1989

Experimental Facility for Kernel Extensions to Support Distributed Database Systems

Bharat Bhargava
Purdue University, bb@cs.purdue.edu

Enrique Mafla

John Riedl

Report Number:
89-930
EXPERIMENTAL FACILITY FOR KERNEL EXTENSIONS TO SUPPORT DISTRIBUTED DATABASE SYSTEMS

Bharat Bhargava
Enrique Mafia
John Riedl

CSD-TR-930
November 1989
Experimental Facility for Kernel Extensions to Support Distributed Database Systems*†

Bharat Bhargava
Enrique Mafia
John Riedl

Department of Computer Sciences
Purdue University
West Lafayette, IN 47907

Abstract

Operating system services can be implemented inside the kernel or at the user level. The decision depends on the performance-complexity tradeoff. Kernel-level functions are efficient but hard to implement. User-level implementations are generally penalized by poor performance and lack of security. This paper proposes a new approach to supplement and/or modify kernel facilities. Our experimental facility called Push is based on an extension language interpreted within the kernel, that provides the flexibility and security required. Our implementation provides the efficiency of kernel-resident code as well as the simplicity and safety of user-level programming. This facility enables experimentation that would be difficult and time-consuming in current environments. The overhead of the Push implementation can be factored out to give a good approximation of the performance of a native kernel implementation. We have used Push to implement kernel-resident communication services. A multicast implementation in Push has an inherent overhead of 0.32 milliseconds per additional site. The corresponding overhead for direct kernel-level implementation is 1.17 milliseconds and for a user-level implementation 0.57 milliseconds.

*Submitted to the 10th IEEE International Conference on Distributed Computing Systems
†This research is supported by NASA and AIRMICS under grant number NAG-1-676, NSF, AT&T, and by a LASPAU fellowship.
1 Introduction

Operating system services have to be constantly added and/or modified in order to adjust the system to changing environments and applications. New or alternative operating system facilities can be implemented either inside the kernel or in user-level processes. Many times, the decision is based on the simplicity versus efficiency argument. Complexity and efficiency are characteristic of kernel-resident code, while simplicity and poor performance are characteristic of user-level code. This paper describes a system called Push, that facilitates changing the functionality of the operating system kernel dynamically. It provides the flexibility and safety of user-level code and the efficiency of kernel-level code.

The Push system consists of a Push machine, a Push assembler, and a set of Push utilities. The Push machine is incorporated in the operating system kernel. It allows the user to run her own code inside the kernel. The Push machine hides the complex kernel data structures and mechanisms from the user, who can express the desired functionality in a high-level programming language. The assembler translates user-level code to the internal representation understood by the Push machine. Push utilities initialize the Push environment, add/delete assembled Push programs to/from the kernel, and print information about loaded Push programs. A prototype of this system has been implemented in the context of the Unix¹ operating system. We have used this prototype to conduct experiments on new kernel-resident communication services [BMR].

This system is appealing in a research environment, where different implementations of new operating system mechanisms and policies have to be tested. This is especially useful for conducting experiments in our Raid distributed database system [BR89]. Many database systems are implemented on top of existing, general-purpose operating systems. Using Push, we are able to experiment with implementations of database functions such as replication, recovery, and transaction management, in the operating system.

1.1 Operating System Support for Distributed Database Systems

Database implementors have suggested that additional support in the underlying operating system is needed for efficiency [Sto81, SDE85, Spe86]. Push provides a facility for experimenting with new or extended operating system services. Examples of these services include buffer management, file system support, process management, interprocess communication, concurrency control, atomicity control, and crash recovery. The services that are present in current operating systems are general-purpose and do not satisfy the demands of distributed transaction processing algorithms [Sto81, SDE85, BM89]. For instance, locking facilities and buffer management are generally implemented by database systems because the services

¹Unix is a trademark of AT&T Bell Laboratories.
provided in operating systems are inadequate.

