing time, infrequent errors are not easily reproduced without suppressed printing.

To illustrate the potential of exploiting DUNIX as an educational tool, a sample session using the DUNIX debugger is provided. In general, all user commands are highlighted in bold-face type. Control A (^A) transfers control from the user level to the debugger. In the interactive mode, the kernel activity is suspended, hence the its state is easily examined. Issuing a quit command (**q**), restarts the execution of the kernel. The "**dunix#** " and "**!** " are the user level and debugger prompts, respectively.

A session begins by compiling a new kernel on the production system (**mirage**) via the **make** command. Once compiled, the new kernel is sent over the console line to the crash computer. As seen below, downloading the newly compiled kernel from the production system to the crash computer(s) is achieved via typing a user command (**dl**) and does not require physical access to either system. As other researchers [7] have observed, requiring physical access to the crash computer to download a new kernel is a considerable deterrent to the use of the system. In this example, the kernel symbol table is also downloaded. This enables the debugger to install breakpoints and to produce a symbolic printout of the calling stack.

Once the kernel is loaded, the production system generates a virtual connection between the user terminal and the console line of the crash computer. Starting from the "VAX750" output, all remaining output presented is generated by the crash system, and user input is processed directly by the crash computer.

```
mirage> make
cc -g -c  cache.c
ld -o dunix -T 80000000  assembly.o procass.o switchas.o cache.o
dev.o clock.o console.o devmem.o exec.o file.o fsvolume.o fmark.o
fname.o pname.o printf.o proc.o proch.o psig.o hotel.o symdef.o
sysync.o syscall.o text.o tty.o  deuna.o hp.o mba.o dh.o uba.o
conf.o exos.o ptty.o null.o ts.o ci.o tu.o msa.o dd.o ld1200.o
nmi.o bi.o migrate.o  Dadhoc.o Dbreak.o Debger.o Dmtr.o Dkernel.o
mirage> dl 3 -S
Downloading dunix;  171008(text)+32768(data)=203776 bytes
                              +25388b of symbols
con: type ^B <cr> to exit.
VAX750
Symbol table:  534 functions, 23657 bytes
6291K bytes of memory (X600000)
kernel page-table at CD800 = 80AFCA00
maxkmem = 12582912, endkmem=11618816
boot-arg = FF
```

As the newly downloaded kernel comes to life, several system parameters may be set. Usually the default parameters are sufficient. In the example below, no modifications to the default values are made with the exception of the system id value (**sysid 3**). Initially the kernel probes the hardware to determine which devices are available (autoconfiguration) and than resets to user level under single user mode. The **dunix #** prompt is produced by the single user shell.

```
READY...! ?var
var - print the kernel-debugger variables
! var
here= 0 - This machine ID
rdev= 0102 - root device
rbase= 128 - base of root device
srdev= 0105 - super-root device
memdev= 0203 - swapping device
maindev= 0224 - device having the main-block (don't change minor#)
sysid= 0 - systems with different ID will not communicate
! sysid 3
! q
UBA0 at F30000
MBA0 at F28000
DH in UBA0 at 0760420 + DM at 0771100
PTTY: 26 ttys
DEUNA(DELUA) in UBA0 at 0774510
FUJI160M on dk00, via MBA0/0 read-only
FUJI160M on dk01, via MBA0/1
dk01b is swap-area; 20443K bytes
dk30a mounted on /
dk30d mounted on ...
1082K(kernel)+5209K(usr) = 6291K(total mem)
Machine #3
dunix#
```

As faulty kernels are likely to corrupt the file system, DUNIX may be used with a write-protected file system. This protection is provided either via software and/or hardware. In the example below, an unsuccessful attempt is made to write onto a read only disk partition. To provide a writeable partition, a clean, temporary disk partition is created (**mktmp**).

```
dunix# cp /etc/passwd new-file
cp: cannot create new-file - Read-only file system
dunix# df

                                        FREE  TOTAL  %FREE
/dev/dk30a on / (R.O.)                  1702K 19456K    8%
dunix# mktmp
mkfs /dev/gdk1e 7000
total - 7000 blocks
inodes - 280 blocks
free-list step = 3
mount -p /dev/gdk1e /tmp
endif
dunix# df
                                        FREE  TOTAL  %FREE
/dev/dk30a on / (R.O.)                  1702K 19456K    8%
/dev/dk31e on /tmp                      6877K  7168K   95%
dunix# cd /tmp
```

