UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE

(2)

RT DOCUMENTATION PAGE

AD-A213 871

1a. R	1b. RESTRICTIVE MARKINGS
2a. S	3. DISTRIBUTION/AVAILABILITY OF REPORT Approved for public release; distribution unlimited.
2b. D	
4. PERFORMING ORGANIZATION REPORT NUMBER(S) NMSU -ECE - 89-007A	5. MONITORING ORGANIZATION REPORT NUMBER(S) ARO 25173.20-EL
6a. NAME OF PERFORMING ORGANIZATION New Mexico State University 6b. OFFICE SYMBOL (If applicable) PARL	7a. NAME OF MONITORING ORGANIZATION U. S. Army Research Office
6c. ADDRESS (City, State, and ZIP Code) Las Cruces, NM 88003	7b. ADDRESS (City, State, and ZIP Code) P. O. Box 12211 Research Triangle Park, NC 27709-2211
8a. NAME OF FUNDING/SPONSORING ORGANIZATION U. S. Army Research Office 8b. OFFICE SYMBOL (If applicable)	9. PROCUREMENT INSTRUMENT IDENTIFICATION NUMBER DAAL03-87-K-0106
8c. ADDRESS (City, State, and ZIP Code) P. O. Box 12211 Research Triangle Park, NC 27709-2211	10. SOURCE OF FUNDING NUMBERS PROGRAM ELEMENT NO. / PROJECT NO. / TASK NO. / WORK UNIT ACCESSION NO.

11. TITLE (Include Security Classification)

(U) ARGOS - A Research GMMP Operating System: Overview and Interfaces (U)

12. PERSONAL AUTHOR(S) Eric E. Johnson

13a. TYPE OF REPORT	13b. TIME COVERED	14. DATE OF REPORT (Year, Month, Day)	15. PAGE COUNT
Research	FROM ______ TO ______	89 August	77

16. SUPPLEMENTARY NOTATION The view, opinions and/or findings contained in this report are those of the author(s) and should not be construed as an official Department of the Army position, policy, or decision, unless so designated by other documentation.

17. COSATI CODES			18. SUBJECT TERMS (Continue on reverse if necessary and identify by block number)
FIELD	GROUP	SUB-GROUP	multiprocessor, operating system, message passing, GMMP architecture

19. ABSTRACT (Continue on reverse if necessary and identify by block number)

This report presents an overview of A Research GMMP Operating System (ARGOS) developed at the NMSU Parallel Architecture Research Laboratory for the prototype Virtual Port Memory multiprocessor. ARGOS is an initial attempt to realize the reliability and performance benefits expected of GMMP multiprocessors.

After a review of GMMP architectures and the Virtual Port Memory machine, the structure and philosophy of ARGOS are presented, followed by interface-level descriptions of each of the system modules. Later reports will present detailed individual discussions of the implementations of these modules.

20. DISTRIBUTION/AVAILABILITY OF ABSTRACT ☐ UNCLASSIFIED/UNLIMITED ☐ SAME AS RPT. ☐ DTIC USERS	21. ABSTRACT SECURITY CLASSIFICATION Unclassified
22a. NAME OF RESPONSIBLE INDIVIDUAL	22b. TELEPHONE (Include Area Code) / 22c. OFFICE SYMBOL

DD FORM 1473, 84 MAR

83 APR edition may be used until exhausted.
All other editions are obsolete.

SECURITY CLASSIFICATION OF THIS PAGE

UNCLASSIFIED

89 226

ARGOS — A RESEARCH GMMP OPERATING SYSTEM: OVERVIEW AND INTERFACES

ERIC E. JOHNSON

NMSU-ECE-89-007A   AUGUST 1989

SUPERSEDES NMSU-ECE-89-007, APRIL 1989

A-1

This work was supported by the U.S. Army Research Office under contract # DAAL03-87-K-0106, and by a grant from the AT&T Foundation.

TABLE OF CONTENTS

TABLE OF CONTENTS

TABLE OF CONTENTS

ARGOS — A RESEARCH GMMP OPERATING SYSTEM: OVERVIEW AND INTERFACES

ERIC E. JOHNSON

ABSTRACT

This report presents an overview of A Research GMMP Operating System (ARGOS) developed at the NMSU Parallel Architecture Research Laboratory for the prototype Virtual Port Memory multiprocessor. ARGOS is an initial attempt to realize the reliability and performance benefits expected of GMMP multiprocessors.

After a review of GMMP architectures and the Virtual Port Memory machine, the structure and philosophy of ARGOS are presented, followed by interface-level descriptions of each of the system modules. Later reports will present detailed individual discussions of the implementations of these modules.

1. INTRODUCTION

1.1 GMMP Architectures

GMMP multiprocessor architectures [3, 6] are global memory architectures optimized for the message-passing model of computation, in which process access environments are strictly isolated. Such architectures attempt to realize both the high performance and relatively simple code and data partitioning of shared memory multiprocessors, and the reliability and verifiability claimed for the message-passing model. A distinct advantage enjoyed by GMMP multiprocessors over traditional "shared memory" (GMSV) machines is the freedom to employ large caches to reduce processor-memory traffic and average memory access times without a necessity to maintain cache consistency, due to the isolation of process access environments [6].

The programmer's model of a GMMP multiprocessor is shown in Figure 1.1. Semantically, programming a GMMP machine is similar to programming a distributed-memory message passing, or DMMP, machine (e.g., a hypercube) except that achieving high performance does not require explicit duplication of code and data, a common strategy for DMMP machines despite the resulting inefficient use of limited physical storage at the processing elements.

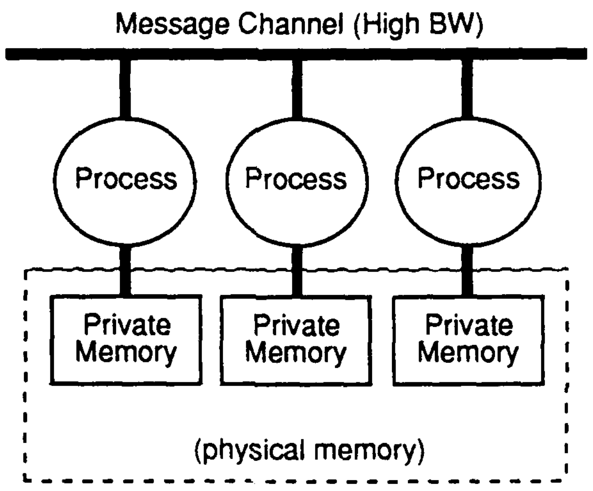

Figure 1.1: Virtual Port Memory Architecture (Programmer's Model)

Global memory machines (GMSV and GMMP) may efficiently perform message passing via re-mapping pages of virtual memory. Using this technique, data may be "moved" from one process access environment to another by merely duplicating page table entries and sending the virtual address of the data (a pointer or capability), rather than by copying data values over an interprocessor or processor-memory channel. Because arbitrarily large data structures may be re-mapped in microseconds, such virtual message passing appears to have very high bandwidth and low, fixed latency. (To maintain strict isolation of process access environments, pages which become physically shared due to this re-mapping must be flagged for copy-on-write protection.)

1.2 The Virtual Port Memory Machine

The Virtual Port Memory machine under construction at New Mexico State University [4, 5, 7] is intended to evaluate the potential of such GMMP architectures to combine the best attributes of both the shared memory and the message passing paradigms while avoiding many of their drawbacks. The features included in the virtual port memory architecture were chosen specifically to explore this class of multiprocessor architectures, and to try to realize their inherent benefits.

In brief, a virtual port memory multiprocessor architecture provides a hardware structure, which includes a global memory with integral address translation and page copy hardware and a broadcast message network, to support a GMMP virtual machine, which provides each process of a computation or concurrent system with an isolated access environment within the system virtual memory and *by-value* message passing.

1.2.1 Conceptual Model

The Virtual Port Memory implementation of the programmer's model of Figure 1.1 is shown conceptually in Figure 1.2. The isolated access environments of the programmer's model are implemented by address mapping, performed by the address translator in the global memory controller (supported by access checking at the PE when required). The ports shown in Figure 1.2 correspond (conceptually) to process capability lists; these "virtual ports" to memory give the architecture its name.

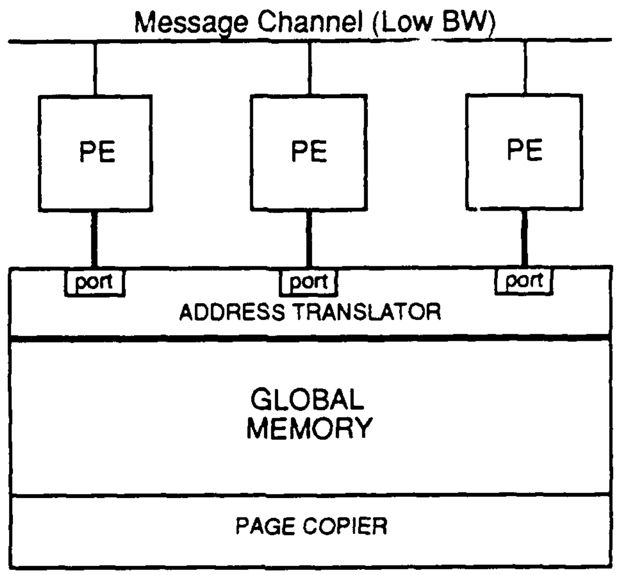

Figure 1.2: Virtual Port Memory Architecture (Conceptual Model)

The high-bandwidth interprocess message channel employed by the programmer to pass data by value (Figure 1.1) is implemented as a much lower-bandwidth interprocessor message channel (Figure 1.2); as noted previously, this physical channel generally carries *pointers* to (or capabilities for) data rather than data *values*, and therefore requires relatively little bandwidth to support a large "virtual" message passing bandwidth.

A noteworthy feature of the VPM architecture is the inclusion of "page copy" hardware in the global memory controller, which performs memory-to-memory data transfers at high speed without consuming bandwidth on the processor-memory network. This unit, which is accessed by messages as a system process, is primarily used to perform the copying of pages incurring copy-on-write faults, but is available for general use as well. Due to the sequential nature of its accesses, this unit can take full advantage of the especially efficient access modes available in current dynamic memory technology.

1.2.2 Prototype Implementation

The NMSU prototype Virtual Port Memory machine is shown in Figure 1.3. It is bus-based, with a dedicated interprocessor channel, the Interprocess Message Bus (IMB), and a pipelined processor-memory channel consisting of a Data Transfer Bus (DTB) and a separate Transaction Request Bus (TRB).

