Middleware Enabled Fault Management for Commercial Operating Systems

April 29, 1997

for submission to: Software Reliability and Fault Tolerance Track of the Computer Science Division at the 15th Annual International Conference of the AOM/IAOM

Author Information:

Charlotte A. Rekiere

Carnegie Mellon University
5000 Forbes Ave.
2201 Hamburgh Hall
Pittsburgh, PA 15232

PH: (412) 621-9406

Fax: (412) 268-5229

email: crekiere@cs.cmu.edu

Daniel P. Siewiorek

Professor CS/ECE
Carnegie Mellon University
5000 Forbes Ave.
1201 Hamburgh Hall
Pittsburgh, PA 15232

PH: (412)268-2570

Fax: (412) 268-5229

email: dps@cs.cmu.edu

Middleware Enabled Fault Management for Commercial Operating Systems

Charlotte A. Rekiere
Daniel P. Siewiorek
Carnegie Mellon University

Abstract: Commercial computer systems have escaped the scrutiny for fault-tolerance typically reserved for mission critical systems. As computer systems become an integral part of daily activities people are beginning to depend on and expect fault-free behavior. The implementation of a fault-management middleware layer to an existing operating system can prove to be an effective way to quickly add fault-management features to commercial computer systems.

This paper evaluates and defines a taxonomy of the implementations of four fault-management middleware layers in three commercial off-the-shelf Operating Systems: pSOS (embedded), Mach 3.0 (micro-kernel) and HP-UX (monolithic kernel).

The middleware development process for HP-UX is described and analyzed for performance and system overhead. Adding assertions shows the ease of implementing fault-management features to the HP-UX middleware. As a demonstration, assertions are used to protect an application from incorrect kernel behavior exposed in the unmodified operating system through running Robustness Benchmarks [Dingman96].

1.0 Introduction

Typically, the expectation and motivation for developing fault management techniques has focused on highly-specialized mission-critical systems. Commercial systems have escaped similar scrutiny and usually do not provide any type of fault management in the general computing environment. Computer systems have become an integral part of daily activities and people are beginning to depend on and expect fault-free behavior. One method to provide fault management policies on commercial systems is the implementation of a middleware layer with the necessary visibility and control for adding fault management techniques. A middleware layer can be described simply as software placed between two existing systems to facilitate their communication and interoperability. [Lee95].

This paper evaluates the implementation of four system call monitor middleware layers in three commercial off-the-shelf Operating Systems. Specifically, pSOS (embedded), Mach 3.0 (micro-kernel) and HP-UX (monolithic kernel). The middleware layers evaluated reside between the operating system and an application. The evaluation has produced a taxonomy of the architectural aspects that facilitate implementation of a fault management middleware layer.

The HPLR (HP Library Replace) middleware development process is described and analyzed for performance overhead. An assertion policy is evaluated on HP-UX with the execution of Robustness Benchmarks [Dingman96]. It is expected that the results of this work will provide a framework for understanding how middleware fault-management techniques can be applied within current operating system architectures. The attributes that are important to maximize the performance and capabilities of the mechanism defined can be applied to the development of future architectures.

2.0 Taxonomy

System calls provide the interface between a process and the operating system. A mechanism for monitoring system call requests can be developed when the method defined to service a system call for a particular operating system is understood. A log of system calls generated can be used to debug application or operating system behavior, analyze system performance and trend analysis. Three operating systems with monitoring mechanisms were evaluated to develop a taxonomy of operating systems features that facilitate system call monitoring: Mach 3.0 [Rashid89], pSOS [Integrated95] and HP-UX [HPa95].

The taxonomy begins with Figure 1 which defines a simple control flow model of a system call request in a generic operating system. Control switches from the application to the kernel when the system call is executed. This switch is defined by the enter path of Figure 1. After the system call is completed, control returns to the application as defined by the exit path of Figure 1. A middleware layer is added in Figure 2 that provides the system call monitor function.

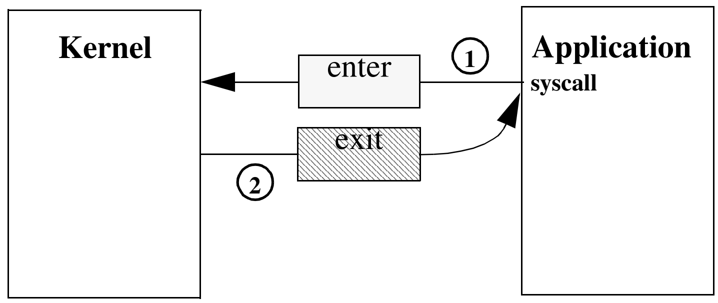

FIGURE 1. Generic Operating System Model

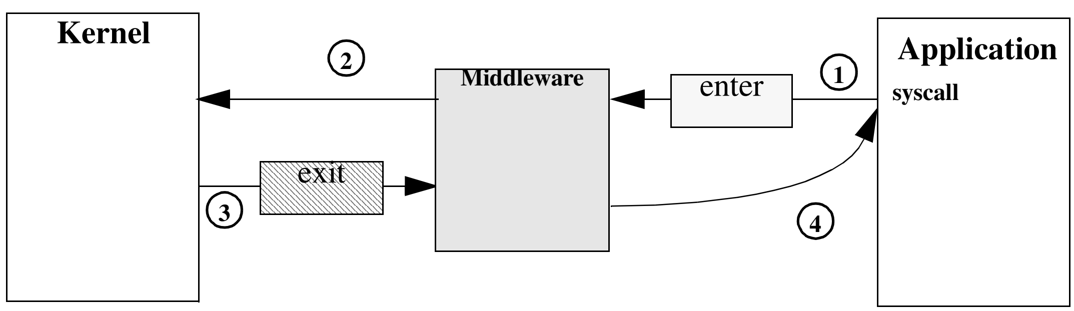

FIGURE 2. Generic Operating System model with Middleware

How the middleware layer is implemented for a specific operating system depends on the mechanism defined to switch control between an application and the kernel. The next set of figures will show three operating systems and four mechanisms to enable system call monitoring following the model established in Figures 1 and 2. A summary of the attributes of each middleware implementation is provided in Tables 1 and 2.