Adequate operating system support for the implementation of database systems results in increased security, higher performance, and simpler coding of the transaction processing algorithms. Hoare proposed the small-kernel approach to operating systems [Hoa72]. His thesis is valid for time-sharing environments, where the basic task of the operating system is to share the computer resources among a variety of users. In this case, generalizing the operating system services to accommodate all potential uses of the system results in obtrusive, unreliable, and inefficient kernels. This is also true for general-purpose, networked systems [YTR+87, Che84]. However, the operating system support demanded by large applications like distributed database management systems can be determined in advance and be included in the kernel. This results in a specialized kernel, which provides optimum support for the implementation of reliable, high-performance database systems.

In section 2, we describe several approaches that have been used to achieve flexible/adaptable operating systems. Section 3 discusses design, implementation, and performance issues of Push. Section 4 describes experiments conducted with Push. Section 5 illustrates potential uses for our system. Finally, section 6 summarizes the paper and describes our future plans in this area.

2 Paradigms for Extensible Operating Systems

Several paradigms to achieve extensibility in operating systems have been proposed and implemented. They include parameterized operating systems, minimal kernels, synthesized code, streams, and the packet filter approach.

Monolithic operating systems offer limited degree of flexibility. Configuration files and compilation or boot-time parameters are used by those systems to alleviate the problem. Digital Equipment Corporation's configuration expert system, XICON, can assist users in the customized configuration of a complete computing system [BM84]. To avoid overcrowding in the kernel, certain operating systems services have be implemented as user-level processes. These processes called daemons, run in close relation with the kernel. However, because all crucial information resides inside the kernel, performance and even consistency cannot be guaranteed. For example, in the context of Unix, the use of a daemon to implement routing protocols introduces inconsistencies between the views of the routing tables for the daemon and the kernel. The Sun² network file system and Unix BSD networking services should have been implemented as daemons or user-level servers [LMKQ89]. Performance considerations forced the implementors to move this code into the kernel.

In the last decade, several small-kernel operating systems have been proposed and implemented [Che84, YTR+87, DRJLA88, RAAs88]. Under this model, the kernel provides only basic services, i.e., process and memory management, and interprocess communication.

²Sun is a trademark of Sun Microsystems, Incorporated.
On top of this infrastructure, a customized operating system can be built to support a given processing and hardware environment. Operating system services are provided as server processes. This allows changes in the server code easily. These servers can provide not only conventional operating system services such as file systems and network communication, but many other services for different applications. For example, we could have lock managers, atomicity controllers, consistency controllers to support distributed transaction processing. This approach is inappropriate for architectures with expensive context switches. For safety reasons, it is desirable that the kernel and the servers in the operating system be implemented in their own hardware protection domain. But this introduces a significant context switch overhead.

The Synthesis kernel suggests a solution that goes beyond the efficiency/power tradeoff that was mentioned above [PMI88]. This approach employs a monolithic kernel and uses several techniques to specialize the kernel code that executes specific requests. These techniques include the elimination of redundant computation and the collapsing of kernel layers. Synthesized code is reported to reduce the conventional execution path of some system calls by a factor of 10–20. This makes sense in general-purpose operating systems, where every user request has to be penalized by layers of code, that may be unnecessary for that specific request. For example, the Unix BSD model for interprocess communication, whose main goal is generality, results in an expensive sequence of procedure calls. Many of those procedure calls are irrelevant to individual messages [BMR87].

Streams increase the modularity and reusability of kernel code in the input–output subsystem [Rit84]. Streams try to eliminate the duplication of functionality existing in conventional device drivers. A stream is a two-way connection between a process and a device driver. Modules that process data flowing along this two-way path can be inserted and deleted dynamically, changing the behavior of the user interface. For instance, a user can create a stream between his process and a network device driver. Communication modules can then be added to that stream to implement a given suite of protocols.

The packet filter presents another alternative to the efficiency/flexibility dilemma for network code implementation [MRA87]. The packet filter demultiplexes network packets according to rules specified by the users. These rules can be quite complex and can be changed dynamically. By running inside the kernel, the packet filter eliminates much of the context switch overhead incurred by user-level demultiplexers. At the same time, the overhead introduced by the interpreter does not significantly affect the performance of network protocols when compared with native kernel code.

