PUMA: An Operating System for Massively Parallel Systems

Stephen R. Wheat[1]
Rolf Riesen[1]

MPCRL
Sandia National Laboratories
Albuquerque, NM 87185-5800

Arthur B. Maccabe[1,2]
David W. van Dresser[1]

Department of Computer Science
The University of New Mexico
Albuquerque, NM 87131

T. Mack Stallcup[3]

Supercomputer Systems Division
Intel Corporation
Beaverton, OR 97006

Abstract

This paper presents an overview of PUMA, (Performance-oriented, User-managed Messaging Architecture), a message passing kernel. Message passing in PUMA is based on portals—an opening in the address space of an application process. Once an application process has established a portal, other processes can write values into the portal using a simple send operation. Because messages are written directly into the address space of the receiving process, there is no need to buffer messages in the PUMA kernel and later copy them into the applications address space.

PUMA consists of two components: the quintessential kernel (Q-Kernel) and the process control thread (PCT). While the PCT provides management decisions, the Q-Kernel controls access and implements the policies specified by the PCT.

1 Introduction

PUMA (Performance-oriented, User-managed Messaging Architecture) is a joint project between the Parallel Computing Sciences Department at Sandia National Laboratories and the Computer Science Department at the University of New Mexico. The PUMA project was initiated in January of 1991 with the goal of developing an operating system that would be compatible with Vertex (the vendor supplied operating system for the nCUBE-2) and could be used to explore alternate message passing schemes. In August of 1991, we completed an initial implementation of this operating system. Subsequently, this implementation has been and is being used to explore high-performance, massively-parallel computing.

1. This work was performed at Sandia National Laboratories, operated for the U.S. Department of Energy under contract No. DE-AC04-76DP00789.

2. On sabbatical at Sandia National Laboratories.

3. On-site at Sandia National Laboratories.

In January of 1992, the joint development team undertook the design and implementation of a new operating system, PUMA. Early in the design process we identified six goals for the PUMA effort:

- PUMA would be developed for massively parallel (MP) environments, that is, environments with thousands of processor nodes and a tightly coupled, reliable communication network.
- PUMA would be portable across MP, distributed memory machines.
- PUMA would be developed to support scalable, performance-oriented applications, i.e., applications that could be scaled to consume all of the resources provided by an MP environment.
- PUMA would provide a reliable and robust environment for the development of applications.
- PUMA would provide an open architecture for the development of application level libraries, i.e., it must be possible to develop efficient, user-level library routines to implement any message passing paradigm.
- The development of PUMA would emphasize efficiency over functionality.

Like many of the operating systems developed for distributed processing environments (e.g., Amoeba [9], Chorous [10], Mach [1][11], and V [5][6]), the PUMA architecture is based on a message passing kernel. However, unlike many of these systems, PUMA has been developed for an environment in which the communication network is trusted and controlled by the kernel. Whenever a kernel receives a message, it knows that the message was accepted and transmitted by a kernel running on another node in the system. This avoids the need to authenticate messages and simplifies many of the tasks that need to be performed by the kernel and application processes.

In the next section we introduce the basic architecture of PUMA, consisting of the quintessential kernel (Q-Kernel) and the process control thread (PCT). In that section we introduce the essential principle used in the

design of PUMA: levels of trust. The third section describes portals, the communication structure. The fourth section describes the communication control polices and mechanisms embedded in the PCT and Q-Kernel. The fifth section describes the structure of the Q-Kernel, while the sixth section describes the structure of the PCT. In the seventh section we compare the structures of PUMA to other systems with similar goals In the final section we summarize the results of running PUMA on a nCUBE2 and an Intel Paragon.

2 An Overview of PUMA

The PUMA architecture is based on three levels. The quintessential kernel (Q-Kernel) represents the lowest level in the PUMA architecture. The Q-Kernel provides basic communication facilities and address space protection. The process control thread (PCT) represents the next level in the PUMA architecture. The PCT provides process management functions (e.g., process creation and scheduling) and group level protection. The server/application process level represents the third level in the PUMA architecture.

Each processor node has a Q-Kernel, a PCT, and a collection of server and application processes. It is worth noting that most of the processor nodes will only have application processes. In most configurations, server processes are only present on processor nodes that provide special resources (e.g., disk drives or networking facilities). Figure 1 illustrates the logical structure of a Puma node.