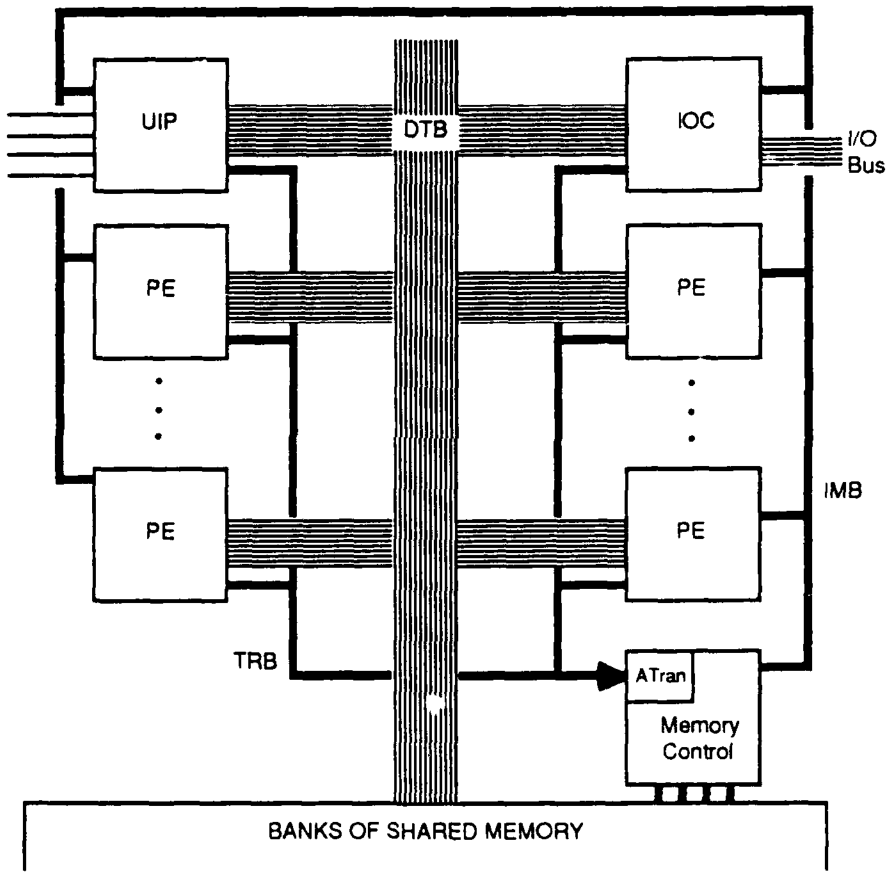

Figure 1.3: Bus-Based VPM Prototype

Requests for memory cycles are sent from processors to the memory controller over the TRB. The memory controller translates the virtual addresses contained in transaction requests to physical addresses, and queues each translated request for subsequent action by the appropriate bank of the global memory. When a request reaches the head of the queue in a memory bank, the cycle controller for that bank of memory runs the requested cycle and coordinates the transfer of data between the processor and the memory over the DTB.

1.2.2.1 Processors

The processors shown in Figure 1.3 are of three types: processing elements (PE), which execute application and system programs; user interface processors (UIP), which communicate with user workstations and execute shell (command interpreter) processes; and I/O controllers (IOC), which manage other I/O transfers. All three types of processor interact with each other via the IMB, and access the global memory using the TRB and DTB.

The major components of a prototype general purpose processing element are shown in Figure 1.4. Each PE contains a 16 MHz 68020 microprocessor with a 68882 floating point co-processor, and a two-way set associative data/instruction cache with a four byte line size and a total capacity of 256 KBytes. (This short line size seems adequate for image processing, but will certainly need to be lengthened for other applications; the second generation of this prototype machine will employ a 128-bit DTB and a cache line size of 16 or 32 bytes.)

The cache controller is a micro-coded state machine; when activated by a cache miss, it generates transaction request(s) to the global memory and performs data transfers over the DTB as directed by DTB control signals generated at the global memory. The message unit consists of send and receive FIFOs under state machine control. Access checking is performed by a lookup table in RAM, indexed by segment number and a local PE process number.

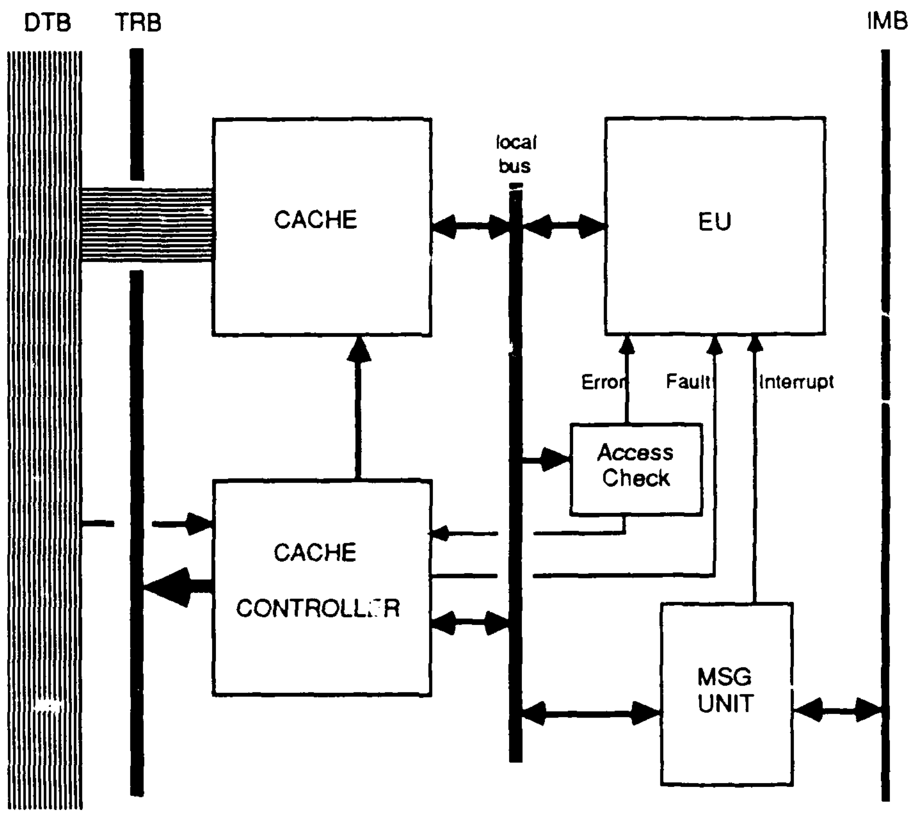

Figure 1.4. Prototype VPM Processing Element

1.2.2.2 Global Memory

The central address translator (ATran in Figure 1.3) could become a bottleneck in this architecture if its worst-case translation time exceeds the inter-arrival time of transaction requests; consequently, it is designed to process transaction requests in a fixed time matched to the speed of the TRB. Conceptually, the ATran contains a large cache of <system virtual address>|<physical address> pairs, and translates requests which "hit" this cache in one TRB clock cycle. Transaction requests resulting in a miss are "kicked out" of the memory controller and returned to their source with a fault indication, and consequently do not congest the ATran.

In addition to performing virtual-to-physical address translation, the memory controller also validates the operation requested by each transaction request, using an access permission field from the ATran cache entry. Entries in this table contain access rights granted to *any* transaction request which reaches the memory controller; restriction of access rights on a process-by-process basis is performed at the processing elements. Access violations detected by the memory controller result in a fault which invokes the operating system (the Local Agent at the PE of the offending process).

The banks of global memory are interleaved to match the memory system bandwidth to that of the DTB. A cycle controller within each bank time-multiplexes access to its DRAM array among transaction requests, page copy cycles, and refresh cycles, with dynamic memory refresh requests generated locally within each bank. The cycle controllers employ an arbiter to share access to the DTB.

The main memory (Figure 1.5) contains four interleaved banks of DRAM, each storing 8 Mbytes (to be upgraded to 32 MBytes each). The system virtual address space for this machine contains 256 segments of up to 1 MByte each, oriented to our image processing applications. With such a small virtual address space, we chose to implement the address translator as a full look-up table in fast SRAM, rather than the more complex (and less predictable) address translation cache which is needed for a larger virtual address space. (A VLSI address translation cache is under development for use in a larger second generation machine, which is intended for more general applications.) The operating system (specifically the Pager) updates this address translation RAM using an "ATran Update" transaction request.

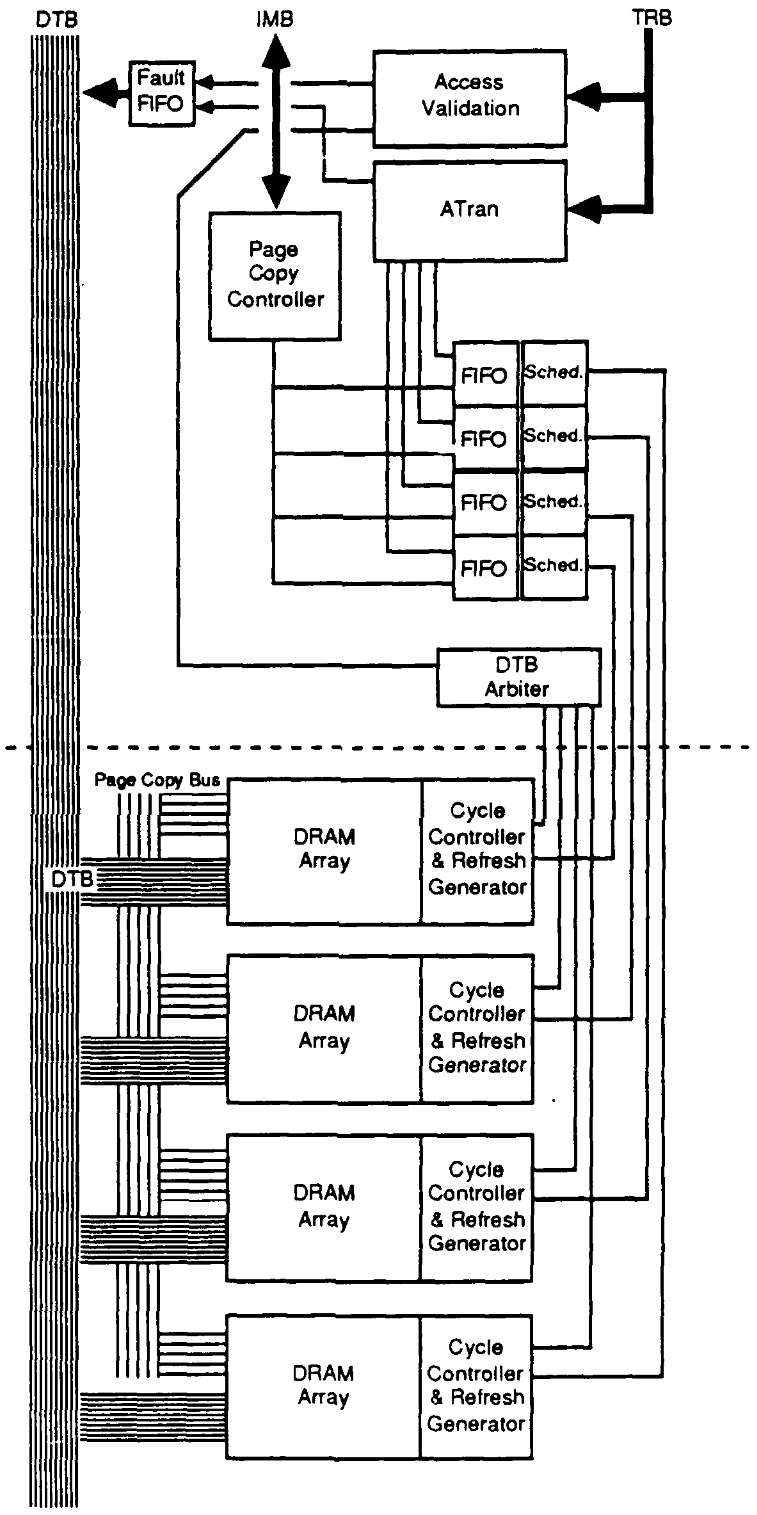

Figure 1.5: Prototype Global Memory and Controller

1.3 Virtual Memory Structure

ARGOS supports a somewhat unusual model of virtual memory, particularly so in light of its intended use in a GMMP environment with strictly isolated process access spaces. Rather than the more usual per-process virtual memories, which sometimes map into a single system-wide virtual memory before finally mapping to physical memory, the ARGOS model of virtual memory consists of a single, segmented system virtual address space: processes directly generate system virtual addresses without an intermediate step of translation, with process isolation enforced by access checking hardware at the processors.