Providing an experimental environment requires sufficient on-line help, otherwise experimentation becomes more of a library search and not a "playful experience". DUNIX provides such support via the **?** command. Simply typing a question mark in the debugger mode prints all the debugger options. A question mark followed by an option name provides a detailed description of the option. **Commands** are executed once. **Variables** are integer values and can readily be examined and modified. A **flag** is activated by typing its name, either with or without a + sign and is cancelled by typing **- flag name**. Flags are usually used to invoke traces at specific events. For example, the **csw** flag traces context switches.

```
dunix# ^A!
! ?
FLAGS: sc, fssc, fsc, ex, sig, exec, bexit, swp, csw, devinit, ST, FILTER,
badname, macheck, yellall, pflt, scall, inmsg, ioctal, cln, trw, ubaspace,
excom, hptr, hperr, tstr, tserr, tstream, tutr, msatr, msamsg, msaer, lderr,
notout, loopb, nohkeeper, delua, mig, t1, t2, t3, t4, t5, t6, t7, t8,
VARIABLES: here, rdev, rbase, srdev, memdev, maindev, conspeed, sysid,
sclock, xpipe, wcpu, invcomed, ldmaxt, interr, maxdil, v1, migto, migcyc,
COMMANDS:menu,?,help,q,flg,var,pt,ct,playb,ssc,ls,a2f,b,mark,hmark,
s,mem,rs,tty,p,proc,mini,file,bi,nmi,hpst,mbast,msast,vol,pbook,port,
text,seg,pgt,ckpt,prdeu,trc,netm,mtr,cmtr,cpu,msys,mscall,d1,d2,d3,hey,
echo,necho,exst,tsst,tust
! q
```

Understanding the execution of a kernel requires the capability of tracing individual events. In the example below, all system calls are traced for the execution of the **ls** command. As shown, both the successful and failed system calls are printed with the appropriate indication of status. All system calls, except **fork**, result in two statement being printed; once at the start and once at the return of the execution of the call. The fork system call results in three statements, once at the start, once upon the return of the child fork, and once upon the return of the parent fork system call.

```
! ?exec
exec - trace successfull exec's
! +exec
dunix# echo XXX > NEW-file
dunix# ls
     -csh[2339] exec: ls
! ?sc
sc - trace system-calls
! sc
! q
      syscall exec(Xd2e0="/bin/ls",Xda70,Xde24) by ls[2339]  RETURNING 0
      syscall fork() by -csh[355]  RETURNING X923=2339
      syscall wait() by -csh[355]
      syscall time() by ls[2339]
      syscall time() by ls[2339]  RETURNING X245bc8e5=609994981
      syscall ioctl(1,X7408,X7fffcad6) by ls[2339]
      syscall ioctl(1,X7408,X7fffcad6) by ls[2339]  RETURNING 0
      syscall brk(X6f9c) by ls[2339]
      syscall brk(X6f9c) by ls[2339]  RETURNING 0
      syscall stat(X41f4=".",X7fffca90) by ls[2339]
      syscall stat(X41f4=".",X7fffca90) by ls[2339]  RETURNING 0
      syscall open(X41f4=".",0) by ls[2339]
      syscall open(X41f4=".",0) by ls[2339]  RETURNING 3
      syscall read(3,X7400,X400) by ls[2339]
      syscall read(3,X7400,X400) by ls[2339]  RETURNING X80=128
      syscall read(3,X7400,X400) by ls[2339]
      syscall read(3,X7400,X400) by ls[2339]  RETURNING 0
      syscall close(3) by ls[2339]
      syscall close(3) by ls[2339]  RETURNING 0
      syscall write(1,X55c4,9) by ls[2339]
NEW-f  syscall write(1,X55c4,9) by ls[2339]  RETURNING 9
ile
      syscall exit(0) by ls[2339]
      syscall wait() by -csh[355]  RETURNING X923=2339
      syscall write(1,X7fffc968,7) by -csh[355]
dunix#  syscall write(1,X7fffc968,7) by -csh[355]  RETURNING 7
      syscall ioctl(3,X7412,X7fffcae4) by -csh[355]
      syscall ioctl(3,X7412,X7fffcae4) by -csh[355]  RETURNING 0
      syscall ioctl(3,X7411,X7fffcaea) by -csh[355]
      syscall ioctl(3,X7411,X7fffcaea) by -csh[355]  RETURNING 0
      syscall read(3,X7fffca1c,Xc7) by -csh[355]
```