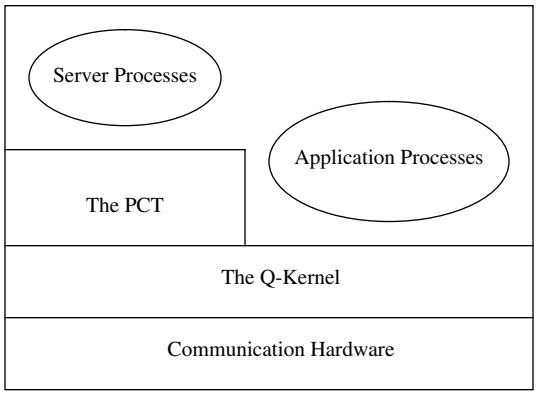

FIGURE 1. The Logical Structure of a PUMA Node

2.1 Levels of Trust

Each level in the PUMA architecture represents a level of trust. The kernels trust the communication hardware to provide correct and secure communication between processor nodes. In addition, each kernel trusts the kernels running on other processor nodes to correctly implement their specified behavior. However, the kernels do not trust the PCTs or server and application processes. Each PCT trusts the hardware, the kernels, and the other PCTs; however, they do not trust the server and application processes. Server and application processes trust the hardware, the kernels, and the PCTs but do not, in general, trust other application processes.

The trust relation is a partial order. The communication hardware represents the most trusted level—all of the other levels trust the communication hardware. The Q-Kernel is the most trusted level of software, while the PCT is the next most trusted level. Trust does not represent a total order because application processes do not necessarily trust one another and, as a consequence, are incomparable with respect to trust.

The PUMA architecture ensures that the data structures maintained by one level can only be corrupted by (a malfunction in) the level itself or a more trusted level. For example, the data structures maintained by the Q-Kernel cannot be corrupted by a (malfunctioning) PCT, server or application process. To ensure this degree of security, implementations of the PUMA architecture need to provide distinct privilege/protection domains for each level of trust.

Figure 2 illustrates the relations between the privilege/protection domains in PUMA. The kernel domain includes all of the physical resources for a processor node. When it begins its execution, the kernel identifies the memory that it needs for its data structures and maps the remaining memory into the address space for the PCT. Whenever the PCT loads a server or application process, it allocates a portion of its address space for the application process and constructs a new protection/privilege domain.

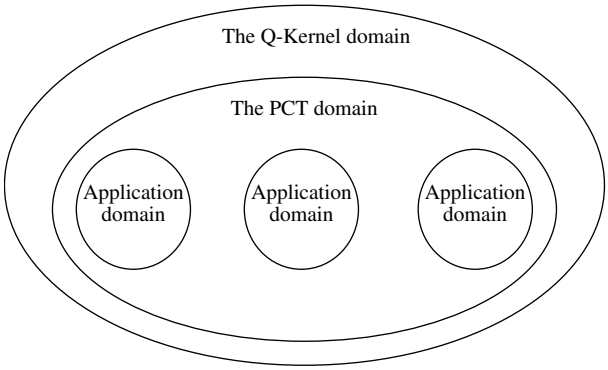

FIGURE 2. Protection/Privilege Domains in PUMA

2.2 Control and Management of Resources

The Q-Kernel is responsible for controlling access to the physical resources provided by a processor node, while the

PCT is responsible for managing access to these resources. For example, the Q-Kernel enforces execution quanta on all server and application processes; however, the PCT is responsible for determining the size of the quanta along with all other scheduling decisions. In many operating systems, control mechanisms and management policies are integrated into a single monolithic kernel. In designing PUMA, three considerations led us to separate these responsibilities into separate modules.

First, we noted that control decisions occur far more frequently than management decisions. As an example, consider the use of the communication network. Management policies determine which processes can be named in a communication request while control decisions determine if a particular communication request is valid (in the context of the management policies). Given this difference in frequency of decisions, the separation between the PCT and the Q-Kernel reflects different levels of concern regarding the efficiency of implementation. In this case, the separation allowed us to concentrate on the efficiency of control activities without needing to consider the impact on management activities.

Second, because the Q-Kernel code must interact directly with the hardware, this separation reflects different concerns regarding portability of code. While we expect that a good deal of the kernel code will need to be modified when we port PUMA to different architectures, we expect that a significantly smaller amount of the PCT code will need to be modified. (Here, we should note that the PCT is not totally portable. In particular, a small amount of the PCT code reflects the address mapping performed by the underlying hardware.)

Third, we noted that management policies are changed more frequently than control mechanisms. (Alternately, a set of control mechanisms can be used to implement a variety of management policies.) In this case, the separation between the PCT and Q-Kernel reflects different levels of concern regarding the openness of the system. In this case, we expect to be able to run several different PCTs on the Q-Kernel. For example, one PCT might only provide single tasking while another might provide prioritized multitasking.