This implementation of virtual memory has some interesting ramifications.

1) Because system virtual addresses (i.e., system segment numbers or SSNs) are unknown before execution time, no absolute addresses may be embedded in code segments; all code must employ indirect, PC relative, or symbolic addressing, with symbolic reference resolution taking place at execution time.

2) Maintenance of translation tables is considerably simplified, compared to the per-process virtual memory approach, because there is only one set of translation tables, and only one stage of translation, all of which may be managed by a single ARGOS process (the Pager).

3) Because address translation takes place at a system-wide level, only one address translation cache (a.k.a. translation look-aside buffer or TLB) is needed in the system (the ATran in our VPM machine), eliminating problems of inter-TLB consistency which arise when each processor has a private TLB.

Because ARGOS segments are essentially files mapped into virtual memory, a segment may be named; this name is used as a path name to place the segment in the file system; transient segments, such as execution segments, may remain nameless.

1.4 Operating System Requirements

The functions to be performed by ARGOS are dictated by the differences between the underlying hardware constituting the "real" machine, and the desired user and process interface and performance characteristics of the "virtual" machine which ARGOS is to construct upon this underlying machine. The preceding sections have presented a general description of the virtual machine with its message-passing programmer's model and segmented virtual memory, and the Virtual Port Memory prototype upon which ARGOS will be implemented. Before proceeding to the detailed description of ARGOS contained in the rest of this report, it is useful here to summarize the general requirements for ARGOS:

1) High-performance message passing, making effective use of the available global memory.
2) Management of the segmented-paged virtual memory: allocation of segments of virtual memory and page frames of physical memory; address translation and maintenance of the translation tables and hardware address translator; paging.
3) Process isolation and management: verification of access rights; copy-on-write protection and fault handling; exception handling.
4) Processor assignment and scheduling: load balancing; time-sharing; support for real time applications.
5) File management: creation and maintenance of tree-structured file system(s); directory searching, including permission checking.
6) I/O management: allocation of I/O resources; control of I/O traffic.
7) User interface: facilities to create, launch, observe, control, and debug parallel applications.

2. ARGOS

2.1 ARGOS Structure

ARGOS consists of a message-passing kernel which is replicated at each PE, and a group of system processes each of which manages a distinct class of resources; the initial implementation includes the following modules (see Figure 2.1):

1) local agents at each processing element which pass messages, schedule processes on their processors, and handle exceptions,
2) a file manager (FileMan), which manages the file system(s), searches directories, and checks access permissions for files,
3) an object manager (ObMan), which allocates and manages segments of the system virtual memory,
4) a Pager which implements demand paging and manages virtual to physical address mapping and the ATran,
5) a page frame manager (PFM), which allocates and recovers physical memory,
6) device managers such as DiskMan, which run on dedicated I/O controllers, and manage their respective I/O devices,
7) a PE allocator (PEA), which assigns processes to processing elements,
8) and user interface processes (UIPs), which run on user interface processors, and execute, or launch processes to execute, user commands.

These processes maintain their resource management data structures in strict isolation from one another, and interact only by passing messages. This eliminates many of the low-level concurrency-related problems often faced by operating system designers, although certainly leaving a number of issues still to be addressed by careful design.

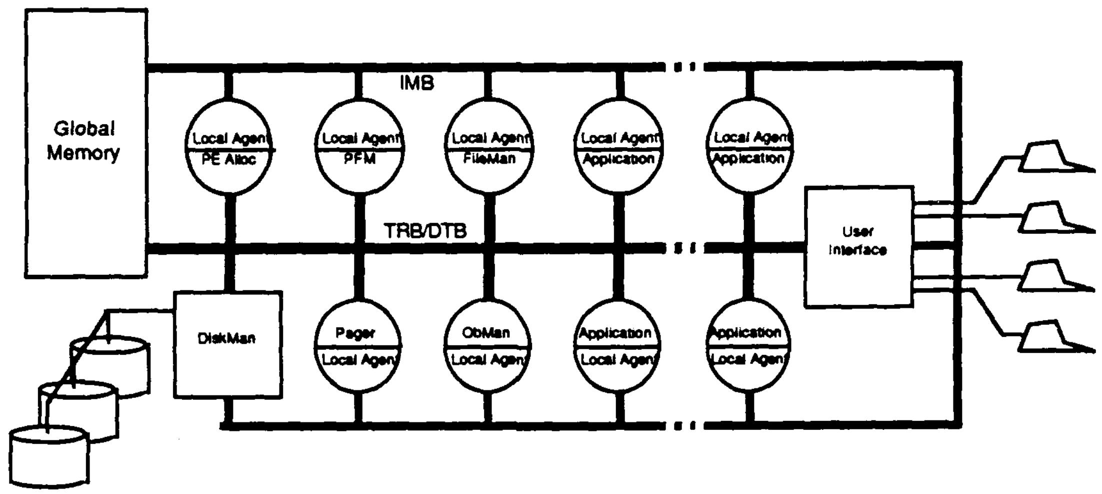

Figure 2.1: ARGOS Structure

2.2 Processes

An ARGOS process (system or application) may be viewed as a thread of control within an isolated access environment, or equivalently as a time sequence of snapshots describing the evolution of the alterable segments of that environment. A process access environment consists of at least two types of segments: code segments (static), and segments which record the execution of the process (dynamic). In particular, a process is created with access to at least two segments: its initial code segment, from which it will begin to fetch and execute instructions, and its execution segment (ExSeg) which contains most of the data structures which are peculiar to that process; these include its process structure (similar to Unix u area), kernel and user stacks, and a heap (Figure 2.2). A process may inherit or acquire access rights to additional segments; these rights are noted in its process structure.

The process structure contains fields for, among other things, process state and scheduling priority, process IDs of self and parent, a table of message queues (see next section) and a mask for sleeping on a set of message queues, a list of accessible segments and access modes (similar to a capability list), the segment number of the present working directory, a table of file descriptors for open files, a table of signal actions, sleep and alarm timers, real and effective user IDs (uids) and group IDs (gids), the user's login name, and the message queue ID of the controlling UIP.

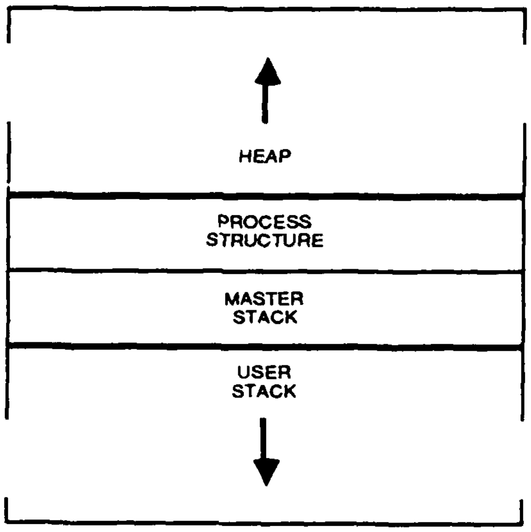

Figure 2.2: Execution Segment

2.3 Message Passing

Interprocess communication and synchronization in ARGOS occur via messages sent by one process to a message queue owned by another process. The process which creates a message queue permanently owns the queue, and communicates its willingness to receive messages through this queue by sending a message queue identifier (MQID) to potential message originators. MQIDs, used by the message-passing hardware to route and receive messages, are formed by concatenating a short index number to the system segment number (SSN) of the execution segment of the owning process: <mqid> ::= <ExSeg SSN> <index>.

Every ARGOS process is created with two message queues: one for use in process management, called its kernel message queue (index = 0), and one for receiving replies to locally-generated requests, called its reply queue (index = 1). When a process is created, the returned `pid`, or process identifier, is the kernel message queue number of the new process. The message queue name space is managed by the local agents. *Link* establishment, termination, and notification of termination are handled above ARGOS, by the processes desiring such services.

The message-passing hardware in the Virtual Port Memory machine uses the following format for messages:

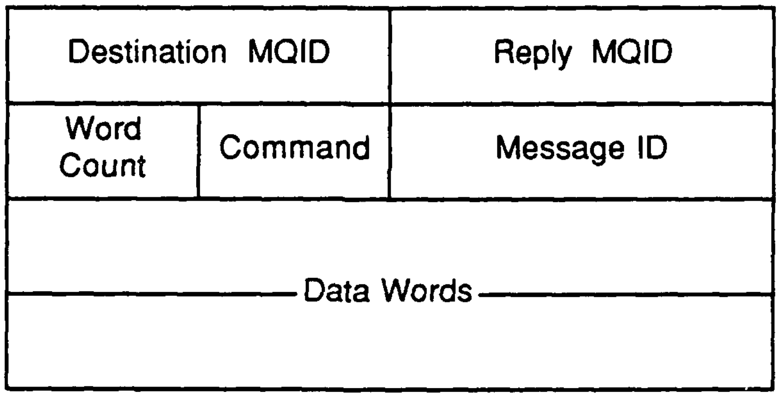

Figure 2.3: ARGOS Message Format

The Destination MQID is used by message receivers to identify messages destined for local processes, and by local agents to locate the correct data structures to receive message contents. The Reply MQID is used for replies to service requests; a reply always echoes the Message ID of the request, so that the requesting process can correctly interpret the reply. The Word Count field indicates the number of 32-bit data words which follow the two-word header; a count of zero is not unusual. The Command field, used only by system processes, encodes brief requests within the message header. Each message type listed in the following chapters is distinguished by the value of its Command field; for example ACK is encoded as 0, NAK as -1, and so on. User process messages have a Command field of zero, and pass all information in data words.

2.4 File System

The ARGOS file system structure is quite similar to that of UNIX [1]. A file system is composed of a tree structure of directories, and one file system may be mounted as a sub-tree of another file system. As in UNIX, a single physical disk may contain multiple file systems, each residing in a distinct partition of the disk.

File management data are maintained in file headers, which are analogous to UNIX inodes. The information in each file header is partitioned into two records: a *file access record*, which is managed by the FileMan, and the *file index*, which is managed by the Pager. File headers are stored as the final bytes of *file header blocks* near the beginning of each disk partition. Each file header block also contains the final "fragment" of file data; this arrangement is expected to reduce disk arm movement somewhat, especially for small files which will fit entirely within the file header blocks.

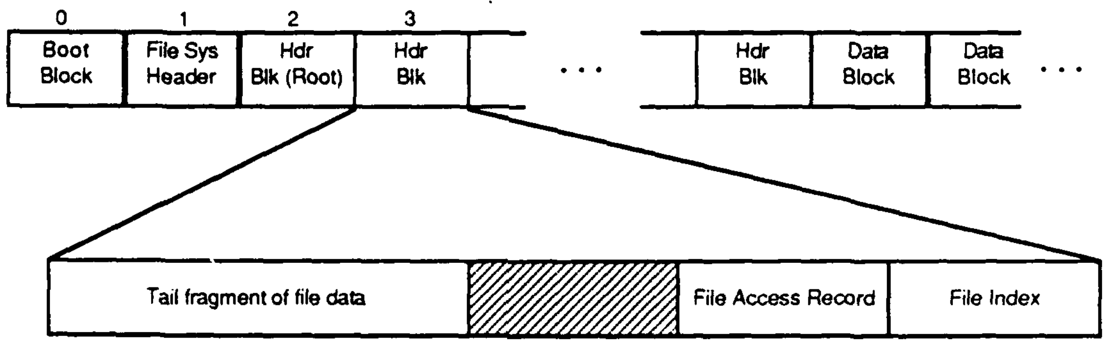

Figure 2.4: ARGOS File System Structure

The file access record contains fields for owner uid and gid, access permissions (read, write, and execute/search for owner, group, and world), file type, access times, link count, and file size (in bytes). The file index contains block numbers for 24 direct blocks, one indirect block, one doubly-indirect block, and one triply-indirect block.