Disk I/O can be traced. Disk partition **dk30a** contains the **/etc/passwd** file, hence only READs are required. The READ and WRITE output related to disk partition **dk31e** results from the kernel initially reading the current (dot) directory prior to writing the password file in that directory. As shown, each disk request produces two output statements. The first is for the queueing and the second is for the processing of the request. Note that failed system calls are also specifically traced in this example.

```
^A ! -sc
! -csw
! ?fsc
fsc - trace failed system-calls
! fsc
! q
dunix# cp /no-file
FAILED   syscall open(X7fffcb77="/no-file",0) by cp[675]  [ENOENT]
cp: cannot open /no-file
dunix# ^A
! ?hptr
hptr - Trace HP I/O
! hptr
! q
dunix# cp /etc/passwd .
dk30a: queue read of block 643 (req@L6ac6c)
dk30a:  READ, blk=643=1542=<4,8,6>,
dk30a: queue read of block 984 (req@L6acf4)
dk30a:  READ, blk=984=2224=<6,9,16>,
dk30a: queue read of block 1314 (req@L6ad7c)
dk30a:  READ, blk=1314=2884=<9,0,4>,
dk30a: queue read of block 1335 (req@L6ae04)
dk30a:  READ, blk=1335=2926=<9,1,14>,
dk31e: queue read of block 282 (req@L6ae8c)
dk31e:  READ, blk=282=160564=<501,7,20>,
dk30a: queue read of block 4227 (req@L6af14)
dk30a:  READ, blk=4227=8710=<27,2,6>,
FAILED    syscall   stat(X1a98=".//passwd",X7fffca9c)   by   cp[739]
[ENOENT]
dk31e: queue read of block 8 (req@L6af9c)
dk31e:  READ, blk=8=160016=<500,0,16>,
dk31e: queue write of block 282 (req@L6ae8c)
dk31e:  WRITE, blk=282=160564=<501,7,20>,
dk30a: queue read of block 4802 (req@L6b024)
dk30a:  READ, blk=4802=9860=<30,8,4>,
dk30a: queue read of block 4805 (req@L6b134)
dk30a:  READ, blk=4805=9866=<30,8,10>,
^A ! -hptr
```

Breakpoints can be set at entry and exit of any procedure of the kernel. For example, the **path2fnm** procedure translates a logical name to a binary name. As shown, a breakpoint is set at the entry and exit of the **path2fnm** function. The calling stack display command (s) displays the calling sequence that resulted in an invocation of **path2fnm.** The +s 1 option lists the values of all local variables on the top of the calling stack.

```
dunix#
dunix# ^A ! ?fssc
fssc - trace file-system system-calls
! fssc
! ?b
b [function [ex]] - Install (entry & exit) break-point at 'function'
! b  path2fnm
! q
dunix# ls /etc/passwd
   syscall exec(Xd2e0="/bin/ls",Xe364,Xe4d4) by -csh[803]
  B.P. path2fnm(53984=Xd2e0, 0, 2147472324=X7fffd3c4)
                                                [in file fsymbol.]
! q
  B.P. path2fnm(53984=Xd2e0, 0, 2147472324=X7fffd3c4), returning 1
                                                [in file fsymbol.]
! ?s
s [count] - Display the 'count' upper stack-frames
! s
  B.P. path2fnm(53984=Xd2e0, 0, 2147472324=X7fffd3c4), returning 1
                                                [in file fsymbol.]
exece?(53984=Xd2e0, 58772=Xe594, 58580=Xe4d4)  [in file exec.]
syscall()  [in file syscall.]
! +s 1
  B.P. path2fnm(53984=Xd2e0, 0, 2147472324=X7fffd3c4), returning 1
                                                [in file fsymbol.]
    fnp=2147472324=X7fffd3c4  path(p)=53984=Xd2e0  mode(p)=0
    fnp(p)=2147472324=X7fffd3c4  rv=-1  dnp=2147472212=X7fffd354
    cnk=1852400175=X6e69622f  copy=L17760  nc=7  rnc=-1  enc=7
    myerr=0  trbl=4  trbl2=2147472324=X7fffd3c4  hookno=L3194
   f=L2d0f4  t=L4c55c  sr=2
! q
   syscall exec(Xd2e0="/bin/ls",Xe364,Xe4d4) by ls[803]  RETURNING 0
   syscall stat(X7fffcb77="/etc/passwd",X7fffcaa0) by ls[803]
  B.P. path2fnm(2147470199=X7fffcb77, 0, 2147472444=X7fffd43c)
                                                [in file fsymbol.]
! q

  B.P. path2fnm(2147470199=X7fffcb77, 0, 2147472444=X7fffd43c),
returning 1
                                                [in file fsymbol.]
! q
   syscall  stat(X7fffcb77="/etc/passwd",X7fffcaa0)  by  ls[803]
RETURNING 0
/etc/passwd
dunix# ^A! -b *
! -fssc
```