3 Design and Implementation

The Push approach is similar to the design of the packet filter [MRA87]. The packet filter is a low level facility for demultiplexing network packets to user processes. A process uses
a logical predicate to specify the packets it wants to receive. Communication protocols can be implemented in the user processes that receive the packets. This avoids direct changes to the kernel each time a new protocol is implemented. Push extends the idea to allow database support services to run in the kernel. Any algorithm can be written in Push, running entirely within the kernel. For instance, a multi-phase commit protocol can be written in this language that would send and receive two rounds of messages with a single system call.

Figure 1: The Push system architecture

Figure 1 shows the details of the Push architecture. The user writes a desired service in a high-level language. The user program is assembled into Push machine code. This code is then loaded into the kernel and stored in a special data structure. Now, the user can use the new operating system feature by invoking the corresponding Push routine with a special system call. This system call activates the kernel-resident Push machine, which runs the Push program on behalf of the user. The Push virtual machine provides the user with a high-level abstraction of basic kernel services, including primitives for process management, file system services, and interprocess communication.

Figure 2 illustrates the alternative approach of having the new service implemented at
the user level, as a separate server process. Note the context switch overhead introduced by the frequent need to cross the user–kernel boundary. The boundary crossing is necessary for two reasons. The user process and the server can communicate only through the kernel. Moreover, the server needs to access kernel tables and routines via the system call interface. For example, if the server process implements multicasting, the number of user–kernel interactions grows proportional to the number of members in the destination multicast group. In contrast, the Push approach requires only one such interaction.

3.1 Design Issues

In designing Push there are several considerations.

1. The Push machine should protect the rest of the kernel address space from access by the Push programs. An erroneous program may produce incorrect results for its users, but it must not violate the integrity of the kernel.
2. Push programs must be efficient to execute. If Push is inherently slow, the primary goal of achieving high performance cannot be met.

3. Push should provide simple timer services to the programs. In a distributed environment, error handling must include support for detecting lost messages.

4. A Push program must not be able to monopolize the CPU.

There are several approaches to protect the kernel address space from arbitrary access by Push programs. The first is to develop a user-level compiler that produces type-safe code, compiling in run-time checks where necessary. The compiler would mark the programs in an unforgeable manner and a privileged loader would be the only program with permission to push programs into the kernel\(^8\). Alternatively, the kernel could accept programs in the high-level language and compile the programs itself. The difficulty with these two approaches is that such a compiler would be difficult to port to new architectures. In addition, the loading of compiled programs safely into the kernel would be tricky. Implementing a compiler in the kernel has the further disadvantage that it would increase the kernel size. We chose to design a virtual machine within the kernel for running user programs. The Push machine is stack-based, with a simple instruction set, and a design that provides for simple implementation.

Performance is a potential problem of the virtual machine approach. Both the size of the virtual machine and the execution time of the Push instructions must be kept low. The size of the Push machine will affect the space left for user processes, and may lead to increasing the paging activity in the system. The virtual machine instruction set is similar to the stack language in [MRA87] which requires about 30 microseconds per interpreted instruction on a Microvax II. For simple functions like packet demultiplexing, performance is clearly better than the corresponding user-level implementation. In order to determine if favorable performance results can be achieved by the use of Push, we have to contrast the interpretation overhead with the disadvantages of user-level code.

In addition to protecting the kernel address space, we must prevent the monopolizing of the CPU by the processes running Push programs. This protection is achieved by running the programs with interrupts enabled. While executing kernel routines such as 'receive', interrupts are disabled as usual, but Push has no command to affect the interrupt status. Hence clock interrupts will occur as usual, and the kernel will make its normal time-slicing decisions. Unfortunately, Unix only replaces the executing process upon entering or exiting the kernel, and Push programs may loop indefinitely within the kernel. Our solution is to add code that checks for runaway Push programs to the clock interrupt routine. If a Push program is running when a clock interrupt occurs, the routine increments a special 'wound' counter in the Push program. If the wound counter is incremented beyond a fixed limit, the interrupt routine terminates the Push program, returning an error message to the

\(^{8}\)For instance, the compiler could include a cryptographic checksum in the compiled program.
user. In addition, the Push program is purged from the table of programs and a message is printed on the console, so that the same program does not continue to monopolize the CPU. Long-running Push programs may need a method to increase the number of clock ticks permitted.