3 Portals

In designing PUMA, we sought to minimize the need to use memory copies during communication. As inter-node communication rates approach (and even exceed) memory copy rates, the need to minimize memory copies has become a critical aspect in the efficient use of the resources provided by an MP machine. As an example, we have been able to achieve inter-node communication rate in excess of 160 Mbytes/second on an Intel Paragon. However, we have only been able to attain memory copy rates of 70 Mbytes/second on the same machine. When communications require a memory copy, the effective throughput drops to approximately 55 Mbytes/second.

Message passing in PUMA is based on the concept of a portal—an opening in the address space of an application process. A portal provides memory in the address space of the receiving process that can be modified or read by other processes. Processes use a message send operation to write to (or read from) the memory associated with a portal in another process.

All message transmissions are asynchronous with respect to the execution of application processes. To send a message, an application process registers a message buffer with the Q-Kernel, specifying a destination process and portal on the destination process. After the message buffer has been registered, the contents of the message buffer are transmitted to the specified portal of the destination process. The sender and receiver processes are notified by their respective Q-Kernels when the transmission is complete.

PUMA provides four types of portals: kernel managed portals, receiver managed portals, sender managed portals, and read memory portals. The memory associated with the portal is in the address space of the application. The different types of portals are distinguished by the management policy associated with the portal memory.

It is possible that a message will arrive when there is not sufficient space in the memory associated with a portal to hold the contents of the incoming message. When this happens, the Q-Kernel discards the message and notifies the receiving process that a message was dropped. It is the application's responsibility to provide flow control in portal usage. Because flow control is not embedded in the Q-Kernel only applications that need external flow control will be burdened with the additional costs associated with flow control (e.g., the implementation of an RTS/CTS protocol). This approach simplifies the structure of the Q-Kernel and reduces the overhead required for self-synchronizing applications.

3.1 Kernel Managed Portals

From the user's perspective, the simplest type of portal is a kernel managed portal. The memory associated with a kernel managed portal is managed as a dynamic heap that is shared between the Q-Kernel and the application process.

To use a kernel managed portal, the application process needs to initialize and register a block of its memory as a kernel managed portal. After the memory has been initialized and registered, the Q-Kernel will dynamically allocate buffer space in the portal heap whenever a

message arrives. After the message has been delivered, the kernel appends the message to a list of received messages. This list is maintained in the portal heap and, as such, is also shared between the application and the Q-Kernel. The application can "receive" messages from the portal message list as needed. When it is done processing a message, the application can remove the message from the list and release the associated memory, allowing the kernel to reuse this memory for another message.

Figure 3 illustrates the structure of a kernel managed portal. As shown, the message list maintained inside the portal memory starts with a dummy header cell. This header cell is created when the portal memory is initialized. Each element of the list has a header followed by the body of the message.

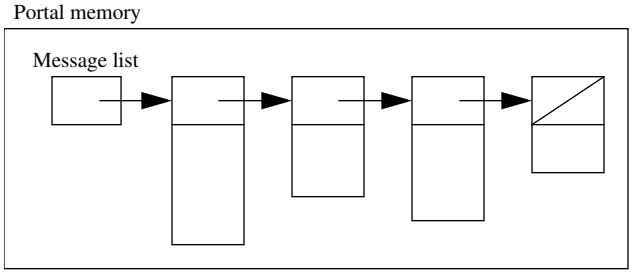

FIGURE 3. A kernel managed portal

Kernel managed portals provide the user with flexibility in managing the memory used for communication. The application only needs to predict the maximum amount of space needed for incoming messages. Moreover, because the memory associated with the portal is part of the application's address space, the application can use this memory for other activities when it is not being used to hold incoming messages.

While kernel managed portals provide the application programmer with flexibility, they have two drawbacks. First, the fact that the kernel has to perform a dynamic allocation on every message receipt increases the time required for communication, in particular by increasing latency. Second, and perhaps more important, is the potential need to copy messages from the portal memory into the memory structures used by the application. Because the kernel manages the space used for incoming messages (using dynamic allocation in the portal heap), the application cannot control the placement of the arriving message and may need to copy messages from the portal memory into other application data structures.

3.2 Receiver Managed Portals

The memory associated with a receiver managed portal is managed by the receiving process. For a receiver managed portal, the receiving process pre-allocates buffers for the messages that it expects to receive from other processes. When messages are sent to a receiver managed portal, they are mapped directly into one of the pre-allocated buffers.

When an application uses a receiver managed portal, it first allocates space for a collection of message buffers and an array of message buffer descriptors—one descriptor for each message buffer. The application then initializes the array of message descriptors so that each descriptor points to one of the message buffers.