In addition to managing space on its disks, and performing reads and writes to them, each DiskManager manipulates the header blocks of the file systems in its charge. The FileManager reads and writes file access records indirectly via messages to DiskManagers; the Pager likewise uses messages to access and update file indices and to request disk I/O. Messages to the Disk Managers are addressed to message queues whose mqids are also used as *file system numbers* (fsn); each file system has a distinct message queue. The parameters in these messages are block numbers, when direct I/O is requested, or *file numbers* (fn) when access to a header block is required; the disk manager computes the block number of a header block as fn + 2 (skipping over the boot block and file system header). File number 0 is always the root directory of the file system.

Under ARGOS, files are accessed by mapping them into segments; unless a file is to be executed, this mapping is direct: byte 0 of a file appears as byte 0 of a segment, and so on. Executable files, however, may contain code and data which must be placed in more than one segment for execution; loading of executable files is performed by local agents in response to the `exec()` system call.

2.5 Structure of Module Descriptions

The remaining chapters of this report each discuss one of the modules composing ARGOS, beginning with the Local Agency, and continuing through all of the system processes. For each module, a brief description of its function is given first, followed by *lists of the service request messages it recognizes*, and *of the service request* messages it generates. More detailed discussions of the individual processes may be found in the reports on those processes.

Where information in this report conflicts with information in the detailed reports, the more recent version may be assumed to reflect the current implementation. When this report was written, ARGOS had, we hoped, concluded its period of *rapid* evolution, but steady change was still anticipated, as ARGOS was intended as a vehicle for operating systems research.

In the descriptions which follow, function arguments and message fields enclosed in square brackets are optional as in `exec(path, [argv])`, where `path` is required but `argv` is optional.

Many of the terms and acronyms used in these descriptions are explained in the Glossary at the end of this report.

3. LOCAL AGENCY

3.1 Function

The "kernel" of ARGOS consists of code replicated at each processor which contains routines for passing messages, scheduling the local processor, and handling hardware-detected exceptions; processes executing this code are collectively called the "local agency." The local agency at each processor comprises a Clerk (an interrupt service routine which handles incoming messages) and local agents, which are the master-mode* phases of application and system processes resident on that processor. The Clerk, as an interrupt service routine, executes in 68020 interrupt mode within the context of the interrupted process; it is not a process in its own right.

The sending of messages is accomplished directly by local agents, which gather the header and data words of messages, load them into a hardware FIFO, and write command words to a control register to send the FIFO contents over the Interprocessor Message Bus (IMB). The sending local agent doesn't release the message sending hardware until it either has successfully sent its message, perhaps after a number of retransmissions, or has exhausted its retry count and given up. Here, successful transmission requires only an indication from the hardware that message receiving hardware on some processor has acknowledged receipt of the message.

Message *receipt*, on the other hand, is completely asynchronous to process execution. The receiver hardware continuously monitors the IMB for the start of a message; when a new message begins, the destination mqid is presented to a lookup table (in static RAM) to determine if the queue belongs to a process resident on the local

* The 68020 supports user, master, and interrupt modes, each with a separate stack; the latter two "supervisor" modes permit execution of privileged instructions.

processor. If it does, the receiver copies the header and data words from the IMB into a receive FIFO, while watching for the end of the message. At the end of the message, the hardware stops loading the FIFO, sends a hardware acknowledgement, and generates an interrupt to the local processor which invokes the message receipt interrupt service routine, the Clerk.

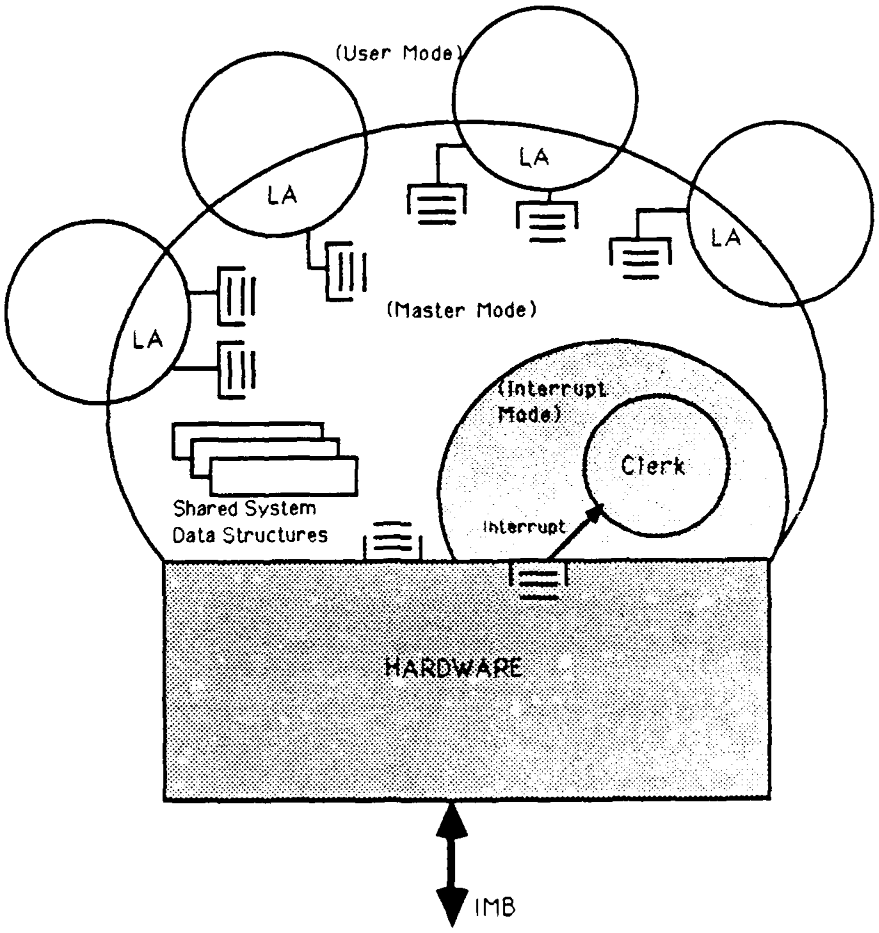

Figure 3.1: Local Agency

The Clerk examines the command field and destination mqid in each received message; the message is then placed in the appropriate process message queue in system virtual memory, unless it is addressed specifically to the Clerk, or contains a command which directs the Clerk to manipulate the addressed process (e.g., add a new segment to its access environment). The virtual address of a message queue is found as follows: the segment number of the ExSeg of the receiving process is extracted from the mqid (see §2.3); the remaining field in the mqid is then used to index into the message queue table in the process structure in that ExSeg; this table contains the addresses for the head and tail of each message queue.

Processes become Local Agents by entering master mode; this can occur only in the following ways: system calls, access faults, or program faults. Although such program faults as division by zero may optionally be handled by user-supplied routines, these routines execute in user mode; all code executed in master mode is ARGOS code.

Access faults, whether detected locally or by the global memory address translator (ATran), raise a 68020 bus error (BERR) exception, which causes the faulted process to execute the BERR handler in master mode. Unauthorized accesses to a segment are detected locally, and normally result in process termination. Faults detected at the ATran result from ATran cache misses, page faults, or unauthorized accesses to specific pages of authorized segments. Many of these faults may be repaired via messages sent to other ARGOS processes by a Local Agent, after which the faulted access may be successfully completed.

Unix-style file I/O is simulated by the local agency by mapping files into segments of virtual memory, and maintaining a file pointer to identify the next byte to be "read" or "written." Paging these mapped files results in some similarity with the Unix block buffer cache [1].

Executable files are formatted as shown in Figure 3.2. The Load Header contains flags indicating the type of processor required to execute the code in the file, and other details about the file as a whole, the number of sections composing the file, and a template for portions of the execution segment for the file, including initial contents of processor registers and of the Master and User stacks. Each Section Header contains type, size, and memory mapping parameters for each section of the file. Finally, the executable file contains the sections of code and data to be mapped into virtual memory, each aligned within the file so that it maps into memory starting at a page boundary.

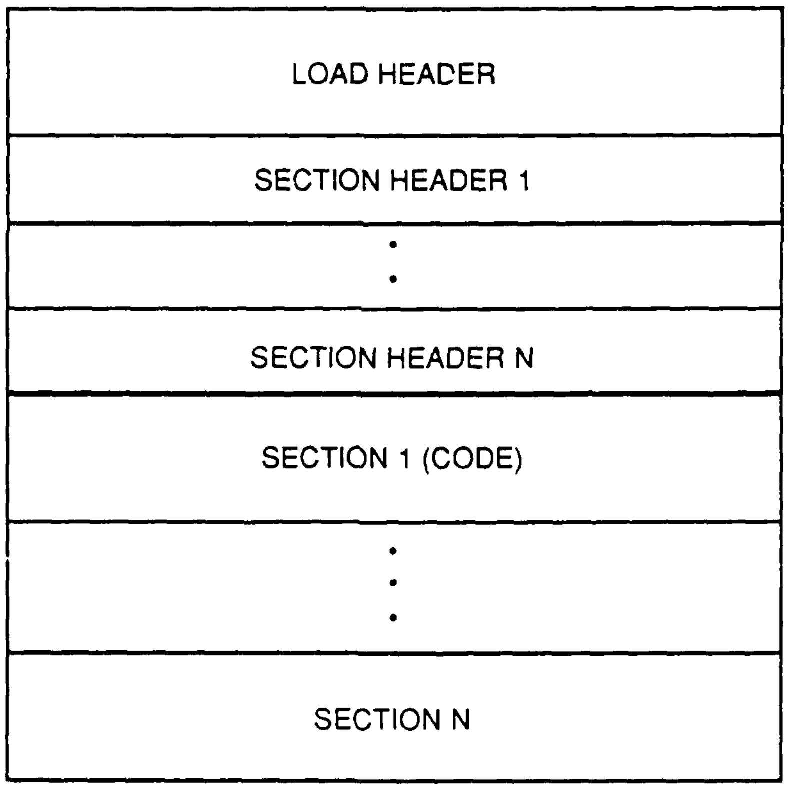

Figure 3.2: Executable File Format

In response to an `exec()` function call, the local agent allocates a nameless segment to become the ExSeg for the new process, maps the Load Header and Section Headers into the middle of this segment so that the templates for the user and master stack and the process structure are correctly located, and reads the Section Header data to determine what to do with the remainder of the file. After the various sections have all been mapped appropriately, the Section Headers are overwritten with the initial heap contents for the new process (e.g., `argv` and environment strings); after any other changes to the new process' environment have been made, the new process is entered into the local process table and becomes eligible for scheduling.

3.2 Function Calls

Interaction between resident processes and the local agency occurs via a system call interface. The functions composing this interface are the Unix-like POSIX functions [2], augmented by a set of GMMP-specific functions for inter-process communication and management of the segmented virtual memory, and a set of process control functions for real-time applications.

3.2.1 POSIX Interface

The ARGOS implementation of the POSIX interface is intended to be as complete as possible given the message-passing model. The following system calls are supported; their effects comply with the POSIX definitions, but may not completely mimic the actions assumed by Unix programmers.