The control flow of the pSOS operating system is represented in Figure 3. pSOS handles system calls through a single software interrupt. When the system is booting an entry in the interrupt vector table is set to point to the location within the kernel to execute the system call requested by an application. To execute a system call, the user application executes an interrupt instruction, and the processor jumps to the location specified in the interrupt vector table [Integrated95].

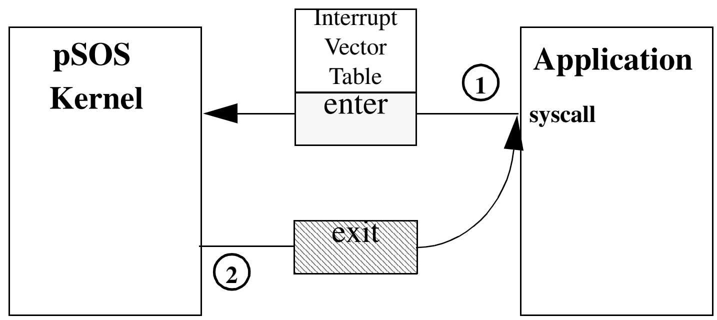

FIGURE 3. pSOS Operating System model

A middleware layer is added by mapping the entry in the interrupt vector table during the boot sequence to point to a routine in the middleware. This change forces all the system calls made by the user application to pass through the middleware prior to entering the kernel. Once control is passed to the kernel the system call is executed and the kernel returns control to the application as shown by the exit path of Figure 4.

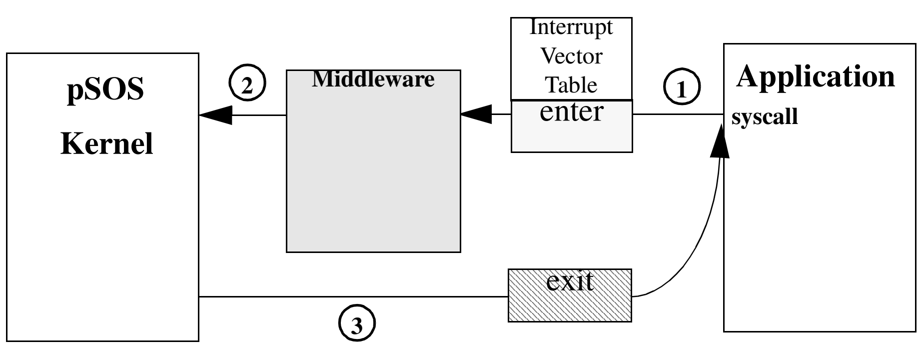

FIGURE 4. pSOS Operating System model with Middleware

Figure 5 shows the control flow of the Mach 3.0 microkernel. In microkernel operating systems, system services are divided between a user-level server, which provides much of the visible operating system interface, and a supervisor-level microkernel that implements low-level resource management. The microkernel provides enough low-level support so that various servers can be implemented to run on top of it [Russinovich94]. The server implemented for the system call monitor evaluation is UNIX 4.3 (UX) [Bach86].

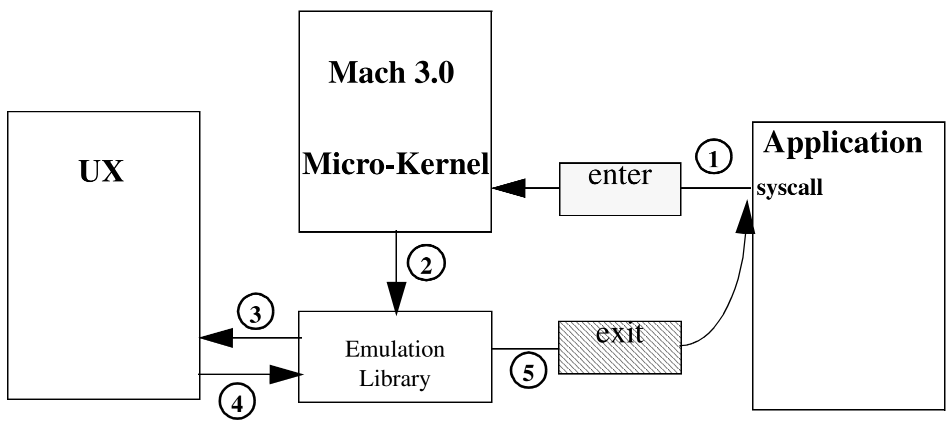

FIGURE 5. Mach Operating System Model

When a system call trap is made, Mach switches control into the emulation library. This library takes the system call parameters and number, and packages a message which is sent to UX. In UX, threads wait for incoming requests and after decoding the message, call the appropriate function. After the request has been serviced, UX packages a reply message and sends it to the application. The emulation library unpackages the reply parameters from the message and control is returned to the instruction following the trap [Russinovich94].