The handling of scarce resources is of primary concern in operating systems. A common error is not reclaiming unused resources or to keep them locked unnecessarily. The debugger provides a mechanism for checking the state of the system resources. As shown below, prior to executing the **echo xxxxx > myfile** command, only 6 active files are used in the system. While executing the echo command, 7 active files are found.

```
dunix# ^A! ?ssc
ssc [ /+/-]ssc [syscall] - trace/trace&stop/untrace the given syscall
! +ssc  creat
! q
dunix# echo xxxxx > myfile
    syscall creat(Xd9f8="myfile",X1b6) by -csh[1027]
! rs
RESOURCES:
USER-MEMORY: 9801[free] + 138[cache] + 193[used] = 10132[total] * 512
bytes
SECODARY-MEM: 644[free] + 0[used] = 644[total] * 31744 bytes
PROC:
     288[free]+1[run]+9[pause]+2[wait]+0[fetus]+0[bizarre]=300[total]
           =12[loaded]+0[out]+288[free];
FILES:      294[free]+6[used]+0[locked]+0[bizarre]=300[total]
FMARKS:     298[free]+2[used]+0[wait] = 300[total]
BUF:        297[free]+0[I/O]+0[lock]+3[used]=300[total]
TEXTS:      138[free]+2[used]+10[cached]+0[bizarre] = 150[total]
TCP:        Ports: 0[used]+50[free]=50[total];
! q
     syscall creat(Xd9f8="myfile",X1b6) by -csh[1027]  RETURNING 5
! rs
RESOURCES:
USER-MEMORY: 9801[free] + 138[cache] + 193[used] = 10132[total] * 512
bytes
SECODARY-MEM: 644[free] + 0[used] = 644[total] * 31744 bytes
PROC:
     288[free]+0[run]+9[pause]+3[wait]+0[fetus]+0[bizarre]=300[total]
           =12[loaded]+0[out]+288[free];
FILES:      293[free]+7[used]+0[locked]+0[bizarre]=300[total]
FMARKS:     297[free]+3[used]+0[wait] = 300[total]
BUF:        297[free]+0[I/O]+0[lock]+3[used]=300[total]
TEXTS:      138[free]+2[used]+10[cached]+0[bizarre] = 150[total]
TCP:        Ports: 0[used]+50[free]=50[total];
! q
dunix# rm  myfile
dunix#
```

If the user is interested in additional details concerning only file usage, that is also possible. For example, by stepping through a file write while examining the system file table, the creation of a new file entry (**Slt#61**) is observed.

```
! +ssc  creat
! q
dunix# echo xxxxx > myfile
    syscall creat(Xe5b8="myfile",X1b6) by -csh[867]
! ?file
file [a] - Print the file-table
! file
File-table:
Slt#0:  /;  dev=014102,  mode=040755(FTDIR),  count=3,  lcount=1,
flg=FACTIVE
Slt#1: ...; dev=014105, mode=040777(FTDIR), count=1, lcount=1, flg=
Slt#5: null; dev=014102, mode=020666(FTCHR), count=1, lcount=1, flg=
Slt#6: console3; dev=014102, mode=020000(FTCHR), count=1, lcount=1,
flg=
Slt#37: tmp; dev=014102, mode=040755(FTDIR), count=1, lcount=1, flg=
Slt#55:  tmp;  dev=014206,  mode=040777(FTDIR),  count=3,  lcount=1,
flg=FACTIVE
! q
    syscall creat(Xe5b8="myfile",X1b6) by -csh[867]  RETURNING 5
! file
File-table:
Slt#0:  /;  dev=014102,  mode=040755(FTDIR),  count=3,  lcount=1,
flg=FACTIVE
Slt#1: ...; dev=014105, mode=040777(FTDIR), count=1, lcount=1, flg=
Slt#5: null; dev=014102, mode=020666(FTCHR), count=1, lcount=1, flg=
Slt#6: console3; dev=014102, mode=020000(FTCHR), count=1, lcount=1,
flg=
Slt#37: tmp; dev=014102, mode=040755(FTDIR), count=1, lcount=1, flg=
Slt#55: tmp; dev=014206, mode=040777(FTDIR), count=3, lcount=1,
    flg=FTMOD+FACTIVE
Slt#61: myfile; dev=014206, mode=0100644(FTREG), count=1, lcount=1,
    flg=FTMOD+FRMOD+FACTIVE
! q
```