3.2.1.1 Process Management Primitives

`fork([options])`	forks a new process on the same processing element; new process shares read- or execute-only segments with parent; alterable segments are marked copy-on-write; options include sole use of PE, and prefetching and locking all segments in memory; returned `pid` is the kernel message queue number of the new process.
`exec(path, [argv])`	replaces process image.
`wait([stat_loc])` `waitpid(pid, [stat_loc], [options])`	waits for status from stopped or terminated child process(es).
`_exit([status])`	terminates process.
`kill(pid, signo)`	sends a signal to a process (or process group).
`sigact(signo, act, [oact])`	sets signal action, returns old action.
`alarm(seconds)`	schedules `SIGALARM` signal.
`pause()`	suspends process until it catches a signal or is terminated.
`sleep(seconds)`	like pause, but wakes up after `seconds`.

3.2.1.2 Process Environment Primitives

`getpid()`	get own process id.
`getppid()`	get parent process id.
`getuid()`	get real user id.
`geteuid()`	get effective user id.
`getgid()`	get real group id.
`getegid()`	get effective group id.
`setuid()`	set real and effective user id (if process has privilege).
`setgid()`	set real and effective group id (if process has privilege).
`getlogin()`	returns user login name.
`uname()`	returns O/S name, version, etc.
`time()`	returns system time.
`times()`	returns time accounting data.
`getenv(name)`	returns value (if any) assigned to `name` in environment list.
`ctermid()`	returns pathname to controlling terminal.

`isatty(fd)`	returns boolean: does `fd` refer to a terminal?
`ttyname(fd)`	returns pathname for terminal associated with `fd`.
`sysconf(name)`	returns value (or limit) of system variable `name`.

3.2.1.3 File Management Primitives

`chdir(path)`	change current working directory.
`getcwd(buf, size)`	returns absolute pathname of current working directory in buffer.
`open(path, oflag, ...)`	opens file named by `path`, and returns a file descriptor.
`creat(path, mode)`	creates file with specified mode at specified location in file system.
`umask(cmask)`	sets file creation mask, returns previous mask.
`link(path1, path2)`	*atomically* links existing file (`path1`) to specified directory entry (`path2`) and increments reference count in inode.

`mkdir(path, mode)`	creates a directory.
`mkfifo(path, mode)`	creates a FIFO.
`unlink(path)`	unlinks specified directory entry from file and decrements reference count; file is deleted when reference count decrements to zero, and no process has the file open.
`rmdir(path)`	removes a directory.
`rename(old, new)`	replaces name field of directory entry.

`stat(path, buf)`
`fstat(fd, buf)`
	returns file status.

`access(path,mode)`	returns boolean: can file be accessed with `mode`?
`chmod(path, mode)`	change file mode.
`chown(path, owner, group)`	change owner of file.
`utime(path, times)`	set file accessed and modified times.

`pathconf(path, name)`
`fpathconf(fd, name)`
	query configurable pathname variables.

3.2.1.4 Input/Output Primitives

`pipe(fd)`	creates a pipe, returns file descriptors for the input and output ends of the pipe.
`close(fd)`	deallocates a file descriptor.
`read(fd,buf,nbyte)`	reads from a file into a buffer; returns number of bytes read; updates file offset.
`write(fd,buf,nbyte)`	writes from buffer to a file; returns bytes written; updates file offset.
`fcntl(fd,cmd,...)`	performs control operations on open file.
`lseek(fd,offset,whence)`	sets file offset.

3.2.2 GMMP Functions

3.2.2.1 Memory Management

`newseg(mode,max,min,[path])`	allocates segment of virtual memory with specified name (if given) and access modes; ref count set to one; returns ssn.
`map(path,[mode])`	maps specified file (absolute or relative pathname) into a segment with specified (or file default) access modes; increments reference count; returns ssn.

`copy(ssn)`	returns ssn of a new segment which shares all pages of given segment copy-on-write.
`pushseg(ssn)`	if segment is named, copies modified pages to secondary storage, else error.
`memlock(ssn)`	prevents pages of segment from being paged out.
`munlock(ssn)`	allows pages of segment to be paged out; (a segment is *automatically* unlocked whenever its reference count decrements to zero, or it is found to be unreferenced by the garbage collector).
`freeseg(ssn)`	removes segment from process access space, decrements reference count.

3.2.2.2 Message Queue Management

`mqcreate()`	creates a message queue; returns a message queue id which may be used by other processes to place messages in this queue; only the creating process may read from this queue.
`mqflush(mqid)`	removes all messages in specified queue.

`mqdestroy(mqid)`	removes all messages in specified queue and blocks further use of the queue. (The queue id may be re-used after the creating process has terminated.)

3.2.2.3 Communication / Synchronization

`send(mqid,cmd,id,len,data, [replyq,[time]])`	sends message to indicated message queue (mqid); message contains `cmd`, message `id`, and a `data` field of length `len`; if a `replyq` is given, the process is blocked pending receipt of a reply to that queue with a matching message id; if, in addition, a `time` is given, this bounds the time the process will wait for a reply.
`remap(mqid,segments)`	remaps a list of segments from the calling process' access environment to that of the process which created `mqid`; calling process is blocked pending a response from the distant local agent.

`rcv(mqid,[time])`	returns a pointer to a message structure containing the source and destination message queue ids, and the `cmd`, `len`, `id`, and `data` fields of the first message in the specified message queue; the process is blocked until a message is available; if a `time` is given, this bounds the time that the process will wait for a message.
`mwait(mask,[time])`	waits for a message from any of the message queues selected by `mask`; if a `time` *is given, this bounds the time that* the process will wait; returns a pointer to a message structure as for `rcv`.
`reply(msg,len,data)`	sends `data` to reply queue specified in `msg`, echoing the message id from `msg`.

3.2.3 Process Control Functions

`getpid()`	returns own pid.
`setpri(pri,[pid])`	sets process scheduling priority; if `pid` is absent, sets own priority; invokes the scheduler.

`getpri([pid])`	returns process priority; if `pid` is absent, returns own priority.
`plock()`	running process is granted sole use of CPU (interrupts still processed); automatically unlocked by process termination or suspension.
`punlock()`	unlocks CPU, invokes scheduler.
`suspend([pid])`	suspends a process; if `pid` is absent, suspends self; if process has locked its CPU, the CPU is unlocked, and a flag is *set in the suspended process structure*; may invoke scheduler.
`resume(pid)`	if process is suspended, it is returned to Ready state; may invoke scheduler; if process was suspended while running with its CPU locked, it immediately resumes running in that state.
`pstatus([pid])`	returns process status.

`migrate([pid])`	process is detached from PE and sent to PE Allocator for reassignment; if a `pid` is supplied, PE Allocator will attempt to co-locate migrating process with specified process.

3.3 Messages Accepted

The Clerk at each local agency has dedicated message queues to which messages may be addressed, which are distinct from those used by application or system processes running on the same processing element. The following messages are recognized by local agency Clerks:

`newproc(exseg)`	from PE Allocator; requests creation of a new process with the given execution environment; expects ACK or NAK.
`signal(signo)`	from any process; addressed to a resident process but intercepted by Clerk, which posts indicated signal `(signo)` to the destination process.
`suspend()`	from any process; addressed to a resident process but intercepted by Clerk, which suspends the destination process.

`resume()`	from any process; addressed to a resident process but intercepted by Clerk; if destination process is suspended, its state is set to Ready.
`debug()`	from the shell process responsible for the destination process; intercepted by Clerk; destination process enters debug mode.
`isgarbage(segments)`	requests determination of whether any of the listed segments are accessible by any local process; circulated among local agencies, which delete segments currently in use; originated by, and returned to, Object Manager.
`ptime()`	from any process; intercepted by Clerk, which returns CPU time used by destination process.
`tick(time)`	from real-time clock; broadcast at regular intervals to all Clerks; carries encoded current time.

3.4 Messages Generated

`open(uid,gid,cwd,path, [mode])`	to File Manager; expects ACK with ssn if access granted (segment mapped), NAK if not.
`chdir(uid,gid,cwd,path)`	to File Manager; expects ACK with new cwd or NAK.
`chown(uid,gid,cwd,path, newuid,newgid)`	to File Manager; expects ACK or NAK.
`chmod(uid,gid,cwd,path, mode)`	to File Manager; expects ACK or NAK.
`link(cwd,path1,path2)`	to File Manager; expects ACK or NAK.
`unlink(cwd,path)`	to File Manager; expects ACK or NAK.
`rename(cwd,old,new)`	to File Manager; expects ACK or NAK.
`mkdir(cwd, path, mode)`	adds a new directory to file system.
`mknod(cwd, path, mode)`	adds a new node to file system.
`newseg(mode, [cwd,path])`	to Object Manager; expects reply message with ssn if successful, NAK message otherwise; cwd|path optional.
`copy(ssn, [mode])`	to Object Manager; segment copied with (possibly reduced) access mode; expects ACK with ssn of new segment, or NAK.

`getsz(ssn)`	to Object Manager; returns the current valid virtual address range of a segment.
`setsz(ssn,max,min)`	to Object Manager; sets the current size of a segment.
`getrfct(ssn)`	to Object Manager; requests the current reference count of a segment.
`incrfct(ssn)`	to Object Manager; increments a segment reference count.
`freeseg(list)`	to Object Manager; decrements reference count(s) of list of segments; no reply.
`translate(ssn,pn,cycletype)`	to Pager, to repair address translation fault; expects ACK if no fault, `suspend()` followed by `resume()` if page fault, or NAK if impossible.
`pushseg(list)`	to Pager; modified pages of listed segments written to disk; expects ACK.
`remap(ssn,[mode])`	to another local agency, but addressed to a process message queue; receiving clerk adds segment `ssn` to access environment of destination process with given `mode`; expects ACK.

`signal(signo)`	addressed to a process but intercepted by its Clerk, which posts indicated signal `(signo)` to the destination process; sent in response to a `kill()` function call.
`suspend()`	addressed to a process but intercepted by its Clerk, which suspends the process.
`resume()`	addressed to a process but intercepted by its Clerk; if destination process is suspended, its state is set to Ready.
`newpe(pid1,pid2)`	to PE Allocator; requests migration of process `pid1`; if `pid2` is given, requests co-location with that process; expects ACK, or NAK if co-location denied.
`zombie(pid)`	to parent of a process which has stopped or terminated (e.g., `_exit()`); `pid` of zombie process is carried in message reply queue field.

4. FILE MANAGER

4.1 Function

As its name suggests, the File Manager, or FileMan, is the system process responsible for all aspects of file system management, including directory searching, access permission checking, and the creation, deletion, opening, and closing of files and directories. FileMan is the only process which can create, open, or modify directories. Related functions performed by other ARGOS processes include mapping files into segments (ObMan), paging file contents into main memory (Pager), and disk I/O and file header manipulation (DiskMan).

All file or directory manipulation requests from other processes are sent (via local agents) to the FileMan. The primary FileMan data structures is a File Table, indexed by SSN, which contains pointers to dynamically-allocated File Table Entries (FTEs), each of which stores the following information about an *open* file (* items from file header):

- file system number (fsn) and file number (fn)
- * owner uid and gid
- * access permissions (r w x for owner, group, and world)
- * file type
- * access times
- * link count
- mount data
- SSN of parent (..), an open sibling, and an open child (familial cone)
- reference count
- name (relative to parent directory)

FileMan keeps all directories from the root to the current working directory of every process open. When a request to open a file arrives, FileMan begins parsing the path name at the position in the FileTable given in the message by the SSN of the process current working directory (`cwd`). As each element of the path name is extracted, the ring of open children of the current directory is examined before reading the contents of the directory file itself, so that directories and files already open may be quickly identified. If the name search fails in the FileTable, the directory is read to attempt to find the file number of the requested file.