Figure 4 illustrates the structure of a receiver managed portal. In this case, the message header information is recorded in the message buffer descriptor and the message bodies are separate from the message headers.

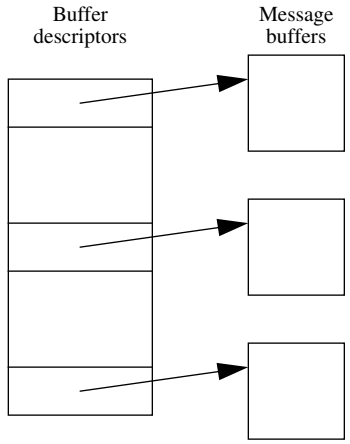

FIGURE 4. A receiver managed portal

The array of buffer descriptors is managed as a circular queue. The Q-Kernel maintains a pointer to the next message buffer descriptor in the portal. When a message arrives, the kernel places the message in the message buffer identified by the next buffer descriptor and advances the associated buffer descriptor pointer to the next buffer descriptor.

In contrast to kernel managed portals, receiver managed portals require more explicit initialization on the part of the application programmer and offer very little flexibility. For example, all of the message buffers associated with a receiver managed portal must be the same length. To use this type of portal, the programmer must have a thorough understanding of the communication patterns exhibited by the application program.

While they are more difficult to use, receiver managed portals can be more efficient in many applications. First, because the receiver can pre-allocate the message buffers, this strategy reduces the latency associated with message reception. Second, because the receiving application can control where messages are delivered, this strategy minimizes the need for the memory-memory copies that might be required when using a kernel managed portal

3.3 Sender Managed Portals

The memory associated with a sender managed portal is effectively managed by the processes that send messages to the portal. To create a sender managed portal, a process simply registers a block of its memory as the portal. Sending processes specify offsets into this block when sending messages. When a message arrives for a sender managed portal, the Q-Kernel transmits the body of the message to the portal memory at the specified offset.

The Q-Kernel does not record any structural information for a sender managed portal. It does not notify the process when messages are delivered to the portal. Moreover, it is possible for processes to overwrite messages sent by other processes. Sending processes must coordinate their use of sender managed portals. This includes notifying the process when a complete message has been delivered to the portal and avoiding overwrites in the portal memory.

Sender managed portals were designed to support parallel servers—collections of processes that provide shared resources for applications. For example, a parallel file server may be partitioned into several processes (perhaps one per disk). To read a block from a file, the application would start by allocating a block of its memory to hold the data. After registering this block of memory as a sender managed portal, the application would send a read request to one of the server processes. Different server processes could then fill in the appropriate portions of the portal memory block. Note that the application process does not need to know how the server is organized to make use of the resource provided by the server.

Notice, there is no way for the

3.4 Read Memory Portals

Read memory portals represent the converse of sender managed portals. While sender managed portals provide a write operation for the address space of another process, read memory portals provide a read operation for the address space of another process. As with the sender managed portal, the memory associated with a read memory portal is managed by the processes that read memory from the portal.

To establish a read memory portal, the application process registers a block of its memory as the portal. Request messages sent to a read memory portal specify an offset, length and reply portal (on the requesting process). When the Q-Kernel receives a message for a read memory portal, it generates a response and sends it to the reply portal of the requesting process. The response message consists of the memory values starting from the specified offset in the portal and has the length specified in the request message.

Like sender managed portals, read memory portals were designed to support parallel servers. For example, to write a block of memory to a parallel file, the application starts by registering the data block as a read memory portal and sends a write request to a server processes. The server processes then send request messages to the read memory portal as they are able to consume another block of the file.

3.5 The Portal Table

An application process may have several portals for receiving (and sending) messages. Each portal has an associated portal descriptor that specifies the portal type (kernel managed, receiver managed, or sender managed) and other important portal information. In the case of a receiver managed portal, the portal descriptor includes the address of the buffer descriptor array, an integer specifying the size of each message buffer, an integer specifying the number of buffers, and an index for the current message descriptor. For a sender managed portal, the portal descriptor only includes the starting address and length of the memory block associated with the portal.

All of the portal descriptors for an application are stored in an array called the portal table. Sending processes provide an index into the portal table of the receiving process when they send a message. When a message arrives, the Q-Kernel on the destination processor node first determines the destination process for the incoming message. The portal index provided by the sender is then used to determine which portal should be used for receiving the message. Once the Q-Kernel has determined the target portal, it uses the portal descriptor to determine how the message should be mapped into the associated portal memory.