Similarly, examining the process table during an **exec** system call execution illustrates the transformation of the **csh[899]** execution to that of the **ps[899]** execution. The trailing number is the process id.

```
dunix# ^A! +ssc exec
! q
p s
   syscall exec(Xd2e0="/bin/ps",Xe67c,Xe4d4) by -csh[899]
! ?proc
proc [index/pointer [count]] - display process
! proc
Processes:
hkeeper[0], ppid=0, tty0, slt#0[L4b878], SWAIT <DEMON>
mkeeper[1], ppid=0, tty0, slt#1[L4b9a4], waiting for work <DEMON>
taxi[2], ppid=0, tty0, slt#2[L4bad0], waiting for WORK <DEMON>
taxi[3], ppid=0, tty0, slt#3[L4bbfc], waiting for WORK <DEMON>
taxi[4], ppid=0, tty0, slt#4[L4bd28], waiting for WORK <DEMON>
taxi[5], ppid=0, tty0, slt#5[L4be54], waiting for WORK <DEMON>
hitman2[6], ppid=0, tty0, slt#6[L4bf80], waiting for work <DEMON>
hitman1[7], ppid=0, tty0, slt#7[L4c0ac], waiting for work <DEMON>
herald[8], ppid=0, tty0, slt#8[L4c1d8], waiting for work <DEMON>
init[323], ppid=0, tty0214300, slt#9[L4c304], SWAIT, doing wait,
-csh[355], ppid=323, tty0214300, slt#10[L4c430], URDY, doing fork,
-csh[899], ppid=355, tty0214300, slt#11[L4c55c], RUN, doing exec,
! q
   syscall exec(Xd2e0="/bin/ps",Xe67c,Xe4d4) by ps[899] RETURNING 0
! proc
Processes:
hkeeper[0], ppid=0, tty0, slt#0[L4b878], SWAIT <DEMON>
mkeeper[1], ppid=0, tty0, slt#1[L4b9a4], waiting for work <DEMON>
taxi[2], ppid=0, tty0, slt#2[L4bad0], waiting for WORK <DEMON>
taxi[3], ppid=0, tty0, slt#3[L4bbfc], waiting for WORK <DEMON>
taxi[4], ppid=0, tty0, slt#4[L4bd28], waiting for WORK <DEMON>
taxi[5], ppid=0, tty0, slt#5[L4be54], waiting for WORK <DEMON>
hitman2[6], ppid=0, tty0, slt#6[L4bf80], waiting for work <DEMON>
hitman1[7], ppid=0, tty0, slt#7[L4c0ac], waiting for work <DEMON>
herald[8], ppid=0, tty0, slt#8[L4c1d8], waiting for work <DEMON>
init[323], ppid=0, tty0214300, slt#9[L4c304], SWAIT, doing wait,
-csh[355], ppid=323, tty0214300, slt#10[L4c430], SWAIT, doing wait,
ps[899], ppid=355, tty0214300, slt#11[L4c55c], RUN, doing exec,
! q
 PID TT STAT  TIME COMMAND
 323 c3 I    0.14 init 3 ff
 899 c3 R    0.14 ps
dunix#
```