Many of the Push programs will need timer services so messages can be retransmitted or timeout failures can be returned to the user. Our design supports a simple timeout facility that invokes the program at a specified label after a certain time (specified in milliseconds) elapses. The timeout is supported by the clock interrupt routine that keeps a list of pending timeouts in an increasing order of time. When a timeout expires, the clock routine checks to see if the program is still active. If so, the clock routine cleans up any queues on which the program was waiting, sets its execution point within the interpreter to the specified address, and returns the calling process to the run queue. When the process is rescheduled, it begins interpreting again at the new address.

3.2 Push Language Details

Push provides a simple stack-based language which can be executed efficiently within the kernel. The programs consist of two sections. The declaration section includes the declaration of input-output parameters, constants, and local variables. Parameters are of three types: input, output, and inout. Parameters and local variables can be defined as integers or as pointers to strings of bytes. Pointers must be assigned before they can be used. The executable section consists of a sequence of Push instructions. In addition to the stack operations, Push provides special operations that allows the user to access basic kernel services. Appendix A summarizes the operations available in Push. One operation is specified per line. Labels, if present, must proceed the operation code and the operands. Comments preceded by the character `%` can be inserted in a separate line or after a Push statement. Appendix B shows a sample Push program that implements multicasting.

The current implementation of the Push system includes an assembler for the stack language. The assembler translates user-level programs into Push machine code. This code is represented as an array of 4-byte words. Each declaration and instruction in the program is represented by one such word. The first byte stores the operation code, the second byte encodes information about the nature of the operand, and the last two bytes are used to store the operand itself. The operand can be a constant, a Push variable, or a pointer to a Push variable. The assembler is 884 lines of C code, and compiles to 60 Kbytes, unoptimized. A Push disassembler is 332 lines of C code, and 20 Kbytes compiled. A future implementation will include a compiler from a subset of C to the assembly language.
3.3 The Push Machine

Assembled Push programs are loaded into the kernel using a special system call, Pushcode. Pushcode takes two arguments: the name of a Push program and the address of the assembled program. The programs are stored in an array and are looked up by name when invoked. A table keeps information about the Push programs loaded into the kernel. This information includes the name of the program, its kernel address, length, owner, and access rights. The owner of a program can execute, remove, and overwrite it. Programs can be marked as sharable. This means that other users beside the owner can execute it. Program names that will be used by several users should be registered before users are permitted to login and marked as sharable. A separate system call is used to remove a program from the kernel's table. A third system call prints information about the loaded programs. A shell-level program accepts the name of a source Push routine, assembles it, and loads the assembled code into the kernel using the Pushcode system call.

A Push procedure that has been loaded into the kernel is invoked by a special system call, Pushrun. The call to Pushrun requires two arguments: the name of the Push procedure to be invoked and a pointer to a vector of arguments for the Push procedure. Each executing Push program is provided with an execution stack which contains the parameters, local variables, and the values dynamically pushed into it while the program is running. When a procedure is invoked, the arguments indicated in the program definition as input or inout are copied into the kernel address space. Arguments indicated as output are copied from kernel to user address space immediately before the Push procedure returns. Push programs can allocate/deallocate memory dynamically. A table records the address, length, and read/write access rights of allocated memory. When a process wants to access a block of dynamic memory to read or write, Push checks the boundaries of that block of memory against the information kept in the table. When the program terminates, all allocated memory is released automatically.

Push runs inside SunOS 4.0 in Sun 3/50's. The interpreter consists of 800 lines of C code, and takes about 10 Kbytes of memory. Ten Push programs of 100 statements each consume 5 Kbytes, including the run time stack. The entire Push implementation increases the size of the kernel by less than 20 Kbytes, which is relatively small compared to the total size of the kernel. We are using a streamlined version of SunOS 4.0, which is 584 Kbytes, including the Push interpreter.