The middleware layer implemented for Mach 3.0 introduces the concept of a sentry mechanism. The central idea behind the sentry concept is that operating system entry and exit points provide sufficient visibility and control to support the majority of standard fault detection and tolerance techniques. These points, called sentry points, allow the encapsulation of operating system calls [Russinovich94]. This encapsulation of a system call is represented in Figure 6. The same path as in a standard system call is followed until the UX thread receives a system call request message. At that time, the thread checks to see if a monitoring policy is enabled for the requesting process for that system call. If any are enabled, their entry sentries are executed. The entry sentry passes control to UX to execute the system call and control returns to the application through the execution of an exit sentry.

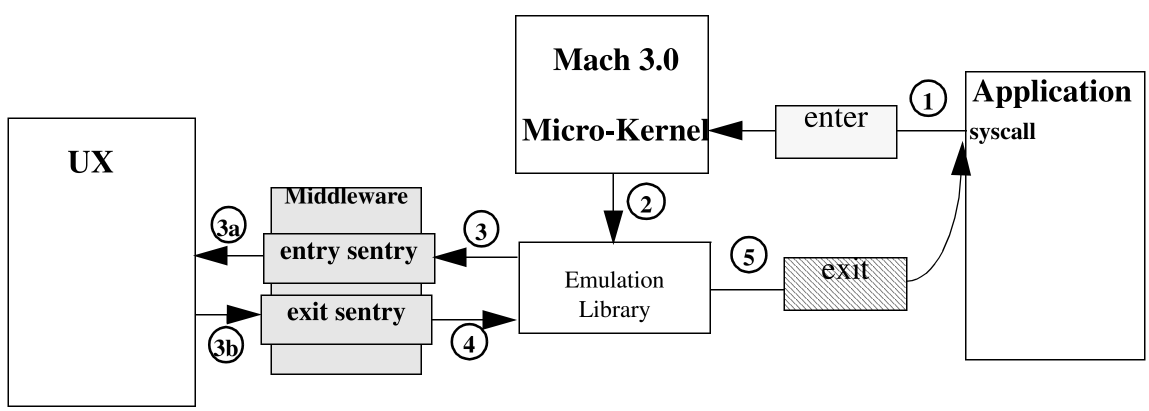

FIGURE 6. Mach Operating System Model with Middleware

Figure 7 represents the control flow for the execution of system calls in HP-UX. HP-UX is categorized as a monolithic kernel in which all system service and low-level control is contained in the kernel. All system calls are funneled through a single entry point in the system space, which is identified by space register 7 (sr7) [HPb95].

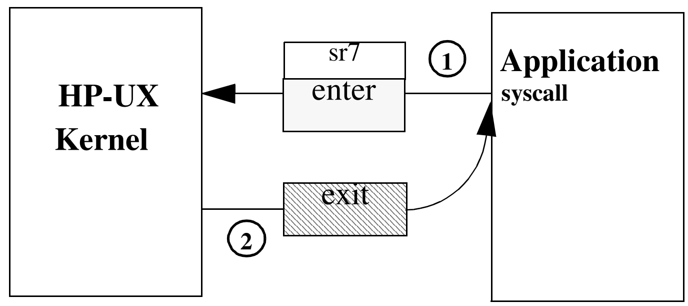

FIGURE 7. HP-UX Operating System model

To enable middleware for a system call, a custom library containing objects with middleware functions is linked with the application. This method is called HP Library Replace (HPLR). The control flow for HPLR is shown in Figure 8. Details of how code objects are linked to the application are provided in section 3.0.

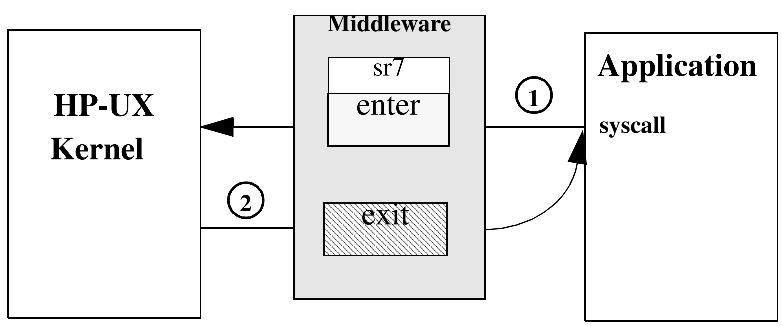

FIGURE 8. HP-UX Operating System model with Middleware (HPLR)

The last method investigated to build the taxonomy is represented in Figure 9. HP-UX is built with Kernel Instrumentation (KI) capabilities that are not enabled unless specifically requested by a user program.   A shareware program called trace1.6 was developed by Kartik Subbarao <Kartik_Subbarao@hp.com> which enables KI for the purpose of monitoring system and kernel calls.

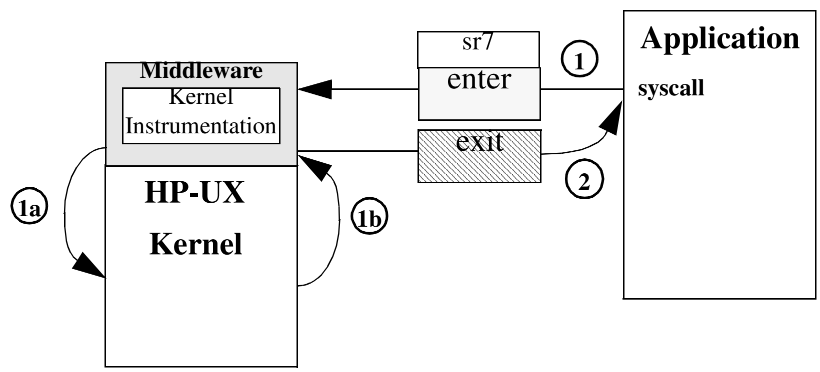

FIGURE 9. HP-UX Operating System model with Middleware (KI)

KI resides within the kernel and is enabled with code from a custom library provided with the trace program. When KI is enabled, data structures defined in the system are updated with the argument information and made available for the trace program to capture.