4 Communication Control

In this section we describe PUMA's communication control strategy. Given the basic communication mechanisms, one application process can flood the portals of another application process by sending messages. While we did not feel it was important to implement flow control mechanisms in the Q-Kernel, we did feel it was important to control which processes could flood the portals of a process.

PUMA's communication control policy is based on the notion of a process group—the group of processes that are loaded in a single parallel application. In most cases, interprocess communication (IPC) is limited to the processes in a single process group. However, the mechanisms of PUMA have been designed to permit dynamic intergroup communication as well as communication with server processes.

4.1 Process Groups

The processes in a parallel application are collectively called a *process group*. When a parallel application program is loaded, the application is assigned a unique *group identifier.*

The notion of process group provides the basis for communication control in PUMA. The PCT maintains a *group control list* for each application and server process on a node. A process can only send messages to processes that are in process groups that are in the group control list of the sending process.

Initially, the group control list for a process only contains the group that the process belongs to. Other groups can be added using the group registration and server connection facilities provided by the PTC.

4.2 Process Identifiers

In addition to its process group, each process is assigned a *rank identifier* that is unique within the group. A rank identifier is an integer in the range 0 to *n*–1, where *n* is the number of processes in the parallel application.

From the perspective of an application process, a process identifier is naturally expressed as an ordered pair consisting of a group and rank identifier. However, when it transmits a message, the Q-Kernel needs to know the physical node number and process index for the receiving process. (A process index is a small integer that identifies the process in the context of the destination node.)

Initially, we considered maintaining the information needed to map <group, rank> pairs to <node, index> pairs in the Q-Kernel. However, we were concerned that this approach would not scale, especially when we considered the possibility of inter-group communication. To avoid these problems, we require that the application maintain the information needed to implement this mapping. To support application programs, the PCTs construct a map that identifies the <node, index> for every process in a process group. This map is made available to a process whenever the process group is added to the group control list for the process.

From the perspective of the Q-Kernel, a *process identifier* consists of a group identifier, a physical node identifier, and a process index. The node identifier and process index are sufficient to identify an arbitrary application or server process in the MP system. The Q-Kernel uses process group identifiers to enforce the inter-application communication policy described earlier.

To send a message, an application process first constructs a process identifier for the destination process and invokes the *send user message* entry for the Q-Kernel. The Q-Kernel checks that the group identifier in the destination is in the group control list for the sending process. If this test succeeds, the Q-Kernel transmits the message to the processor node specified in the process identifier. Figure 5 illustrates the mappings and checks performed when an application process sends a message to another application process.

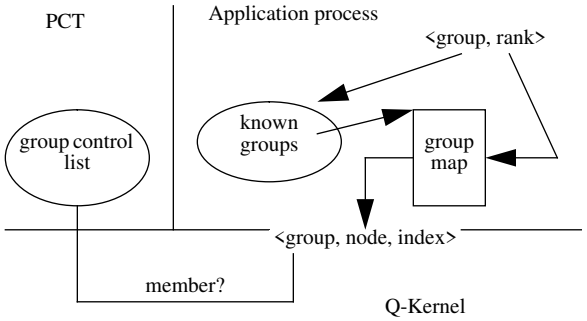

FIGURE 5. Sending an application message

The sending Q-Kernel only certifies that the group specified in a send operation is in the application's group control list. An application process could still specify an invalid <node, index> pair, i.e., a pair that is not in the specified group. To detect this abuse, the Q-Kernel on the receiving node confirms that the group identifier specified by the sending process matches the group identifier of the destination process. If this test fails, the Q-Kernel on the destination processor node rejects the message and discards the contents of the message.

Because final validation of message transmission rest with the destination Q-Kernel, malicious application processes can introduce (invalid) contention in the communication network. However, these processes cannot violate the basic communication control policy.

4.3 Server Communication

When a process group identifier is added to the group control list for an application process, the application process can send messages to any portal on any process in the process group. While this degree of flexibility is useful in many circumstances, it is certainly inappropriate for servers—process groups that provide controlled access to shared resources. Server processes must be able to control which processes and portals an application process can use. Otherwise a malicious process in one process group can interfere with the requests of another process group.

To communicate with a server process, an application process must first open a connection to the server. This connection is established through the PCT on the application's processor node. In this context, the PCT acts as a *name server*, locating an acceptable server. Once the PCT has identified a server, it sends a connection request to the server process. (Because the PCT maintains the group control lists for all processes on the local node,

including itself, it can easily send messages to any server or application process.) The connection request message includes any information that the server process needs to identify the user associated with the application process initiating the connection. In this context, the PCT acts as an *authentication server* for the server process.