If the requested file is found, FileMan allocates a FTE, gets the file access record from the appropriate DiskMan to fill in the FTE, and sends a message to the Object Manager (ObMan) requesting that the file be mapped into a segment of system virtual memory. (This same request is made by FileMan when it needs a directory opened.) When the file has been mapped (by the Pager), the SSN of the segment is returned to FileMan, which adds the new FTE to the FileTable, and returns the SSN to the process which requested that the file be opened.

4.2 Messages Accepted

`open(uid,gid,cwd,path, [mode])`	returns `ACK` with ssn if access granted (segment mapped), `NAK` if not.
`creat(cwd,path,mode,uid, gid)`	creates new directory entry at given location with given access modes.
`mkdir(cwd,path,mode)`	adds a new directory to file system.
`mknod(cwd,path,mode)`	adds a new node to file system.

`chdir(uid,gid,cwd,path)`	returns ACK with new cwd or NAK.
`chown(uid,gid, cwd, path, newuid, newgid)`	returns ACK or NAK.
`chmod(uid,gid,cwd,path, mode)`	returns ACK or NAK.
`link(cwd,path1,path2)`	creates new directory entry given by cwd|path1, and links it to inode referenced by cwd|path2; returns ACK or NAK.
`unlink(cwd,path)`	from a Local Agent; checks File Table for process accessibility; if free, unlinks name from file, removes directory entry; if this was last link, frees file header and releases disk space; returns ACK when done, or immediate NAK .
`rename(cwd,old,new)`	returns ACK or NAK.
`writehdr(ssn,size, modified)`	from Object Manager; update file header data including size and access times.
`close(ssn,size,modified)`	from Object Manager; update file header data including size and access times; mark File Table Entry as closed, but retain for possible reuse until ssn reused.

4.3 Messages Generated

`map(fsn,fn,mode,size)`	to Object Manager; requests that a segment be allocated, and that the file given by `fsn|fn` be mapped into it; expects ACK with ssn, or NAK.
`reopen(ssn,mode)`	to Object Manager; a file which was previously mapped into a segment can re-opened in place.
`hdralloc()`	to DiskMan; allocate a new file header in the file system indicated by the mqid; returns `fn`.
`hdrfree(fn)`	to DiskMan; free a file header; mqid is fsn.
`getfar(fn)`	to DiskMan; requests the file access record from file header for the indicated file; mqid is fsn.
`putfar(fn,far)`	to DiskMan; update file access record of file header; mqid is fsn; returns ACK/NAK.

5. OBJECT MANAGER

5.1 Function

Whereas FileMan manages access to files as they are stored in file systems, the Object Manager (ObMan) manages access to segments of virtual memory, including files when they are mapped into segments. ObMan's responsibilities consist of the allocation and recovery of segments of system virtual memory, management of their sizes, and control of access to the data contained in those segments.

The principal ObMan data structure is the Object Table (ObTab), which contains pointers to dynamically-allocated ObTab Entries (OTE), each of which holds the access mode, fsn, fn, maximum and minimum virtual addresses, and the reference count of an active segment. These OTEs are also linked in a hash queue by fsn|fn, so that multiple mapping of files can be avoided.

The range of valid virtual addresses of a segment is assigned by ObMan when the segment is created. ObMan calculates the range of valid page numbers in the segment, and passes these values to the Pager, which then checks for out-of-bounds references whenever an address fault occurs. A segment's size can be queried and set by messages to ObMan. If a segment is written to disk, ObMan supplies its size to set the file length.

When ObMan is requested to copy a segment, it assumes that copy on write protection is required for both copies; otherwise, no copy need be made, since read- or execute-only segments may be freely shared by merely incrementing their ObTab reference counts. The duplicate segment initially reflects all of the pages of the original segment, even if they have not yet been demand-paged into memory from a file, but the duplicate does not inherit a file name to write to; it is in that sense "nameless." Only one segment which maps a file retains the right to write to that file.

When the ObTab reference count of a segment drops to zero, ObMan notifies the Pager and FileMan that the segment is no longer in use; all three processes retain some data in their tables about the segment, however, until the segment number is reused, in order to minimize the time required to re-map the file should it be opened again soon. A file mode bit called the "sticky" bit causes ObMan to keep a file mapped into a segment, even when the segment's reference count drops to zero, until the FileMan sends ObMan a message indicating that the file need be retained no longer (e.g., when the file is unlinked or unmounted). Writing to a file need not reset its sticky bit.

5.2 Messages Accepted

`map(fsn,fn,mode,size)`	from File Manager; allocates ssn, invokes Pager with the given file data; returns ACK with ssn, or NAK.
`reopen(ssn,mode)`	from File Manager; a file which was previously mapped into a segment can re-opened in place.
`newseg(mode,max,min, [cwd, path,uid,gid])`	allocate a new segment; returns ACK with ssn if successful, NAK otherwise. `cwd, path, uid,` and `gid` are optional; if supplied, a `creat` call is made on File Manager.

`name(ssn,cwd,path,uid,gid)`	from Local Agent; updates the file name associated with a segment; a `creat` call is made on File Manager (using present `mode`), which expects `fsn|fn`.
`copy(ssn,[mode])`	segment copied with (possibly reduced) access mode; returns `ACK` with ssn of new segment, or `NAK` if unknown ssn.
`write(ssn)`	modified pages of segment are written to disk (Pager), and file header is updated (FileMan).
`getrfct(ssn)`	returns the current reference count of a segment.
`incrfct(ssn)`	increments a segment reference count.
`freeseg(ssn)`	segment reference count decremented; segment marked as free if reference count equals zero; no reply.
`getsz(ssn)`	returns the current valid virtual address range of a segment.
`setsz(ssn,max,min)`	sets the current size of a segment.

5.3 Messages Generated

`creat(cwd,path,mode,uid, gid)`	to FileMan; requests creation of a new directory entry at given location with given access modes.
`writehdr(ssn,size, modified)`	to FileMan; update file header data including size and access times.
`close(ssn,size,modified)`	to FileMan; update file header data including size and access times; mark File Table Entry as closed, but retain for possible reuse until `ssn` reused.
`map(fsn,fn,mode,ssn,max, min)`	to Pager; requests that the given file be mapped into a segment of system virtual memory with the given access modes.
`remap(ssn,mode)`	to Pager; re-map a previously mapped file; attempt to recover page frames containing tables and data.
`newseg(mode,ssn,max,min)`	to Pager; create a temporary segment with no backing location in the file system.

`copy(mode,ssn1,ssn2)`	from Object Manager; create a nameless segment `ssn2` whose pages all map to corresponding pages in `ssn1` with copy-on-write protection and access modes (possibly reduced) as specified.
`resize(ssn,max,min)`	to Pager; revise the range of valid pages in a segment.
`name(fsn,fn,ssn)`	to Pager; update location in file system to use for writing a segment to disk.
`flush(ssn)`	to Pager; requests that all modified pages of a segment be written to file; expects ACK or NAK when done or failed.
`float(ssn)`	to Pager; requests that all valid pages of segment not in memory be backed up on paging device, and that link to file system be severed.
`free(ssn)`	to Pager; free any page frames in use solely by this segment (discard modified pages); free page table and disk index (or PageMap) associated with segment.

`isgarbage(segments)`	to Local Agents; requests determination of whether any of the listed segments are accessible by any local process; circulated to local agencies, which delete ssns currently in use, and returned to ObMan.

6. PAGER

6.1 Function

The Pager translates virtual addresses to physical addresses, attempting to keep the most useful such translations cached in the ATran, and supervises the movement of pages of virtual memory between main and secondary storage. The related function of attempting to keep the working sets of the active processes in page frames of main memory is properly in the domain of the Page Frame Manager.

The Pager data structures were chosen to efficiently support GMMP memory manipulations while minimizing the amount of system virtual memory required. It was also desired to have a single virtual address by which all copies of a copy-on-write shared page could be named, and a single location for storage and retrieval of Pager data related to each shared page.

The data structure chosen to support copy-on-write sharing of pages of segments is a *familial cone* of segment maps (SegMaps) as shown in Figure 6.1. Similar to a tree, a familial cone has a single apex (root); any child has a single parent, and each child has a pointer to its parent; the parent, however, has only a single child pointer, so all of the children of a parent are linked in a sibling ring for access from the parent.

A SegMap data structure is allocated for each active segment, and contains sundry information about the segment (Figure 6.2), including pointers to a page table and disk index (apex SegMaps only) or to a PageMap (non-apex SegMaps).

An apex SegMap is created whenever a new segment is created or a file is first mapped into a segment; it has no parent, siblings, or, initially, children. A new page table is created, with the range of valid pages determined by maximum and minimum page number values supplied by the ObMan.

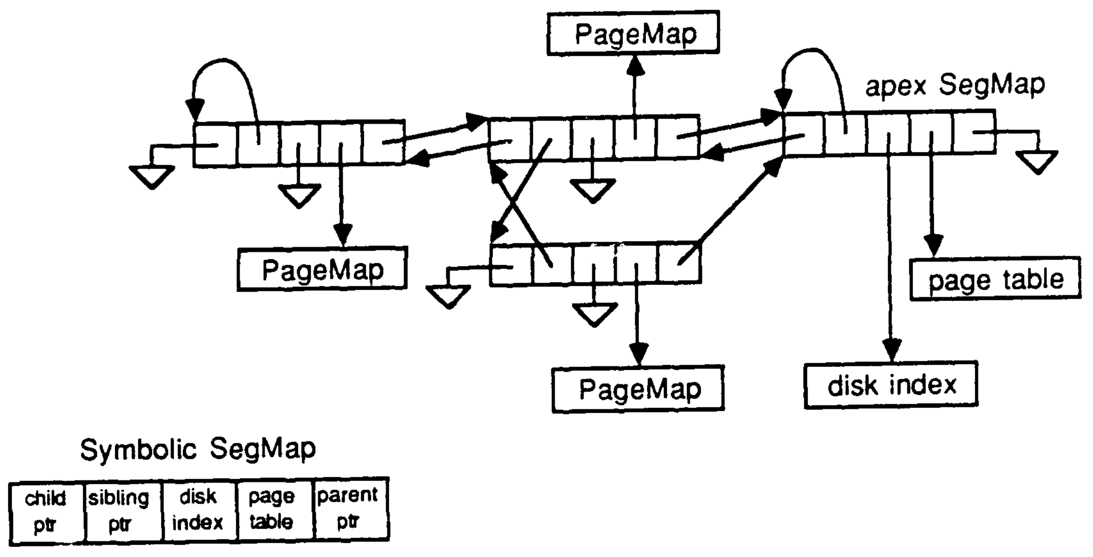

Figure 6.1: Familial Cone of SegMaps

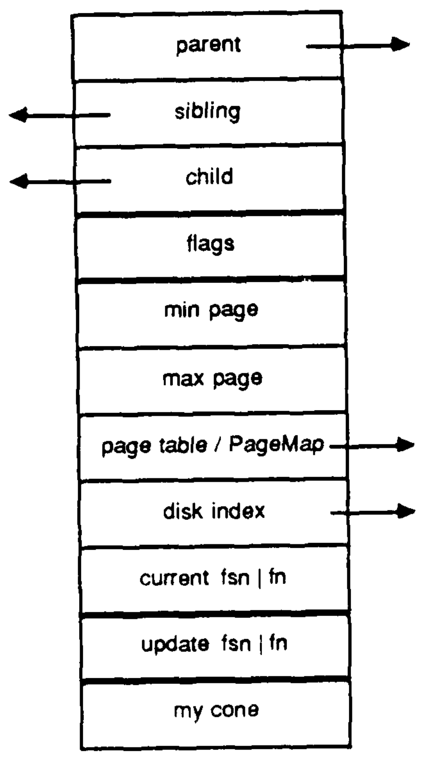

Figure 6.2: Format of SegMap

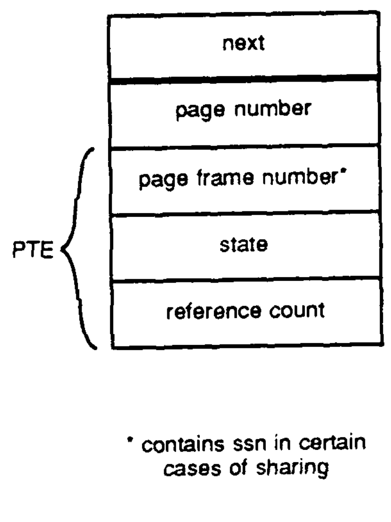

Figure 6.3: Format of PageMap Entry

If a file is associated with the new segment, the Pager sends messages to the appropriate DiskMan to read the file index from the file header block (and, when appropriate, indirect blocks) to build a disk index for the SegMap.