2.1 Middleware Attributes

Tables 1 and 2 summarize the main features of each middleware implementation evaluated for the taxonomy. The tables are split between implementation and fault-management attributes with the preferred attributes in bold-face type. Each attribute is described in detail in [Rekiere97]. Mach3.0 sentry and HP Trace are designed with all the preferred implementation attributes. HPLR is designed with all the preferred fault-management attributes. Overall, the Mach 3.0 sentry implementation meets the most preferred attributes defined in both tables.

TABLE 1. Middleware Implementation Attributes

Attribute	**pSOS**	**Mach3.0 sentry**	**HPLR**	**HP Trace**
Application (**T-Transparen**t, D-Dependent)	T	T	D	T
Middleware Enable (K-Kernel Boot, **U-User Program,** L-Library Link)	K	U	L	U
Libraries Required (**S-Standard,** C-Custom)	S	S,C	S,C	S,C
Attach to running process (**Y-Yes**, N-No)	N	Y	N	Y

TABLE 2. Middleware Fault-Management Attributes

Attribute	**pSOS**	**Mach3.0 sentry**	**HPLR**	**HP Trace**
System Call Visibility (**E-Enter, X-Exit**)	E	E,X	E,X	X
Log System Call (**N-Name, A-Arguments, R-Return Value, E-Error Codes**)	N	N,A,R	N,A,R,E	N,A,R,E
Roll-back Visibility (**Y-Yes**, N-No)	N	Y	Y	N
Assertion Visibility (**Y-Yes**, N-No)	Y	Y	Y	N

3.0 HP Library Replace (HPLR) Middleware Implementation

The development platform for the HPLR implementation is an HPPA RISC processor HP9000 series 750 workstation running the HP-UX 10.10 operating system. HP-UX is based on the UNIX System V Release 4 operating system[HPa95]. HPLR middleware is designed with two main objectives. First, to generate a log file of monitored system calls, arguments, return values and error codes. Second, to provide sufficient visibility for the implementation of assertions and roll-back recovery fault management features. The application and operating system features involved in executing a system call that are modifiable must be understood to achieve these objectives.

Three areas were considered for implementation of the middleware: kernel source, application binary, and the system library. The system library was selected and is reflected in the name of the method we chose, HP Library Replace (HPLR). The HPLR implementation creates a middleware-enabled application by linking a custom library to an application. This is similar to hard coding middleware into an application binary but with the added benefit of machine independence. Machine independence is achieved because the binary is built at the link level rather than at the machine code level. The following four sections provide an overview of system library and linking on HP-UX, a detailed description of the middleware layer implementation, an overview of system call execution on HP-UX and an analysis of HPLR's impact on system performance.

3.1 System libraries and the linker

System call code is located in the system library libc. System libraries contain object files generated by compilers and assemblers. System call code is linked to an application to create an application executable during the link-edit phase of compilation. The compilation phases of HP-UX are shown in Figure 10. The compiler (cc) produces an object file (main.o) from the application source (main.c). An object file contains definitions of code and data in a format the linker uses to build the application executable (a.out). The definitions can be local, global or external.

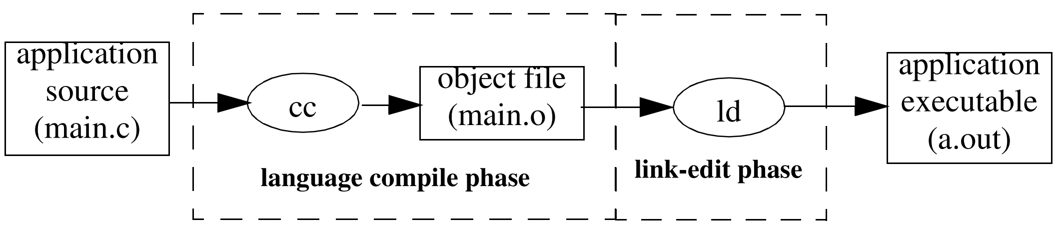

FIGURE 10. HP-UX Compiler Phases

A local definition of a routine or data indicates that it is accessible only within the object file in which it is defined. Global definitions contain the code for a named function, data or procedure. An external reference is the request for a local or global definition of a function, data or procedure in a program [HPc95]. System call code in libc is a global definition that an application requests with an external reference. Status information for the local, global and external definitions contained in object files is maintained by a symbol table. Each object file can contain multiple definitions of any or all types. Object file contents is determined by the source code it was created from.

Figure 11 shows an example of the symbol information for writecall.o, t_write.o and write.o. Writecall is a C program that executes a single write system call. Object files t_write.o and write.o contain the definitions to execute the write system call. The first column of the symbol table is the address of each symbol or reference. The second column denotes the symbol type. The last column shows the symbol name.

Object file symbols:

[writecall.o]:
0000000003 T main
0000000000 U write

/usr/lib/libc.a[t_write.o]:
0000000003 T _write
0000000000 U _write_sys
0000000003 TS write

/usr/lib/libc.a[write.o]:
0000000003 T _write_sys

Symbol Types:

T indicates a global definition
U indicates an external reference
d indicates a local definition of data
S secondary reference

FIGURE 11. Symbols defined for dwrite.o, t_write.o, write.o

A representation of how the linker matches external references and global definitions during the link-edit phase is shown for standard archive libraries in Figure 12. The writecall.o object file contains an external reference to write. The linker matches this external reference to the global definition of write in t_write.o. The three symbols contained in t_write.o are: _write, _write_sys, and write. The write and _write symbols are different names representing the same function to support compiler external reference naming conventions.