Modifying performance related algorithms, e.g. CPU and disk scheduling, necessitates the capability of viewing the performance of the system. As shown below, such capability exists in DUNIX. The **dbg** command invokes the debugger from the user level. Thus the **dbg cmtr; cp /etc/hosts .; dbg mtr** command results in the clearing of the system meters and the displaying of the performance involved in copying /etc/hosts to the dot directory.

```
dunix# ! ?cmtr
cmtr - Clear all meters
! ?mtr
mtr Display meters
! q
dunix# dbg cmtr; cp /etc/hosts .; dbg mtr
# 8.5 sec.
# CPU: 42%idle(3.6s) + 0%usr(0.0s) + 57%sys(4.8s) = 100%total(8.5s)
#    63%syslo(3.0s) + 11%syshi(0.5s) + 26%sysint(1.2s) = 100%sys(4.8s)
#  830 syscalls (97/s), 524 con-sw (62/s), 0 swapin (1/8s)
# user i/o: 411Kby read (48K/s), 411Kby write (48K/s), 10240by exec
(1K/s)
# cache:  1889 calls = 950 rd + 532 rdwr + 407 wr; maxdirty=77,
minfree=218
# -cache: 755 i/o = 412 rd + 3 r.f.w. + 340 wr; 75 dirty; 398 rd-ahead(398
used)
# MBA: 758 interrupts (95/sec)
# HP0:   410 trns(48/s), 22 pos(3/s), 419K bytes(49K/s), 0 retries
#    88%busy(7s) + 12%idle = 57%flow+4%pos+28%wait+12%idle = 100%
# HP1:   345 trns(40/s), 9 pos(1/s), 353K bytes(41K/s), 0 retries
#    69%busy(5s) + 31%idle = 40%flow+2%pos+27%wait+31%idle = 100%
dunix#
```

None of the above examples have dealt with the distributed aspects of DUNIX. In the accompanying screen display, a user examines the implementation of the change file mode **(chmod)** system call. The top right hand window displays the source code. The source code consists of two routines: **chmod** and **chmodl**. The chmod system call initiate with first routine which executes on the local computer. The second routine, invoked by the **chmod** routine, executes on the machine containing the desired file. In the example presented, two files are modified. The larger window on the left-hand-side illustrates that all activity is local for the local file. Also shown in the larger window is the local activity needed for the remote file. The remote activity is performed on the remote machine (machine 2), shown in the lower, smaller window on the right-hand-side. Note that the actual code is identical regardless of where the file is situated: local or remote. The three "CPU load" windows are monitoring the production system. The two crash computers (machines 2 and 3) are totally independent from the remaining three machines, thus are not represented in the performance meters.

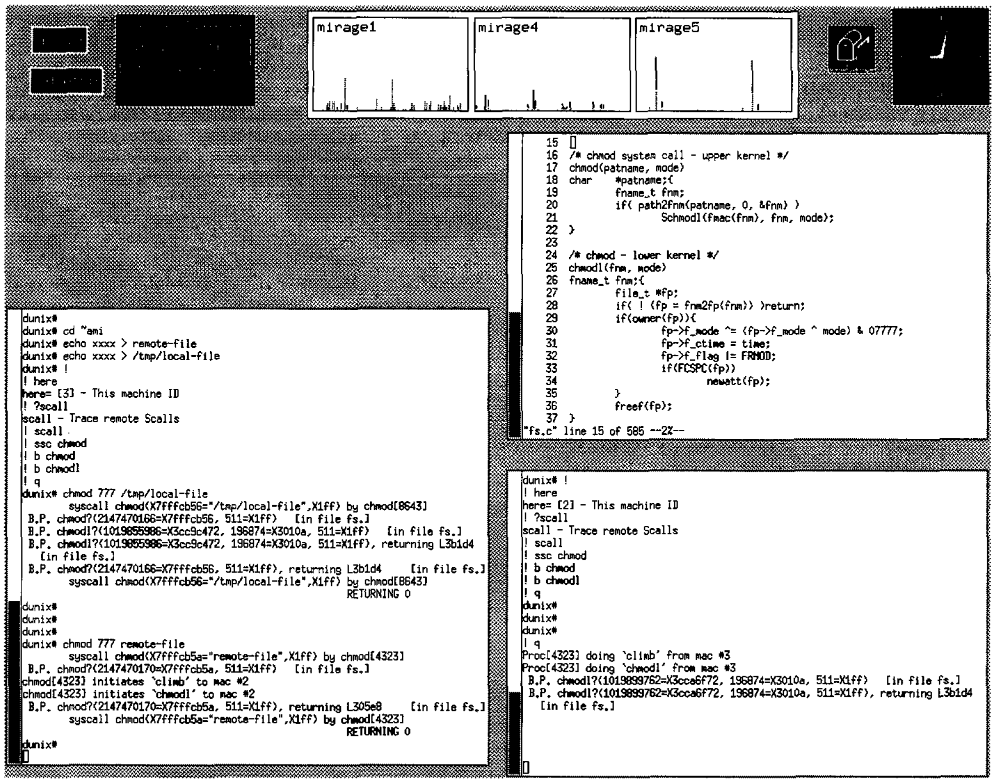
The sample session ends by sending a thank you message to a colleague in Syracuse University, Syracuse New York while tracing only the **exec** system calls. As seen the **smtp** command is executed, the message is sent and the connection is terminated.

```
dunix# ! ?exec
exec - trace successfull exec's
! exec
! q
dunix#   mail   Frieder@top.cis.syr.edu
         -csh[1251] exec: mail Frieder@top.cis.syr.edu
Subject: Welcome to Euromicro
Gideon,
         You are now part of the mail example in Euromicro.
Ami, Ophir, and Mark.
^DEOT
dunix#   send[1347] exec: sh -c /usr/lib/smtp/smtpqer 'to
         sh[1379] exec: csh -f /usr/lib/smtp/smtpqer to
         csh[1411] exec: valueof SMTP_QUEUE
         csh[1443] exec: mktemp Tmp1379
         csh[1475] exec: cat
         csh[1507] exec: ln Tmp1379a 1002
         csh[1539] exec: rm Tmp1379a
         csh[1571] exec: cp /dev/null 1002.err
         csh[1667] exec: smtp top.cis.syr.edu root Fried
         csh[1731] exec: cat 1002.log
         csh[1763] exec: rm 1002.log
         csh[1795] exec: rm -f 1002.err 1002.c
         csh[1827] exec: rm -f 1002
```