If the server process accepts the connection request, it responds with a portal index to be used by the application process in subsequent communications. (The PCT on the server's node must send this response, since server processes cannot send messages directly to PCTs on other nodes.) When it receives a connection acceptance message, the application's PCT arranges to have the application's process group identifier added to the group control list for the server process. Then the application's PCT constructs a process identifier for the server process and records this information along with the portal index in a *server connection descriptor*. The PCT completes the application's connection request by returning a *server handle* that the application process can use to identify the server connection descriptor.

After establishing a connection to a server process, an application process can use the *send server message* entry for the Q-Kernel to send messages to the server process.To use this entry, the application supplies a server handle along with the message to be transmitted. The Q-Kernel uses the server handle supplied by the application process to determine the process identifier and portal index for the destination process. Note, the Q-Kernel does not consult the group control list when sending a server message.

5 The Quintessential Kernel (Q-Kernel)

The Q-Kernel is the most trusted level in the PUMA architecture. This is the only level in PUMA that has direct access to the address mapping and communication hardware. The Q-Kernel treats the communication network as a trusted and reliable resource. When a message arrives, the Q-Kernel assumes the communication accurately reflects a message sent by a Q-Kernel running on another node. The Q-Kernel does not need to authenticate the source of the message or validate the contents of the message.

5.1 Data Structures

The Q-Kernel maintains two data structures: the process *context table* and the *outgoing message queue*. The Q-Kernel uses the process context table to switch between protection/privilege domains. In addition, entries in the context table have references to the per process data structures maintained by the PCT, e.g., the process control blocks.

The first entry in the context table is associated with the PCT. This entry is initialized when the Q-Kernel is loaded. The remaining entries are established by the PCT whenever it loads an application or server process. The number of entries in the context table, and hence, the number of processes per processor node, is fixed when the Q-Kernel is loaded.

The outgoing message queue has an entry for every message that has been registered in the Q-Kernel but not transmitted. This may include messages from the PCT, messages from application processes, messages from server processes, and messages generated in response to messages sent to read memory portals.

To maintain a fixed number of entries in the outgoing message queue, the Q-Kernel establishes an upper bound on the number of entries that any process can have in the queue at any time. If a process exceeds this upper bound, the Q-Kernel rejects transmission requests from the application (including responses to its read memory portals) until the application has more queue entries available.

Note that the Q-Kernel does not block application processes. However, an application process can arrange to have the PCT block its execution and have the Q-Kernel generate a signal when there are queue entries available.

5.2 Entry Points

The Q-Kernel can be activated by a user level call (a Q-Kernel entry point), an exception (e.g., divide by zero), an interrupt associated with the communication hardware, or a timer interrupt. We begin by considering the user-level entry provided by the Q-Kernel.

The Q-Kernel provides seven entry points. Two are associated with message transmission (send_user_msg and send_server_msg). The third is used to restore an execution context (run_context). The fourth is used to suspend the execution of an application process (quit_quantum).The fifth and sixth are used to establish protection/privilege domains for application processes (create_context and extend_context). The seventh is used to establish the execution context for the PCT (set_PCT).

Three of the Q-Kernel entry points (create_context, extend_context, and set_PCT) are restricted to the PCT, and cannot be successfully invoked by an application or server process. Two other entry points (send_server_msg and quit_quantum) are only useful for application and server processes. The two remaining entry points (send_user_msg and run_context) can be used by application processes as well as the PCT.

Figure 6 summarizes the Q-Kernel entry points. We have already discussed the activities that the kernel performs in response to a call to the *send_user_msg* and

send_server_msg entries. In the remainder of this section, we consider the activities that the Q-Kernel performs in response to the remaining entries.

```
send_user_msg( void *buf, size_t len, int tag,
        proc_id dest, portal_id portal,
        int offset, int *flag );

send_server_msg( void *buf, size_t len, int *flag,
        descriptor desc );

run_context( int context_num,
        saved_state *proc_state,
        int time_quantum );

quit_quantum( void *mail_box );

create_context( cntx_index context );

extend_context( cntxt_index context,
        address_map map );

set_PCT( address thread_start );
```

FIGURE 6. Prototypes for the Q-Kernel Entry Points

The *run_context* entry restores an execution context. This entry is used by the PCT to resume the execution of an application process, i.e., to dispatch the process. Application processes can also use the run_context entry to resume execution of a saved context after handling a signal. (In many machines, the application process will be able to restore a saved context without invoking the kernel; however, in some machines an application process cannot easily restore the entire processor status word.)