Children of a SegMap are created when the segment is copied. All pages modifiable by either parent or child are marked for copy-on-write protection. The PageMap of a child SegMap lists only those pages in which it differs from its parent, and is initially empty. A PageMap entry (Figure 6.3) contains fields to store a page number and the address of the next PageMap entry in a linked list, in addition to the fields contained in a page table entry (PTE).

A full page table, indexed by page number (pn), is maintained only for the segment at the apex of a cone of SegMaps. Page table entries contain fields for the page frame number (PFN) of a page, the state of the page (invalid, demand zero, demand fill, paged out, recoverable from PFM free list, etc.) and a reference count.

The Pager keeps a pool of allocated page frames available for ready use, replenishing this pool as needed by messages the Page Frame Manager (PFM). Because no process can access memory using physical addresses, the Pager is the only process which can use PFNs, and is the only process which ever requests page frames.

6.2 Messages Accepted

`translate(ssn,pn,cycletype)`	from a Local Agent; repair address translation fault; returns `ACK` if immediately successful, `suspend()` followed by `resume()` if page fault, or NAK if impossible.

`map(fsn,fn,mode,ssn,max, min)`	from Object Manager; requests that the given file be mapped into the given segment of system virtual memory with the given access modes and range of valid pages; pages marked "demand fill."
`remap(ssn,mode)`	from Object Manager; re-map a previously mapped file; attempt to recover page frames containing tables and data.
`newseg(mode,ssn,max,min)`	from Object Manager; create a temporary segment with the given range of valid "demand zero" pages, with no backing location in the file system (nameless).
`copy(mode,ssn1,ssn2)`	from Object Manager; create a nameless segment `ssn2` whose pages all map to corresponding pages in `ssn1` with copy-on-write protection and access modes (possibly reduced) as specified.
`resize(ssn,max,min)`	from Object Manager; revise the range of valid pages in a segment.
`name(fsn,fn,ssn)`	from Object Manager; update location in file system to use for writing a segment to disk.

`flush(ssn)`	from Object Manager; requests that all modified pages of a segment be written to file; returns ACK when done, or NAK if unknown ssn.
`float(ssn)`	from Object Manager; requests that all valid pages of segment not in memory be backed up on paging device, and that link to file system be severed (segment becomes nameless).
`free(ssn)`	from Object Manager; free any page frames in use solely by this segment (discarding modified pages); free page table and disk index (or PageMap) associated with segment.
`pageout(list)`	from PFM, with list of pages (ssn|pn) to be written to paging device; returns `pffree()` when page is free (after disk write if page has been modified).
`pfnr(list)`	from PFM, with list of pages which are no longer recoverable from PFM free list.

6.3 Messages Generated

`pfalloc(count)`	to PFM; allocate `count` page frames; expects ACK with list of PFNs, or NAK.
`pfassign(PFN,ssn,pn)`	to PFM; mark indicated page frame as "in use" with the indicated virtual address.
`pffree(list)`	to PFM; free the indicated list of page frames.
`pfrcvr(list)`	to PFM; recover indicated page frames from free list, mark as "in use"; expects ACK or NAK.
`getindex(fn)`	to DiskMan; requests file index of indicated file; mqid is fsn.
`putindex(fn,index)`	to DiskMan; update file index portion of file header; expects ACK or NAK; mqid is fsn.
`read(fn,ssn,pn,list)`	to DiskMan; read `list` of blocks into system virtual memory starting at ssn|pn from file `fn`; mqid is fsn.
`write(fn,ssn,pn,list)`	to DiskMan; write `list` of blocks from system virtual memory starting at ssn|pn to file `fn`; mqid is fsn.

`alloc(blocks,flags)`	to DiskMan; allocate `blocks` blocks from file system given by mqid; may request that blocks be contiguous.
`free(list)`	to DiskMan; free a list of disk blocks.
`getfrag(fn,ssn,pn)`	to DiskMan; read tail fragment of file to given virtual memory location.
`putfrag(fn,ssn,pn)`	to DiskMan; get tail fragment of file from the indicated virtual address; mqid is fsn.
`pglstsz()`	to paging device DiskMan; requests number of pages in pageout batch.
`pageout(list)`	to paging device DiskMan; write list of pages (ssn|pn) to paging device, ACK.
`pagein(ssn,pn)`	to paging device DiskMan; read page from paging device; expects ACK.
`pfree(ssn,pn)`	to paging device DiskMan; free space on paging device occupied by page; no ACK.
`pfile(ssn,pn,fsn,list)`	to paging device DiskMan; copy page from paging device to `list` of blocks in file system `fsn`, then free paging space; expects ACK.

7. PAGE FRAME MANAGER

7.1 Function

The Page Frame Manager (PFM) manages the use of main memory: it allocates page frames of physical memory and initiates the page-out of inactive pages. The principal data structures of the PFM are a Page Frame Table and in-use and free lists.

The Page Frame Table contains an entry for every page frame available in the physical address space. Each entry contains fields for the state of the page frame, a pointer into the free list, and a virtual memory segment number and page number associated with the page frame. Each page frame is in one of five states: unusable (hardware missing or failed), free, allocated, in use, or locked (being freed by Pager).

Page frames are initially free. They are allocated to system processes in response to `pfalloc()` messages, and remain in the allocated state until assigned virtual addresses by `pfassign()` messages; when a page frame is assigned a virtual address, its state changes to "in use" and it becomes eligible for paging. Pages holding same system data structures are never "assigned," and are therefore immune to pageout.

The PFM maintains a list of free page frames available for allocation to other processes. When the length of this list falls below a low water mark, the PFM begins scanning its in-use list for page frames to free. When it has collected enough page frames to reach its high water mark, it sends the list of pages to be freed to the Pager in a `pageout()` message. Upon receipt of a `pffree()` message in response, the PFM marks the pages as free, and enters them on the free list. The page frames retain their previous contents until reallocated, however, and can be recovered from the free list by the Pager in a `pfrcvr()` message. When page frames previously in use are allocated by the PFM from the free list, a `pfnr()` message is sent to the Pager, listing the previous virtual addresses of page frames which are no longer recoverable.

7.2 Messages Accepted

`pfalloc(count)`	allocate `count` page frames; returns list of PFN, or NAK.
`pfassign(PFN,ssn,pn)`	mark page frame as in use, and record its virtual address.
`pffree(list)`	free the indicated list of page frames.
`pfrcvr(list)`	recover indicated page frames from free list, mark as in use; returns ACK or NAK.

7.3 Messages Generated

`pageout(list)`	to Pager, with list of pages; expects `pffree` in acknowledgement.
`pfnr(list)`	to Pager, with list of virtual addresses of pages which are no longer recoverable.

8. DISK MANAGERS

8.1 Function

ARGOS disk manager processes run on the disk I/O controller (IOC) processors. Each allocates disk space on the disk drives attached to that IOC, manages I/O transfers to and from its disks, and manipulates the file headers in the file systems (partitions) on those disks.

ARGOS processes specify "logical device numbers" (i.e., disk partitions) by sending requests to the unique message queue allocated for each partition when the system powers up; each DiskMan process itself also has a unique queue which is used for messages not directly associated with a particular partition, such as formatting and partitioning an entire disk. Requests for allocation of disk blocks are directed to the message queue assigned to the file system of interest. Read and write requests are also sent to a file system mqid (except for paging requests, as discussed below), and list the logical block numbers to be read or written and the starting system virtual address of the data in main memory.

Requests to read or write file headers are directed to a file system mqid, and specify the file number (originally assigned by the DiskMan) of the desired file. The DiskMan keeps a cache of recently-accessed header blocks, both for quick response to read requests, and so that most header block updates can update the affected component (file access record, file index, or fragment) in local IOC memory and then write the block, without requiring a preliminary disk access to read in the unaffected components.

Unlike file system I/O, in which the Pager directly uses logical block numbers, disk management for paging is handled entirely within the DiskMan which has the paging partition(s). Paging requests are addressed to a single message queue designated for paging requests, and contain only the system virtual addresses of the page(s) to be swapped in or out. The DiskMan swaps enough pages out at once to make the disk transfer efficient, storing whatever odd lot of pages it has accumulated contiguously on the paging device. Upon receipt of a request to swap a page back in, the DiskMan refers to its list of swapped pages to find its location, and initiates the transfer. Pages swapped back into memory are retained on the paging device until explicitly released. However, pages which are transferred from the paging partition to a file system are removed from the paging device immediately.

The latter type of transfers may require the allocation of a buffer segment in system virtual memory when the paging device and file system are managed by different Disk Manager processes. This requirement is a result of the decision to allow no process to access main memory using physical addresses.

8.2 Messages Accepted

8.2.1 Allocation

`alloc(blocks,flags)`	from Pager; allocate `blocks` blocks from file system given by mqid; may request that blocks be contiguous.
`free(list)`	from Pager; free a list of disk blocks.
`hdralloc()`	from FileMan; allocate a new file header in the file system indicated by the mqid; returns `fn`.

`hdrfree(fn)`	from FileMan; free a file header.

8.2.2 File I/O

`getfar(fn)`	from FileMan; requests the file access record from the indicated file header.
`putfar(fn,far)`	from FileMan; update file access record of file header; returns ACK or NAK.
`getindex(fn)`	from Pager; requests file index of indicated file.
`putindex(fn,index)`	from Pager; update file index portion of file header; returns ACK or NAK.
`read(fn,ssn,pn,list)`	from Pager; read `list` of blocks into system virtual memory starting at `ssn|pn` from file `fn`.
`write(fn,ssn,pn,list)`	from Pager; write `list` of blocks from system virtual memory starting at `ssn|pn` to file `fn`.
`getfrag(fn,ssn,pn)`	from Pager; read the tail fragment of a file to the indicated virtual memory location.
`putfrag(fn,ssn,pn)`	from Pager; get the tail fragment for a file from the indicated virtual address.

8.2.3 Paging

`pglstsz()`	from Pager; requests number of pages in pageout batch.
`pageout(list)`	from Pager; write list of pages (ssn|pn) to paging device, return ACK.
`pagein(ssn,pn)`	from Pager; read page from paging device, return ACK.
`pfree(ssn,pn)`	from Pager; free space on paging device occupied by page; no ACK.
`pfile(ssn,pn,fsn,list)`	from Pager; copy page from paging device to `list` of blocks in file system `fsn`, then free paging space, and return ACK.