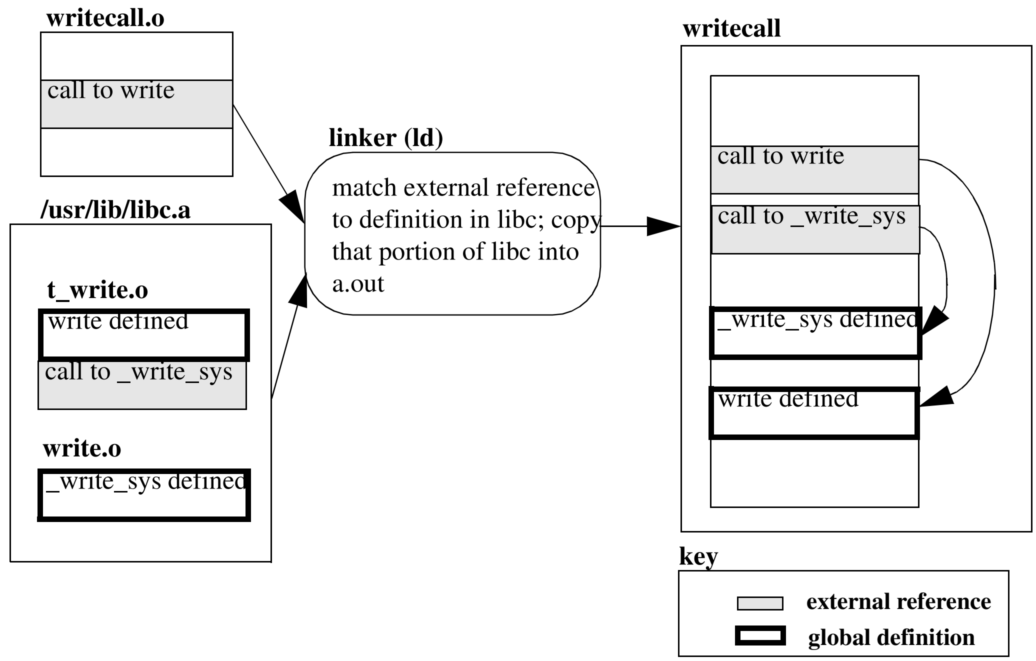

FIGURE 12. Matching external reference for write system call using standard libc.a

The _write_sys global definition contains code to enter the kernel and execute the actual system call. Using symbol table information, the linker generates the writecall binary exe-cutable file by matching external references to global definitions. When an application is linked to archive libraries, all code referenced by the application program is copied into the binary executable.

3.2 Creation of Middleware-Enabled libc

The middleware-enabled libc is created by replacing the global definition of system calls with middleware-enabled global definitions. The middleware enabled objects must have identical global definition names to the ones replaced so the linker will match them to the correct external references. When an application is linked with the custom libc, the middleware becomes part of the executable for the application. The application unknowingly enables the middleware by making an external reference to the middleware-enabled system call. Figure 13 shows an executable created with the modified libc for the write call.