For the most part, the Q-Kernel does not distinguish between the PCT and an application process when it is handling a *run_context* invocation. However, it does allow the PCT to establish a time quantum for the execution of an application process. If this time quantum expires, the application process will be interrupted and the Q-Kernel will transfer control back to the PCT.

An application process can use the *quit_quantum* entry to suspend its activities. This capability can be used to wait until a message arrives and, as we will discuss, to request a service from the PCT. When an application process invokes the quit_quantum entry point, the Q-Kernel saves the context of the process and runs the PCT.

The last three entry points are restricted to the PCT and cannot be called by application or server processes. Two of these entries, *create_context* and *extend_context*, are used to establish the address space for an application or server process. The PCT calls the create_context entry to create an initial (empty) address map for an application process. After creating the initial map, the PCT can extend the application's map by adding portions of its memory to the application's map. Whenever the PCT extends the address map for an application context, the Q-Kernel validates that the memory used in extending the map is owned by the PCT (and not the Q-Kernel).

The final entry point, *set_PCT*, is used to establish the restart address for the PCT. When PUMA is first loaded, the Q-Kernel builds a complete execution context for the PCT and transfers control to the initial start address for the PCT. After initializing its data structures, the PCT registers its continuation address with the Q-Kernel using the set_PCT entry. Subsequent entries into the PCT use the continuation address instead of the initial start address.

5.3 Exceptions

When an application process encounters an exception during its execution (e.g., divide by zero, address fault, etc.), the Q-Kernel records the pertinent information, sets a signal bit in the PCB (maintained by the PCT) and invokes the PCT. The PCT can then handle the exception by terminating the process or transferring control to the application's signal handler.

5.4 Communication Interrupts

The Q-Kernel handles two types of communication interrupts: transmit complete and message arrival. When the Q-Kernel determines that an outgoing transmission was completed, it notifies the sending process that the transmission has been completed (by incrementing the flag and setting a signal bit in the sender's PCB). The Q-Kernel also notes that the sending process has a free queue entry and initiates the transmission of the next outgoing message, if there is one.

When the Q-Kernel receives an interrupt for an arriving message, it uses the process index in the incoming message as an index into its context table. This entry identifies the PCB for the destination process. The PCB identifies the portal table which, in turn, identifies the destination portal. Before it initiates the message receipt, the Q-Kernel verifies that the group identifier in the message matches the group identifier of the receiving process and that the message will not violate the domain constraints for the destination process portal. When the message body has been received, the Q-Kernel updates the destination portal's descriptor and sets a bit in the array of pending signals for the process.

5.5 Timer Interrupts

Whenever the Q-Kernel receives a timer interrupt, it means that the current application has exceeded its time quantum. In this case, the Q-Kernel simply runs the PCT. The PCT

6 The Process Control Thread (PCT)

As we have discussed, the PCT provides many of the services typically associated with an operating system, e.g., process creation, process scheduling, and server connection. The PCT is run whenever an application process completes its allocated quantum, either because of a timer interrupt or because the application invoked the quit_quantum entry. It is worth noting that the PCT is not run after every entry into the Q-Kernel.

6.1 An Overview of the PCT

The PCT begins its execution by checking for requests from the application process whose execution was suspended. When an application process needs a PCT service, it prepares a PCT request and places the request in a mailbox structure. In most cases, an application process will prepare a single request, place the request in its mailbox, and voluntarily relinquish the CPU using the quit_quantum entry into the Q-Kernel. An application can prepare several requests and link them into its mailbox. The PCT only examines the application's mailbox after the application suspended its activities using a quit_quantum request. In particular, the PCT will not examine an application's mailbox if the application's execution was suspended due to a timer interrupt. This strategy avoids race conditions associated with the mailbox structure.

After checking the mailbox of the suspended process, the PCT examines its portals for messages from other PCTs. PCTs send messages to other PCTs when they need information about the resources and services provided by another node. This type of information is needed when an application initiates a server connection or spawns a process.

After handling requests from other PCTs, the PCT tries to find a runable application or server process. If the PCT does not find a runable process, it enters a loop in which it responds to new requests from other PCTs and checks for a runable process. When it finds a runable process, the PCT selects one of the runable processes as the next process to be executed. The PCT always completes its activities by invoking the Q-Kernel run_context entry, thus dispatching the selected process.

6.2 Signals

Each process has a vector of pending signals. In addition, each process maintains a vector of blocked signals, a vector of ignored signals, and a signal handler function. When a process has a pending signal that is neither blocked nor ignored, the PCT notes that the process has an outstanding signal. Any process with an outstanding signal is runable.