4 Sample Experiments

To illustrate the utility of the Push software, we implemented the multicast and multi RPC programs listed in appendices B and C. We compared the performance of the Push programs with the performance of similar services implemented at the kernel and user levels. We used the SE suite of protocols in these experiments. SE (Simple Ethernet) is a set of
streamlined, low overhead communication protocols for the Ethernet [BMR87]. The three services compared in each of these experiments provide the same functionality.

4.1 Push multicasting

The programs considered for this experiment send the message to the set of destinations in the multicasting group and return. The user-level SE multicast utility is implemented on top of the SE device driver, which provides point to point Ethernet communication. In order to support multicast, this utility has to call the device driver for each member in the multicast group. The kernel-level SE multicast utility uses the multiSE device driver [BMRS89]. This device driver can send the same message to a group of destinations on the Ethernet with one system call. Figure 3 shows these three approaches for multicasting.

![Figure 3: Approaches for Multicasting](image)

In table 1 we compare the performance of the three multicast methods. Kernel-level SE multicast shows the best performance, and user-level SE multicast the worst. The difference

---

4The times were collected using Peter Danzig's and Steve Melvin's timer board. It uses the timer chip AM9513A from Advanced Micro Devices, Inc. The timer has a resolution of up to four ticks per microsecond.
between the times for kernel-level SE and Push is due to the interpretation overhead of the Push program. On the other hand, the multiSE driver takes significantly more effort to implement, debug and maintain. Writing and testing the Push multicast program are a matter of minutes.

<table>
<thead>
<tr>
<th>Number of destinations</th>
<th>kernel SE</th>
<th>user SE</th>
<th>Push</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1.2</td>
<td>1.2</td>
<td>2.7</td>
</tr>
<tr>
<td>5</td>
<td>4.2</td>
<td>5.9</td>
<td>6.6</td>
</tr>
<tr>
<td>10</td>
<td>8.0</td>
<td>11.7</td>
<td>11.0</td>
</tr>
<tr>
<td>15</td>
<td>11.7</td>
<td>17.5</td>
<td>15.6</td>
</tr>
<tr>
<td>20</td>
<td>15.4</td>
<td>23.4</td>
<td>20.2</td>
</tr>
</tbody>
</table>

Table 1: Multicasting timing (in ms)

A more precise picture of the intrinsic performance of the three methods is presented in table 2. The table shows the overhead added per additional destination in the multicasting group. This overhead includes the time consumed by the network interface, which is fixed. In our case, this time (0.6 ms) includes the conversion of the message to mbufs and their transmission over the cable. The first column represents the net overhead of each method. The execution of the loop in the Push program (13 Push instructions) takes about 320 μs, which averages 25 μs per instruction.

<table>
<thead>
<tr>
<th>Multicast method</th>
<th>Variable overhead</th>
<th>fixed overhead</th>
<th>Total overhead</th>
</tr>
</thead>
<tbody>
<tr>
<td>Kernel-level</td>
<td>0.15</td>
<td>0.60</td>
<td>0.75</td>
</tr>
<tr>
<td>Push</td>
<td>0.32</td>
<td>0.60</td>
<td>0.92</td>
</tr>
<tr>
<td>User-level</td>
<td>0.57</td>
<td>0.60</td>
<td>1.17</td>
</tr>
</tbody>
</table>

Table 2: Incremental processing time per destination (in ms)

Mbufs are special buffers used by the Unix communication subsystem.
4.2 Push multi RPC

The setup for this experiment is shown in figure 4. The user-level program has to make a separate system call for each send and receive. The Push program needs only one system call. It sends the message to all destinations and collects the answers before returning to the user. The user can set a timeout to detect site failures.

![Diagram of Multi RPC methods]

(a) User-level SE multi RPC  
(b) Kernel-level SE multi RPC  
(c) Push multi RPC

Figure 4: Multi RPC methods

Table 3 reports the results of this experiment. We did not implement a kernel-level version of multi RPC. The numbers in the first column are estimates that we obtained using the measurements observed in [BMR87].