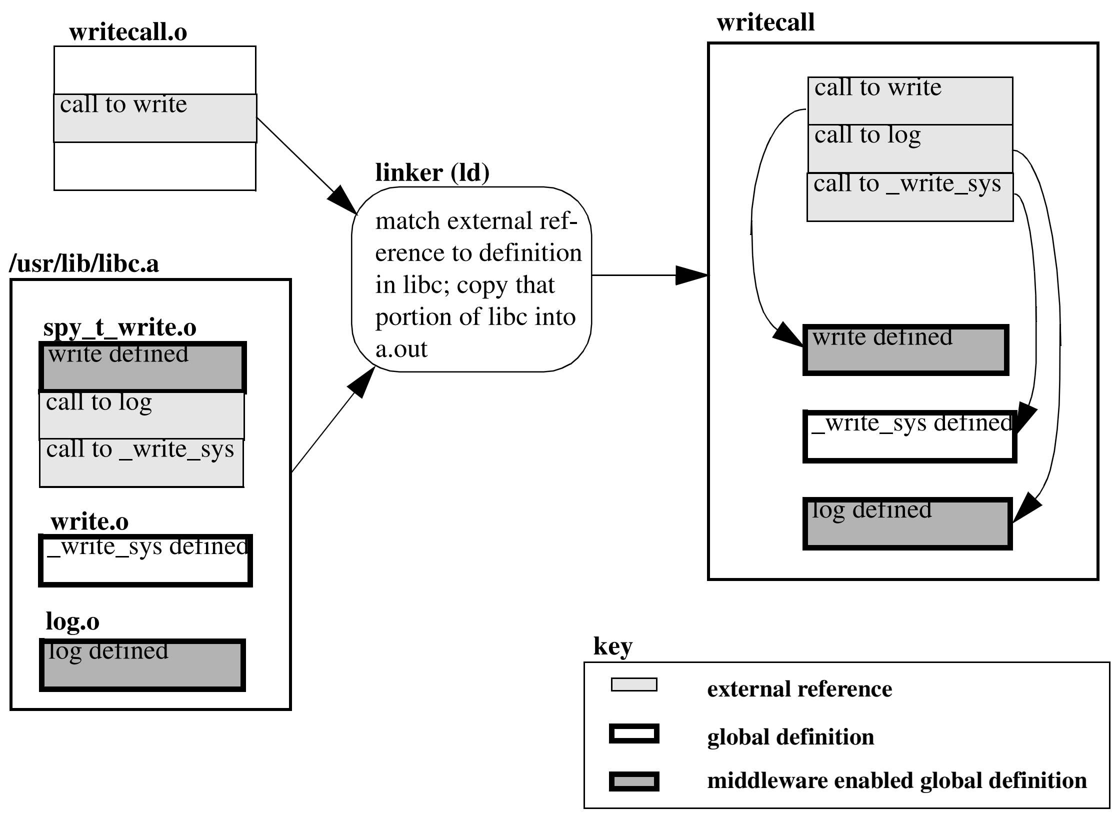

FIGURE 13. Matching external reference for write system call using middleware enabled libc.a

The main difference between Figure 12 and Figure 13 is the addition of spy_t_write.o and log.o object files. spy_t_write.o and log.o contain the middleware functions to perform monitoring. The middleware-enabled object files for system calls were given unique names of the form spy_t_syscall.o to prevent writing over the original default definitions contained in libc. The log.o object code defines the log function used by all middleware-enabled system calls to create the log file. Each monitored system call must have the global definition for the system call replaced with a middleware-enabled version. The HPLR middleware is invoked when the application executes a system call.

3.3 HP-UX System Call Execution

When the HPLR middleware is enabled, two symbols must be available in libc. One that matches the external reference made by an application that contains the middleware logging code. The second to execute the actual system call. Approximately 40% of the system calls in libc already have a separate symbol and code for the actual system call. In the cases where a separate symbol was not available one was created. Following the convention in libc, the actual system call symbols are of the form _syscall_sys.

Code for an actual system call is dependent on the target operating system's mechanism for entering system space, executing the call and returning control to the application. All system calls on HP-UX are funneled through a single entry point in the system space, which is identified by space register 7 (%sr7). System space is a portion of the virtual memory space in HP-UX where the kernel executes system calls and performs low-level system services. A system call is made on HP-UX by executing a BLE (branch and link external) instruction to enter the system space and loading the unique system call number into general register 22. A list of the system call numbers as well as the location of the system call entry point is in the standard include file: /usr/include/sys/syscall.h.

3.4 Performance and Benchmarks

A set of benchmarks were run to verify the HPLR middleware implementation. Benchmarks were used for verification rather than an application program to ensure a repeatable test environment. Unlike an application, a benchmark executes a finite number of system calls in a consistent order. Table 3 provides a breakdown of the type and frequency of each system call used by the benchmarks selected. The benchmarks include the BYTE magazine benchmarks (dwrite, dread, fcalla, fcalle, fibo, float, iofile, loop, pipes, scall, sieve, bytesort), Usenet (iocall) and Dhrystone (dryr, drynr) benchmarks. The benchmarks were compiled by Rick Richardson of PC Research Inc. and are available in a file called miscbench.tar.gz (http://src.doc.ic.ac.uk/public/public/packages/unix-c/benchmarks/).

TABLE 3. Count of System Calls made by Benchmarks

Benchmark / System Call	fcalla	fcalle	loop	sieve	fibo	float	dryr	drynr	bytesort	dwrite	dread	pipes	iofile	iocall	scall
brk							1	1							
close										2		2	2	1	
creat										1			1	1	
execve	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
exit	1	1	1	1	1	1	1	1	1	1	1	2	1	1	1
fork												1			
getpid															25000
ioctl					1	1	1	1	1				1		
lseek											256		1000	2002	
open										1	1		1	1	
pipe												1			
read											256	1025	500	4004	
times							4	4							
unlink											1			1	
write					1	2	2	2	12	256		1024	631	1001	
diversity (d)	2	2	2	2	4	4	6	6	4	6	6	7	9	9	3
frequency (f)	2	2	2	2	4	5	10	10	15	262	516	2056	2138	7013	25002
RDF [ratio: d/f]	1	1	1	1	1	.80	.60	.60	.267	.023	.012	.0034	.0042	.0013	.0001
	a					b				a				b	
	Group 1									Group 2					

Table 3 is separated into groups based on call frequency, diversity and the ratio of diversity to frequency (RDF). Call frequency is the total number of calls for a given benchmark. Call diversity is a count of the number of unique call types for a given benchmark. The groupings were established to determine which benchmarks most closely represent a typical user program to use for performance analysis of HPLR. System call diversity and frequency data for typical user programs from [Dingman96] is shown in Table 4.

The order of magnitudes of RDF and frequency data in Table 4 were used to split Table 3 into Groups 1 and 2. The benchmarks in Group 2 of Table 3 have a similar range of magnitudes for RDF and frequency as Table 4. Table 4 contains two distinct groups based on the order of magnitudes of RDF and frequency between gdb and xv in the table. Using the RDF value for xv as the dividing line, the split of Table 4 was applied to Group 2 of Table 3 to form sub-groups 2a and 2b. Benchmarks with an RDF higher then 0.0028 comprise Group 2a and benchmarks with an RDF lower then .0028 comprise Group 2b.