Before the PCT runs a process, it checks to see if the process has an outstanding signal. If it has an outstanding signal, the PCT copies the saved context into the stack for the process and establishes the signal handler as the current execution context. This makes it relatively easy to stack signal handlers and passes the burden for storing stacked contexts back to the application or server process.

6.3 Blocking

To block its execution, an application or server process sets its execution state to blocked and invokes the quit_quantum entry provided by the Q-Kernel. When the PCT looks for runable processes, it notes that the process is not runable and the process will remain blocked until it receives a signal.

7 Related Work

Many researchers have recognized the need to minimize memory copies during communication[4]. Mach, for example, uses page mapping to avoid memory copies[3]. Incoming messages are held in kernel pages until requested by an application. When an application requests a message, the kernel maps the pages holding the requested message into the application's address space. This approach can avoid the costs associated with memory copies and presents the application programmer with simple IPC semantics. In developing PUMA, we had two concerns with the memory mapping approach. First, some MP machines (e.g., the nCUBE2) do not provide adequate address mapping hardware to support this approach. Second, to avoid memory copies, applications must receive messages in buffers that are aligned on page boundaries. This requirement may waste physical memory and impose a significant burden on the organization of the data structures used in an MP application.

The "active messages" approach avoids memory copies by invoking a user level message handler whenever a message arrives from the communication network[7]. Because the user level handler controls the placement of incoming messages, this approach should minimize the need to copy messages. However, because the user level message handler has direct access to the communication hardware, this approach may not be appropriate when several applications share the resources provided by an MP machine.

8 Results

In January of 1993 we completed a preliminary implementation of PUMA for the nCUBE2. This implementation achieves message throughput rates as high

as 2.17 MB/second per channel, or 98% of the channel capacity. Our current message latencies (application level to application level) are approximately 110 microseconds. With turning we expect that we can reduce the latency by 25% or more.

In May, we began porting PUMA to the Intel Paragon. We completed an initial implementation in August of 1993. This implementation achieves message throughput rates as high as 165 MB/second per channel, or 94% of the channel capacity. Every processor node on the Paragon provides two i860 processors. Message latencies in our Paragon implementation are approximately 50 microseconds when we use a single processor per node and as low as 30 microseconds when we use the second processor to handle all message traffic.

9 Acknowledgments

We are grateful to the members of the original SUNMOS team. Without their efforts, we would not have been in a position to undertake the development of PUMA. We would also like to acknowledge the efforts of the other members of the PUMA team. Including: Kenneth Ingham, Clint Kaul, Michael Levenhagen, and Francisco Reverbel.

10 References

1 M.J. Accetta, et.al. Mach: A New Kernel Foundation for UNIX Development. In *Proceedings of the Summer 1986 USENIX Conference*, pp. 93-113, July 1986.

2 J. Alemany and E.W. Felten. Performance Issues in Non-blocking Synchronization on Shared-memory Multiprocessors. In *Proceedings of the Eleventh Annual ACM Symposium on Principles of Distributed Computing*, pp. 125-134, August 1992.

3 J.S. Barrera III. A Fast Mach Network IPC Implementation. In *Proceedings of the USENIX Mach Symposium*, November 1991.

4 C.M. Burns, R.H. Kuhn, and E.J. Werme. Low Copy Message Passing on the Alliant CAMPUS/800. In *Proceedings of Supercomputing'92*, pp. 760-769, November 1992.

5 D.R. Cheriton. The V Kernel: A Software Base for Distributed Systems. *IEEE Software*, 1(2):19-42, April 1984.

6 D.R. Cheriton. The V Distributed System. *Communications of the ACM*, March 1988.

7 T. von Eicken, D.E. Culler, S.C. Goldstein, and K.E. Schauser. Active Messages: a Mechanism for Integrated Communication and Computation. In *Proceedings of the 19th International Symposium on Computer Architecture*, May 1992.

8 A.B. Maccabe and S.R. Wheat. The PUMA Architecture: An Overview. Sandia National Laboratories Technical Report SAND93-1372.

9 R. van Renesse and A.S. Tanenbaum. Short Overview of Amoeba. In *Proceedings of the USENIX Workshop on Micro-kernels and Other Kernel Architectures*, pp. 1-10, April 1992.

10 M. Rozier, et.al. The Chorus Distributed Operating System. *Computing Systems*, 1(4), 1988.

11 R. Zajcew, et.al. An OSF/1 UNIX for Massively Parallel Multicomputers. In *Proceedings of the Winter 1993 USENIX Conference*, pp. 449-468, January 1993.