5. SUMMARY

The study of operating systems is a fundamental component of current undergraduate computer science syllabi. Traditionally, operating systems education consists of reading textbooks and research papers, coding relatively small, model operating system, possibly only some of individual components of the system, and participating in various classroom discussions. We believe that in addition to the traditional teach approach, experimentation with an actual system is necessary. Specifically, traditional teaching approaches introduce the student to the fundamentals of operating systems, but to obtain deep insight to the intricacies of systems, experimentation is required.

This paper focussed on the exploitation of DUNIX, a complete, modular, distributed operating system, as a teaching tool for experimental operating systems laboratory.



```
/* chmod system call - upper kernel */
chmod(patname, mode)
char    *patname;{
        fname_t fnm;
        if( path2fnm(patname, 0, &fnm) )
                Schmodl(fmac(fnm), fnm, mode);
}

/* chmod - lower kernel */
chmodl(fnm, mode)
fname_t fnm;{
        file_t *fp;
        if( ! (fp = fnm2fp(fnm)) )return;
        if(owner(fp)){
                fp->f_mode ^= (fp->f_mode ^ mode) & 07777;
                fp->f_ctime = time;
                fp->f_flag |= FRMOD;
                if(FCSPC(fp))
                        newatt(fp);
        }
        freef(fp);
}
"fs.c" line 15 of 585 --2%--
```

```
dunix#
dunix# cd ~ami
dunix# echo xxxx > remote-file
dunix# echo xxxx > /tmp/local-file
dunix# !
! here
here= [3] - This machine ID
! ?scall
scall - Trace remote Scalls
! scall
! ssc chmod
! b chmod
! b chmodl
! q
dunix# chmod 777 /tmp/local-file
        syscall chmod(X7fffcb56="/tmp/local-file",X1ff) by chmod[8643]
 B.P. chmod?(2147470166=X7fffcb56, 511=X1ff)   [in file fs.]
 B.P. chmodl?(1019855986=X3cc9c472, 196874=X3010a, 511=X1ff)   [in file fs.]
 B.P. chmodl?(1019855986=X3cc9c472, 196874=X3010a, 511=X1ff), returning L3b1d4
   [in file fs.]
 B.P. chmod?(2147470166=X7fffcb56, 511=X1ff), returning L3b1d4     [in file fs.]
        syscall chmod(X7fffcb56="/tmp/local-file",X1ff) by chmod[8643]
                                                    RETURNING 0
dunix#
dunix#
dunix#
dunix# chmod 777 remote-file
        syscall chmod(X7fffcb5a="remote-file",X1ff) by chmod[4323]
 B.P. chmod?(2147470170=X7fffcb5a, 511=X1ff)   [in file fs.]
chmod[4323] initiates `climb' to mac #2
chmod[4323] initiates `chmodl' to mac #2
 B.P. chmod?(2147470170=X7fffcb5a, 511=X1ff), returning L305e8     [in file fs.]
        syscall chmod(X7fffcb5a="remote-file",X1ff) by chmod[4323]
                                                    RETURNING 0
dunix#
```

```
dunix# !
! here
here= [2] - This machine ID
! ?scall
scall - Trace remote Scalls
! scall
! ssc chmod
! b chmod
! b chmodl
! q
dunix#
dunix#
dunix#
! q
Proc[4323] doing `climb' from mac #3
Proc[4323] doing `chmodl' from mac #3
 B.P. chmodl?(1019899762=X3cca6f72, 196874=X3010a, 511=X1ff)   [in file fs.]
 B.P. chmodl?(1019899762=X3cca6f72, 196874=X3010a, 511=X1ff), returning L3b1d4
   [in file fs.]
```