4.3 Performance Improvements

Performance can be improved in several ways. The general purpose memory allocator for the SunOS kernel is too inefficient, especially for small chunks of memory. We measured 500 μs for the allocation–deallocation of 50 bytes. We plan to have our own memory allocation scheme to avoid this overhead. The relative high start-up cost (we measured 2.7ms for a single
destination and 0.9ms per additional destination) can be optimized by reducing the number of times Push has to cross the user/kernel boundary during input-output of parameters. Finally, the Push machine itself can be made more powerful to reduce the interpretation overhead (Push programs would consist of less instructions). For example, instead of the sequence `push a, push l, push m, send`, which is currently used to send the message m to network address a, we would have one instruction, namely `send m, l, a`.

5 Applications of Push

There are two types of applications for Push. It can be used as an experimental tool or as an operational tool. For example, if there are several alternatives to implement a given operating system functionality, programmers can quickly produce high level language prototypes of those alternatives. The prototypes can then be tested in the target environment before making the final implementation in the kernel. Push has the advantage that the rest of the system is not disrupted while the experiments are taking place. There is no need to recompile and reboot the kernel. In addition, the protection scheme of Push avoids system crashes due to bugs in the new services. When Push is used as an operational tool, Push routines can be added to or deleted from the kernel dynamically during normal operation of the system. This feature introduces a form of adaptability to the system. In the following paragraphs, we describe some applications for which Push can be an efficient and powerful tool.

Communication protocols. We have used Push to implement kernel-resident multicast and multi-RPC primitives. Distributed transaction processing algorithms heavily depend on those two mechanisms. 4.3BSD Unix does not provide either of them. SunOS does not have datagram multicasting. It has user-level point-to-point RPC, though. In both cases, it

<table>
<thead>
<tr>
<th>Number of destinations</th>
<th>kernel level SE</th>
<th>user level SE</th>
<th>Push</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2.2</td>
<td>3.0</td>
<td>6.6</td>
</tr>
<tr>
<td>5</td>
<td>9.5</td>
<td>14.9</td>
<td>14.6</td>
</tr>
<tr>
<td>10</td>
<td>18.5</td>
<td>29.7</td>
<td>25.0</td>
</tr>
<tr>
<td>15</td>
<td>29.5</td>
<td>44.3</td>
<td>35.6</td>
</tr>
<tr>
<td>20</td>
<td>36.5</td>
<td>59.0</td>
<td>46.2</td>
</tr>
</tbody>
</table>

Table 3: Multi RPC timing (in ms)
is the responsibility of the database developer to implement those facilities either explicitly, as user-level services or implicitly, hidden inside the algorithms. Having these facilities as kernel primitives leads to better performance, increased power, and modularity. Performance is improved primarily because of the elimination of context switches. The user is presented with high-level abstractions that reduce the amount and complexity of the application code. Finally, the same facility can be shared by different users and/or applications.

Commitment Protocols. In Camelot [Spe86], the authors suggest that certain distributed transactions protocols can be added to the operating system to improve performance and to raise the level of the operating system interface [Spe86]. In database-oriented operating systems, commitment protocols can be added to the kernel. During transaction processing, the addresses of the participant sites can be registered. When the system wants to commit the transaction, a single command in the database code will suffice. The performance is improved because of the reduced user-kernel interaction. The database system can also readily switch between alternative commitment protocols according to the demands of the system. Two-phase commit protocols are often used despite their blocking drawback [Ske82]. This is because the message exchanges that take place during each phase impose a significant overhead on the system. The performance improvements provided by Push can make the implementation of three-phase commit protocols a practical solution to the blocking problems.

Stream Modules. As mentioned in section 2, the stream model is a solution to the lack of modularity existing in the Unix I/O subsystem. Currently, only kernel-resident stream modules can be pushed to and popped from a stream. Push offers increased flexibility by allowing users to write and push their own modules, once the initial raw stream has been created. Here, we see a synergism, produced by the cooperative use of streams and Push. New communication protocol suites can be implemented and tested using a stream connecting the user with the network interface. Modules written in Push can then implement the different layers of the protocol suite.