TABLE 4. Typical Application Diversity to Frequency Ratios [Dingman96]

	emacs	**gcc**	**gdb**	**xv**	**bitmap**
diversity (d)	23	20	23	26	18
frequency (f)	851	434	927	9140	13027
ratio (d/f)	.0270	.0461	.0248	.0028	.0014

Figures 14 and 15 show the execution time with logging enabled to libc.a execution time ratio for selected buffer sizes. The ratio gives a representation of the overhead created by the logging of system calls. As the buffer sizes increase the overhead decreases and the ratio approaches one. When a buffer is full it writes the values to disk and proceeds to fill and write data to disk until the benchmark finishes execution. The buf=all data value is the execution time when the logged data is written only to main memory. In summary, when logging is enabled to write to main memory only, there is a 1.5 times overhead compared to the standard libc and writing to disk adds a 5 to 10 times overhead the standard libc.

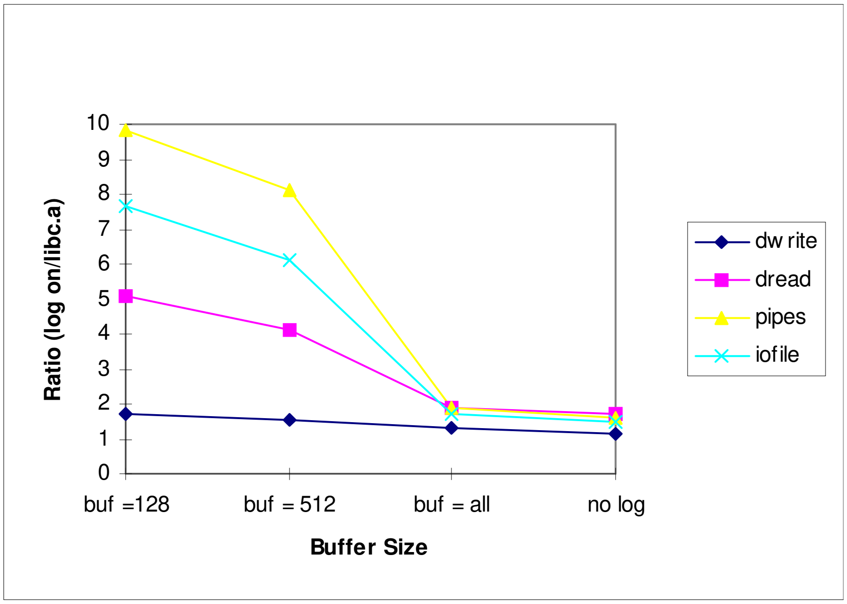

FIGURE 14. Group 2a benchmark execution time ratio: logging enabled to libc.a

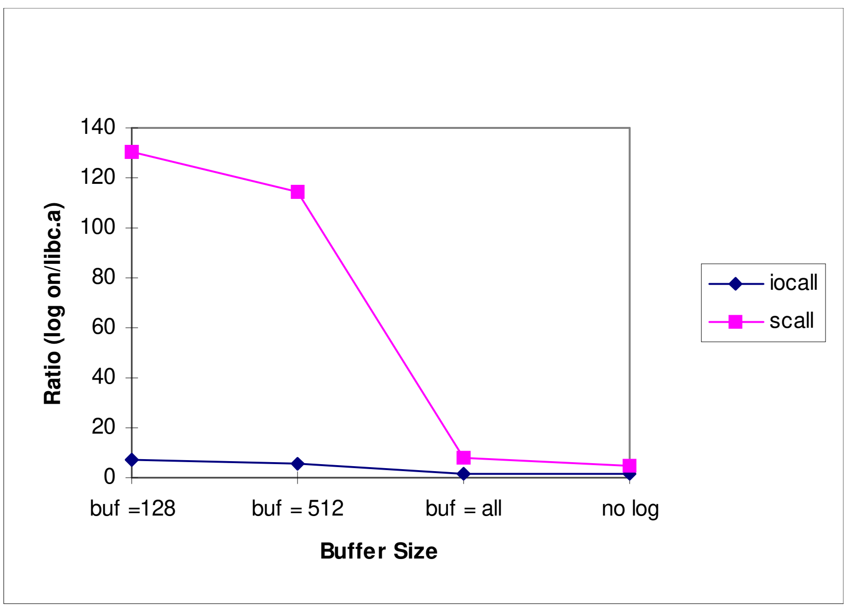

FIGURE 15. Group 2b benchmark execution time ratio: logging enabled to libc.a

The magnitude of the overhead is directly related to the number of log entries. Each log entry represents the execution of one system call. While overhead is high when writing logs to the disk, main-memory-only logging is effective. If the system call logging was expanded to include a Roll-back recovery mechanism every system call executed by an application would not be required to be logged[1]. In Roll-back recovery, a process uses information stored to restore a global consistent state and restart execution from that state, instead of restarting the computation from the beginning [Elnozahy93]

3.5 HPLR Implementation Summary

The HPLR system call monitor mechanism has been shown to be an effective fault-management middleware layer for HP-UX and accomplishes the first goal for HPLR stated in Section 3.0. The second goal, to provide sufficient visibility for the implementation of assertions and roll-back recovery, is achieved based on the location of the middleware in the execution process. [Russinovich94] defines two data sets necessary to perform roll-back recovery: user inputs, and events leading to data request from users. User inputs are visible to HPLR when the application makes a system call request and passes control to HPLR. Capturing events leading to data request from users can be done by enabling the system call monitor. Verifying argument parameters with assertions is possible when HPLR is passed control from the application. If an invalid argument is found, HPLR can

1. [Russinovich94] describes a roll-back recovery mechanism that logs only user input and events leading to user input. This reduces the number of system calls and arguments logged compared to the system call monitor mechanism. The roll-back recovery mechanism was run with 9% total overhead.

return an error to the application rather than executing the system call. Section 4.0 discusses the implementation of assertions in HPLR middleware.