DUNIX is chosen from among the numerous available distributed systems due to the numerous features that aid in education that DUNIX incorporates. Some of these features include, a powerful debugger, supporting a write-protected filesystem, kernel downloading capabilities that do not require physical access to the hardware, etc.. The advantages provided by these features were discussed. A sample session using DUNIX was presented.

If a deep insight into the inter-workings of a distributed system is desired, experimentation with an actual system is beneficial. This paper presented a possible system to use as a teaching tool that can provide the desired insight.

REFERENCES

[1] Armand, F., *et. al.*, "Towards a Distributed UNIX System: the CHORUS Approach," EUUG, Manchester, UK, September, 1986.

[2] Baskett, F., *et. al.*, "Task Communication in DEMOS," Proc. of the Sixth Symposium on Operating System Principles, November, 1977.

[3] Biersack, E. W., Personal communication concerning the use of the TURBO Operating System at the Munich University of Technology for education, 1989

[4] Cheriton, D. R., and Zwaenepoel, W., "The Distributed V Kernel and its Performance for Diskless Workstations," Proc. 9th Symposium on Operating System Principles, 1983.

[5] Comer, D., Operating System Design: The XINU Approach, Prentice-Hall, Englewood Cliffs, NJ, 1984.

[6] Dasgupta, P., *et. al.*, "The Clouds Distributed Operating System: Functional Description, Implementation Details and Related Work," IEEE Eighth Int'l Conf. on Distributed Computing Systems, San Jose, California, June, 1988.

[7] De Witt, D. J., *et. al.*, "The Crystal Multicomputer: Design and Implementation Experience," IEEE Transactions on Software Engineering, SE-13(8), August, 1987.

[8] Dijkstra, E. W., "The structures of THE multiprogramming system," CACM, Vol. 11, No. 3, pp. 341-346, March 1968.

[9] Leach, P. J., *et. al.*, "UIDs as Internal Names in a Distributed File System," Proc. Symposium on Principles of Distributed Computing, Ottawa, Canada, August, 1982.

[10] Leach, P. J., *et. al.*, "The Architecture of an Integrated Local Network," IEEE Journal on Selected Areas in Communications, SAC-1(5), November, 1983.

[11] Litman, A., "DUNIX: A Distributed UNIX system", Proceeding of the EUUG Conference, September, 1986.

[12] Litman, A., "The DUNIX Distributed Operating System", Operating Systems Review, January 1988.

[13] Lions, J., *A Commentary on the Unix Operating System*, University of New South Wales, Australia, 1977.

[14] Mullender, S. J. and Tanenbaum, A. S., "Protection and Resource Control in Distributed Operating Systems," Computer Networks, vol.8, no. 5,6, 1984.

[15] Mullender, S. J., "Principles of Distributed Operating System Design", Ph.D. Thesis, Amsterdam, The Netherlands, 1985.

[16] Needham, R. M., and Herbert, A. J., *The Cambridge Distributed Computing System*, Addison-Wesley, Reading, Mass., 1982.

[17] Rashid, R., and Robertson, G., "Accent: A Communication Oriented Network Operating System Kernel," Proc. of the Eighth Symposium on Operating System Principles, SIGOPS 15(5), 1981.

[18] Rashid, R., *et. al.*, "Machine-Independent Virtual Memory Management for Paged Uniprocessor and Multiprocessor Architectures," Proc. of the Second Int'l Conf. on ASPLOS, SIGOPS 21(4), 1987.

[19] Tanenbaum, A. S., and Van Renesse, R., "Distributed Operating Systems," *ACM Computing Surveys*, 17(4), December, 1985.

[20] Tanenbaum, A. S., *Operating Systems: Design and Implementation*, Prentice-Hall, 1987.

[21] Walker, B., *et. al.*, "The LOCUS Distributed Operating System," Proc. 9th Symposium on Operating System Principles, 1983.