8.2.4 Disk Management

`format(devid, list)`	format a disk with the listed parameters; returns mqid for entire disk.
`part(sizes)`	partition a disk into file systems of the given sizes; mqid designates disk to be partitioned; returns mqids for the new file systems.

`init(files, blocks, boot)`	initialize a file system; copy boot block from indicated system virtual address; mqid is fsn.
`fschk()`	requests audit of file system integrity; mqid is fsn.

8.3 Messages Generated

`newseg(mode,max,min)`	to ObMan; requests allocation of a buffer segment for inter-DiskMan transfers.
`freeseg(ssn)`	to ObMan; free a buffer segment.
`read(fn,ssn,pn,list)`	to another DiskMan; read `list` of blocks into system virtual memory starting at `ssn|pn` from file `fn`; mqid is fsn.
`write(fn,ssn,pn,list)`	to another DiskMan; write `list` of blocks from system virtual memory starting at `ssn|pn` to file `fn`; mqid is fsn.

9. PE ALLOCATOR

9.1 Function

The PE allocator process(es) assign processes to processors and initiate their execution; one PE allocator exists for each type of PE in a system. New processes come into being in ARGOS (after power-up) by process forking at PEs or UIPs. In the case of PE forking, the new process may choose to execute on its original host PE, or it may migrate to another PE via a `newpe` message to the appropriate PE allocator. Processes forked at UIPs are sent to a PE allocator for PE assignment in a `launch` message. In either case, a PE allocator assigns the process to a new PE, and sends it there in a `newproc` message, which is received by the local Clerk. The Clerk adds the process to the local set of processes, if space in the local process table is available.

The PE allocator(s) attempt to make intelligent decisions in assigning processes to processors by monitoring the load on each PE via processor-specific messages circulated among the PEs.

9.2 Messages Accepted

`launch(exseg)`	requests creation of a new process with the given execution environment.
`newpe(pid1,pid2)`	requests migration of process `pid1`; if `pid2` is given, requests co-location with that process; returns ACK when migration complete, or NAK if co-location denied.

9.3 Messages Generated

`newproc(exseg)`	to local agency; requests creation of a new process with the given execution environment; expects ACK or NAK.
`cpuload()`	to a local agency; requests recent utilization of CPU; responses are processor-specific.

10. USER INTERFACE

10.1 Function

The ARGOS user interface processes execute "shell" (command interpreter) programs on User Interface Processors (UIP). These processes are created at system power-up, one per attached workstation or communication port; a new processes is forked for each connection established on a communication port. A user interface process may be entirely resident on a UIP, or it may be split between a remote host or workstation and local interface processor.

A user interface process carries out user commands by sending messages to the appropriate ARGOS processes to obtain information (e.g., the contents of a file system directory), or to launch new processes. When a new process is created, the user interface process creates its execution segment, copying the User-level context from its own context, and sends a `launch` message to the appropriate PE allocator.

10.2 Messages Accepted

`zombie(pid)`	from a child process which has stopped or terminated (e.g., `_exit()`); `pid` of zombie process is carried in message reply queue field.

10.3 Messages Generated

`launch(exseg)`	to PE allocator; requests creation of a new process with the given execution environment.
`debug()`	to a child process; intercepted by a Clerk; child process enters debug mode.
`ptime()`	to a child process; intercepted by Clerk, which returns CPU time used by process.
`open(uid,gid,cwd,path, [mode])`	to File Manager; expects ACK with ssn if access granted (segment mapped), NAK if not.
`chdir(uid,gid,cwd,path)`	to File Manager; expects ACK with new cwd or NAK.
`chown(uid,gid,cwd,path, newuid,newgid)`	to File Manager; expects ACK or NAK.
`chmod(uid,gid,cwd,path, mode)`	to File Manager; expects ACK or NAK.
`link(cwd,path1,path2)`	to File Manager; expects ACK or NAK.
`unlink(cwd,path)`	to File Manager; expects ACK or NAK.
`rename(cwd,old,new)`	to File Manager; expects ACK or NAK.
`mkdir(cwd path, mode)`	adds a new directory to file system.

`mknod(cwd, path, mode)`	adds a new node to file system.
`newseg(mode,[cwd,path])`	to Object Manager; expects reply message with ssn if successful, `NAK` message otherwise; `cwd|path` optional.
`copy(ssn,[mode])`	to Object Manager; segment copied with (possibly reduced) access mode; expects `ACK` with ssn of new segment, or `NAK`.
`getsz(ssn)`	to Object Manager; returns the current valid virtual address range of a segment.
`setsz(ssn,max,min)`	to Object Manager; sets the current size of a segment.
`getrfct(ssn)`	to Object Manager; requests the current reference count of a segment.
`incrfct(ssn)`	to Object Manager; increments a segment reference count.
`freeseg(list)`	to Object Manager; decrements reference count(s) of list of segments; no reply expected.

`translate(ssn,pn,cycletype)`	to Pager, to repair address translation fault; expects `ACK` if immediately successful, `suspend()` followed by `resume()` if page fault, or `NAK` if impossible.
`signal(signo)`	addressed to a child process; intercepted by its Clerk, which posts indicated signal `(signo)` to the destination process.
`suspend()`	addressed to a child process; intercepted by its Clerk, which suspends the destination process.
`resume()`	addressed to a child process; intercepted by its Clerk; if destination process is suspended, its state is set to Ready.

11. OTHER PROCESSES

11.1 I/O Device Drivers

For UNIX compatibility, I/O device drivers should appear as special files in the file system. ARGOS implements device drivers as processes executing on I/O controllers, which are known to the system by their pids or mqids (which are the same).

Translation of special file names to mqids is performed by the FileMan. The `/dev` directory is opened by the FileMan at system power-up; as device driver processes are assigned mqids, these are recorded by FileMan in the appropriate positions in this segment for later use in this translation.

11.2 Name Server

The NameServer process also maintains a list of symbolic name to mqid translations. Unlike the `/etc` segment, which only contains translations for I/O devices, the NameServer may be used by any process to `post` a translation meaningful to that process. In this way, processes may cooperate without a requirement for a common parent process. The NameServer accepts the following messages:

`post(mqid,name)`	an explicit request to make a symbolic name known.
`lookup(name)`	returns corresponding mqid, or NAK.
`remove(mqid)`	removes all names corresponding to `mqid`; may be sent by a process, or by a local agent when a process which has posted names terminates.

CONCLUSION

ARGOS has been designed to explore operating system concepts for GMMP multiprocessors. The intended function of the prototype Virtual Port Memory machine on which ARGOS will run is real-time image processing, information fusion, and system control, rather than the more typical applications for multiprocessors in large-scale scientific computations; this has given ARGOS a somewhat different orientation than that of multiprocessor operating systems designed for the "scientific" regime. It will be interesting to evaluate ARGOS' performance for the two different application domains, to see if a real-time orientation significantly affects its ability to manage and support computationally-intensive processing.

It will also be interesting to see how well the system process partitioning works, and whether the data structures chosen (especially the SegMap/PageMap arrangement) are the best we can do. Some areas for future investigation are possibly merging the Page Frame Manager with the Pager, if the Pager load turns out to be sufficiently light, and employing variable-sized "regions" versus fixed-size segments for partitioning system virtual memory.

ACKNOWLEDGEMENTS

The design of ARGOS has drawn on good ideas from many previous operating systems, including Multics [8], Unix [1], Mach [9], the AT&T Oryx/Pecos system [10], the Sandia Hawk real-time system [11], and SMRTE (a previous effort by the author).

Many individuals have contributed ideas and/or criticism to the design of ARGOS, including John Johnston at NMSU, Jane Gambill, Hector Urroz, and others at Bell

Laboratories, and Victor Holmes at Sandia National Laboratories. Arthur Karshmer, Joseph Pfeiffer, and John Johnston at NMSU graciously gave of their time to review the final draft of this report.

REFERENCES

1. Bach, *The Design of the Unix Operating System,* Prentice-Hall Software Series, Kernighan ed., Prentice-Hall, Englewood Cliffs, NJ, 1986.

2. **IEEE Std 1003.1-1988,** *Portable Operating System Interface for Computer Environments,* IEEE, 1988.

3. Johnson, E.E., "Completing an MIMD Multiprocessor Taxonomy," *Computer Architecture News,* **16**(3): 44-47, 1988.

4. Johnson, E.E., "A Prototype Virtual Port Memory Multiprocessor," Technical Report NMSU-ECE-88-003, NMSU, May 1988.

5. Johnson, E.E., "The Virtual Port Memory Multiprocessor Architecture," Technical Report NMSU-ECE-88-001, NMSU, January 1988.

6. Johnson, E.E., "GMMP Multiprocessor Architectures," *Proceedings, International Conference on Computing and Information* (1989).

7. Johnson, E.E., "The Virtual Port Memory GMMP Multiprocessor," *Proceedings, International Conference on Computing and Information* (1989): 127-130.

8. Organick, E.I., *The Multics System: An Examination of Its Structure,* MIT Press, Cambridge, Massachusetts, 1972.

9. Rashid, R.*et al.*, "Machine-independent Virtual Memory Management for Paged Uniprocessor and Multiprocessor Architectures," *Proceedings, Second International Conference on Architectural Support for Programming Languages and Operating Systems* (1987): 31-39.

10. Sager, G.R. and al, e., "The Oryx/Pecos Operating System," *AT&T Technical Journal,* **64**(1): 251-268, 1985.

11. Holmes, V.P. and Harris, D.L., "A Designer's Perspective of the Hawk Multiprocessor Operating System Kernel," *Operating Systems Review,* **23**(3): 158-172, 1989.

GLOSSARY

Term	(see page)	Meaning
ARGOS	1	a research GMMP operating system.
cwd	39	current working directory; an index into a File Manager table of open directories; stored in process structure and sent to File Manager to identify starting point for parsing relative path names.
DiskMan	57	disk manager process.
ExSeg	13	execution segment of a process; holds its process structure, stacks, and other record of execution.
familial cone	48	a data structure which links the children of one parent (siblings) into a ring; the parent has a pointer to one element of this ring, while all elements of the ring have pointers to the parent.
`far`	16	file access record, similar to UNIX inode *sans* index; stored in file header block in a file system.
FileMan	38	File Manager process; manages file access and searches directories.
fn	17	file number; selects one file within a file system (see fsn); corresponds to Unix inode number.

fsn	17	file system number (logical disk number); same as the mqid of the message queue used for requests to that file system.
FTE	38	file table entry.
gid	14	group identifier, as in Unix.
mqid	14	message queue identifier; an integer composed of the SSN of the process which created the queue, concatenated with an index value to distinguish among the queues owned by a process.
PageMap	49, 50	a linked list of pages accessible by a process which differ from the corresponding pages in a segment from which this segment of interest was copied.
pid	15	process identifier; equals message queue number of kernel message queue for a process, so that messages may be directed to the pid of a process.
PFM	55	page frame manager process.
pfn	50	page frame number; used as an index by the Page Frame Manager to identify blocks of physical memory.
pn	50	page number; used by pager as index into segment page tables; part of virtual address generated by processing elements.

PTE	**49**, 50	page table entry; a full page table is allocated by the Pager only for the first instance of a segment; all copies then use PageMaps to list pages in which they differ.
SegMap	48, **49**	segment map; a data structure used by the Pager to store information about an active segment.
sibling ring	48	see familial cone, above; a circularly-linked list of children of one parent.
ssn	14	system segment number; used by Object Manager, File Manager, and Pager as an index into segment tables; part of virtual address generated by processing elements.
uid	14	user identifier, as in Unix.