4.0 Assertions and Robust Benchmarks

Robust Benchmarks stress a system by invoking system calls containing both valid and invalid parameter values. How the operating system handles those parameters is observed. It is expected that the incorrect system calls are representative of one form of errors made by application designers or corrupted data. The suite of Robust Benchmarks applied to Mach 3.0 in [Dingman96] was ported and run on HP-UX. The suite tests the read, write, open, close, stat, fstat, and select calls. These calls were selected because they were the most frequent functions in traces of user sessions[1] [Dingman96]. The robust test involving the select call is used to show assertions in HPLR because it resulted in fails on HP-UX.

The Robustness Benchmark test is built from the list of values in Table 5. Each value is passed as an argument to select until all the possible combinations of parameters have been used resulting in 405 test cases. Table 6 represents the Class 2 failure mode[2] argument combinations. The fails are considered in Class 2 because the select call exhibits the correct behavior even though some arguments are invalid and should have resulted in a return error message.

1. System calls were monitored for multiple user sessions running the following: emacs, gcc, gdb, bitmap, xv [Dingman 96].
2. [Dingman96] defines a six-level classification system of Robust Benchmark fail modes. Class 2 failures are defined as: system reports success when error should have been reported.

TABLE 5. Robust Benchmark Test Argument Combinations for the select call.

Number of file descriptors (nfds)	Read file descriptors (readfds)	Write file descriptors (writefds)	Except file descriptors (exceptfds)	Maximum interval to wait for selection (timeout)
0	NULL	NULL	NULL	NULL pointer
5	Valid pointer	Valid pointer	Valid pointer	Valid pointer
255	Bad pointer	Bad pointer	Bad pointer	-1 (bad pointer)
15000				
-1				

TABLE 6. Select Class 2 Fail Argument Combinations

nfds	readfds	writefds	exceptfds	timeout	Class 2 fail count
NULL	ALL 3	ALL 3	ALL 3	NULL	27

When timeout is NULL, as it is for all the cases in Table 6, select is defined to wait for a signal. The 27 fail cases were classified as Class 2 failures because the select call waits even though other arguments are invalid. This is considered a silent error because the call may or may not cause a failure when select gets a signal. In cases where timeout and nfds are non-NULL, invalid parameters are correctly caught by the operating system and return correct error codes.

Monitoring middleware was generated for the select call using the HPLR method described in Section 3.0. An assertion policy was added to the middleware to detect the invalid arguments. The assertion checks if nfds and timeout are NULL and returns error to the application. Since this case is always invalid (nfds must be greater than 1 because the bit mask is built from 0 to nfds -1). The bit mask parameters are irrelevant to check for this assertion because the operating system correctly returns error for invalid readfds, writefds, and execptfds if nfds is not NULL.

5.0 Summary

A taxonomy has been defined representing system call monitor fault management middleware layers for three commercial off-the-shelf operating systems and four middleware implementations: pSOS (embedded OS), Mach 3.0 (micro-kernel), and HP-UX (monolithic kernel). The features of each implementation were compared and described. The taxonomy provides a framework for understanding how fault-management middleware can be applied to commercial operating systems.

The HPLR middleware implementation on HP-UX was described in detail. Benchmarks run on the HPLR method showed the overhead and performance impact of system call monitoring with various logging configurations. Logging to main memory rather than any of the main memory/disk logging combinations added the smallest percentage overhead per system call logged.

Robust Benchmarks [Dingman96] were run on HP-UX and results for the select call were described. Assertions were added to the HPLR middleware for select to avoid incorrect kernel behavior. The HPLR method proved to be an effective fault-management layer without requiring modification to kernel or application source code. The implementation of fault-management middleware by replacing system call object code in an archive format system library was introduced. This method can be applied to any UNIX-type operating system that provides the capability to modify or replace individual objects in system libraries.

References

[Bach86] M.J. Bach, “The Design of the Unix Operating System”, Prentice-Hall, Englewood Cliffs, NJ, 1986.

[Dingman96] C. P. Dingman, “Robustness Benchmarking”, CMU Ph.D. Thesis, August 1996.

[Elnozahy93] E. N. Elnozahy, “Manetho: Fault Tolerance in Distributed Systems Using Rollback-Recovery and Process Replication”, Rice Technical Report, Rice COMP TR93-212, October 1993.

[HPa95] Hewlett-Packard Company, “HP-UX Reference”, HP LaserROM HP-UX Release 10.0, June1995

[HPb95] Hewlett-Packard Company, “HP-UX Memory Management White Paper”, HP LaserROM HP-UX Release 10.0, June1995

[HPc95] Hewlett-Packard Company, “Programming on HP-UX”, HP Part No. B2355-90652, Hewlett Packard Company, January1995

[Integrated95] Integrated Systems Inc., “pSOS System / 386 Release 2.0 Manual”, March 1995

[Lee95] R.E. Lee, “Middleware on the HP3000”, *Interact*, August 1995,

[Rashid89] R. Rashid, R. Baron, A. Forin, D. Golub, M. Jones, D. Julin, D. Orr and R. Sanzi, “Mach: a Foundation for Open Systems”, *Proc. 2nd Workshop Workstation Operating Syst.*, Sept. 27-29, 1989.

[Rekiere97] C. Rekiere, “Middleware Enabled Fault Management for Commercial Operating Systems”, CMU Master’s Thesis Report, April 1997.

[Russinovich94] M.E. Russinovich “Application-Transparent Fault Management”, CMU Ph.D. Thesis, August 1994.