Extended file systems The response time of transaction processing depends on the performance of the underlying file system. The user interface presented by the file system may not be convenient to implement transaction processing algorithms [Sto81]. We are designing experiments to extend the Unix file system to accommodate it to the demands of database systems. Push routines can change the semantics of file system operations. The user can implement additional file system operations. For example, Push routines can implement indexed access to file records, provide encryption capabilities, support recovery from crashes, etc. Similar extensions to the Unix file system have been proposed in [BP88]. Two problems with their approach are performance and security. There, the extensions to the file system
were implemented in user-level servers.

6 Summary and Future Work

For services that demand a constant interaction with the kernel, the performance advantages of Push over user-level implementations are clear. On the other hand, implementations of kernel code demand more effort than the corresponding implementations using Push. If performance is a real issue, and services have to be implemented inside the kernel, Push still can be used to test the services before the actual implementation takes place. The overhead in size and interpretation time introduced by Push are relatively small and their effects on the performance of an operating system service can be predicted with acceptable accuracy. We can determine the number of instructions executed by a Push program, and we have good estimates for the interpretation times of each Push instruction. This is important when using Push as an experimental tool to compare the performance of two potential kernel implementations of an operating system function.

Our next task is to define a simple yet powerful Push interface to basic operating system services already existing in the kernel. This will allow us to extend the range of Push applications. We want to extend the current implementation so that new operating services implemented with Push can be tested in the context of the Raid distributed database system [BR89]. We are not as interested in the individual performance of operating system services as we are interested in the global effect that the whole operating system has on distributed transaction processing.
References


### A Summary of the Push Operations

<table>
<thead>
<tr>
<th>Operation</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>push $i$</td>
<td>push the value $i$ on the stack</td>
</tr>
<tr>
<td>pop $v$</td>
<td>pop a value off the stack and assign it to variable $v$</td>
</tr>
<tr>
<td>dec $v$</td>
<td>decrement the value of $v$</td>
</tr>
<tr>
<td>inc $v$</td>
<td>increment the value of $v$</td>
</tr>
<tr>
<td>add</td>
<td>pop 2 values off the stack, place their sum back on the stack</td>
</tr>
<tr>
<td>sub</td>
<td>pop 2 values off the stack, place their difference back on the stack</td>
</tr>
<tr>
<td>jmp $l$</td>
<td>jump to label $l$</td>
</tr>
<tr>
<td>jeq $l$</td>
<td>pop two elements off the stack; jump to label $l$ if they are equal</td>
</tr>
<tr>
<td>jneq $l$</td>
<td>pop two elements off the stack; jump to label $l$ if they are not equal</td>
</tr>
<tr>
<td>alloc $v$</td>
<td>pop the stack, allocate that many bytes, to $v$</td>
</tr>
<tr>
<td>free</td>
<td>pop the stack, free the block that starts at that address</td>
</tr>
<tr>
<td>copy $a b$</td>
<td>copy $l$ bytes from $a$ to $b$</td>
</tr>
<tr>
<td>compare $a b$</td>
<td>compare $l$ bytes from addresses $a$ and $b$; place 0 on the stack if they are equal, 1 otherwise</td>
</tr>
<tr>
<td>send $m a$</td>
<td>send $l$ bytes starting at $m$ to network address $a$</td>
</tr>
<tr>
<td>recv $m a$</td>
<td>receive at most $l$ bytes at address $m$, place source address at $a$</td>
</tr>
<tr>
<td>settimer $s$</td>
<td>set a timer for $s$ seconds; if the timer expires jump to label $l$</td>
</tr>
<tr>
<td>stoptimer</td>
<td>disable a timer set earlier</td>
</tr>
<tr>
<td>treset</td>
<td>start timing</td>
</tr>
<tr>
<td>tprint</td>
<td>stop timing, place elapsed time on the stack</td>
</tr>
<tr>
<td>print $v$</td>
<td>print integer $v$</td>
</tr>
<tr>
<td>prints $v$</td>
<td>pop the stack, print that many bytes, starting at address $v$</td>
</tr>
<tr>
<td>return</td>
<td>return to the user level Pushrun call</td>
</tr>
</tbody>
</table>
B  Push Multicast Program

%Push multicast procedure

| addrlen | def | 6 |
| addr    | in  | address |
| addrcnt | in  | integer |
| msg     | in  | address |
| msglen  | in  | integer |
| nxtaddr | var | address |

    push addr                % nxtaddr = addr
    pop nxtaddr

loop    push nxtaddr        % send (msg, msglen, nxtaddr)
    push msglen
    push msg
    send

    push nxtaddr        % nxtaddr = nxtaddr + addrlen
    push addr len
    add
    pop nxtaddr

    push addrcnt         % addrcnt = addrcnt - 1
    dec
    dup
    pop addrcnt

    jgt loop              % if (addrcnt > 0) goto loop

return
C  Push Multi RPC Program

%Push multi RPC procedure

<table>
<thead>
<tr>
<th>addr1en</th>
<th>def</th>
<th>6</th>
</tr>
</thead>
<tbody>
<tr>
<td>addr</td>
<td>in</td>
<td>address</td>
</tr>
<tr>
<td>addrcnt</td>
<td>in</td>
<td>integer</td>
</tr>
<tr>
<td>msg</td>
<td>in</td>
<td>address</td>
</tr>
<tr>
<td>msglen</td>
<td>in</td>
<td>integer</td>
</tr>
<tr>
<td>replies</td>
<td>out</td>
<td>address</td>
</tr>
<tr>
<td>replen</td>
<td>in</td>
<td>integer</td>
</tr>
<tr>
<td>error</td>
<td>out</td>
<td>integer</td>
</tr>
<tr>
<td>nxtaddr</td>
<td>var</td>
<td>address</td>
</tr>
<tr>
<td>nxtrepl</td>
<td>var</td>
<td>address</td>
</tr>
<tr>
<td>faddr</td>
<td>var</td>
<td>address</td>
</tr>
<tr>
<td>resp</td>
<td>var</td>
<td>address</td>
</tr>
<tr>
<td>j</td>
<td>var</td>
<td>integer</td>
</tr>
<tr>
<td>k</td>
<td>var</td>
<td>integer</td>
</tr>
</tbody>
</table>

push addr  % nxtaddr = addr
pop nxtaddr

l1 push nxtaddr  % send (msg, msglen, nxtaddr)
push msglen
push msg
send

push nxtaddr  % nxtaddr = nxtaddr + addr1en
push addr1en
add
pop nxtaddr

push addrcnt  % addrcnt = addrcnt - 1
dec
dup
pop addrcnt

jgt l1  % if (addrcnt > 0) goto l1

20
push 6 % faddr = alloc (6)
alloc faddr
push 4 % resp = alloc (4)
alloc resp
push addrcnt % i = addrcnt
pop i
push &12 % settimer (20, &12)
push 20
settimer

13 push faddr % recv (resp, &replen, faddr)
push &replen
push resp recv
push addrcnt % j = addrcnt
pop j
push addrs % nxtaddr = addrs
pop nxtaddr
push replies % ntrepl = replies
pop ntrepl

14 push 6 % if (compare (faddr, nxtaddr, 6) != 0)
push nxtaddr % goto 15
push faddr compare
cmp 15

push replen % else
push ntrepl % copy (resp, ntrepl, replen)
push resp % goto 16
copy
jmp 16

15 push nxtaddr % nxtaddr = nxtaddr + addrlen
push addrlen
add
cmp 21

pop nxtaddr

def
push  ntrepl  \% ntrepl = ntrepl + replen
push  replen  
add
pop  ntrepl

push  j  \% j = j - 1
dec
dup
pop  j
gtt  l4  \% if (j > 0) goto l4

l6  push  i  \% i = i - 1
dec
dup
pop  i
gtt  l3  \% if (i > 0) goto l3

stoptimer  \% stoptimer
push  0
pop  error  \% error = 0
return

l2  push  1
pop  error  \% error = 1
return

22