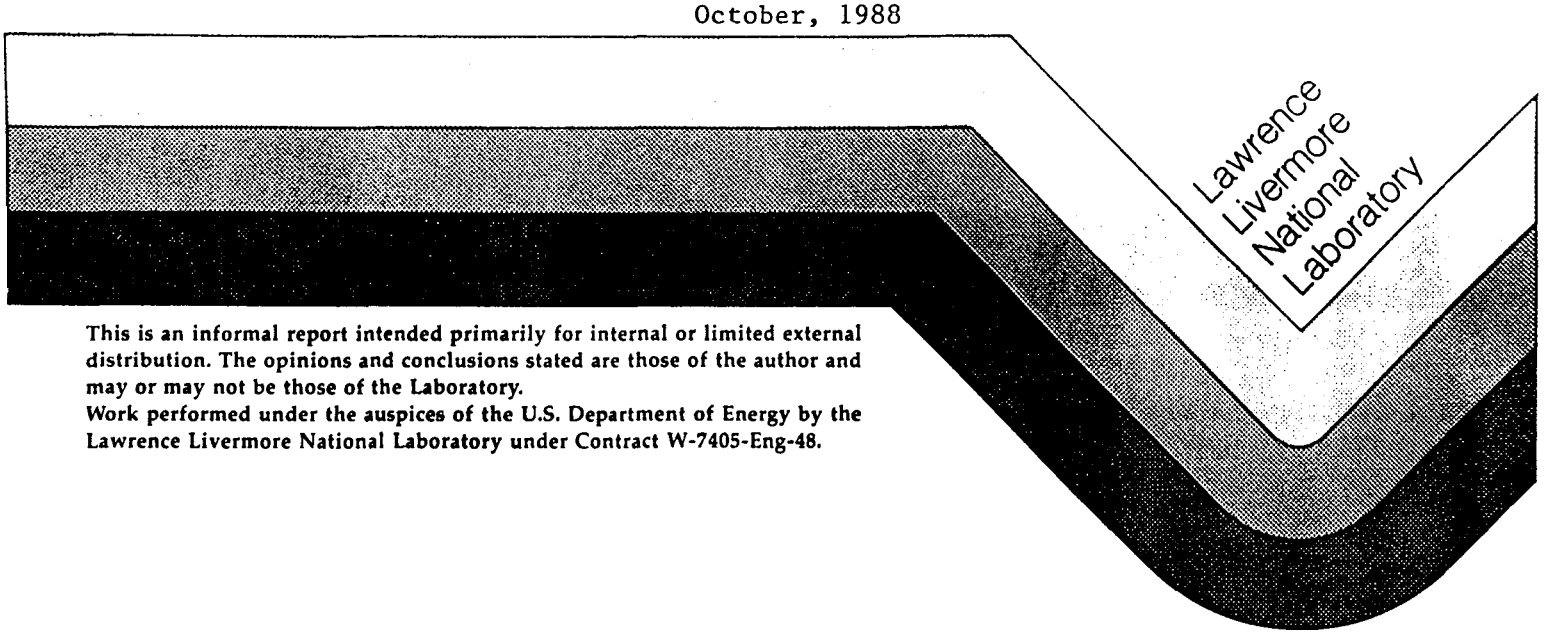
UCID- 21567

# AMBER

Jay A. Pattin



October, 1988



---

This report has been reproduced
directly from the best available copy.

Available to DOE and DOE contractors from the
Office of Scientific and Technical Information
P.O. Box 62, Oak Ridge, TN 37831
Prices available from (615) 576-8401, FTS 626-8401.

Available to the public from the
National Technical Information Service
U.S. Department of Commerce
5285 Port Royal Rd.,
Springfield, VA 22161

| Price Code | Page Range |
|---|---|
| A01 | Microfiche |
| **Papercopy Prices** | |
| A02 | 1- 10 |
| A03 | 11- 50 |
| A04 | 51- 75 |
| A05 | 76-100 |
| A06 | 101-125 |
| A07 | 126-150 |
| A08 | 151-175 |
| A09 | 176-200 |
| A10 | 201-225 |
| A11 | 226-250 |
| A12 | 251-275 |
| A13 | 276-300 |
| A14 | 301-325 |
| A15 | 326-350 |
| A16 | 351-375 |
| A17 | 376-400 |
| A18 | 401-425 |
| A19 | 426-450 |
| A20 | 451-475 |
| A21 | 476-500 |
| A22 | 501-525 |
| A23 | 526-550 |
| A24 | 551-575 |
| A25 | 576-600 |
| A99 | 601 & UP |

UCID--21567

DE90 011766

# AMBER

## Abstract

Amber is an operating system being developed by the S-1 project at the Lawrence Livermore National Laboratory. It is designed to execute on large, shared memory symmetric multiprocessors, but is also adaptable to workstation-like uniprocessors.

The Amber kernel requirements specified that Amber be able to support a broad range of computing needs including interactive, time sharing use; compute-intensive tasks; and, within limits, real time control systems. Amber was designed to work in secure environments, without allowing the security model to impinge upon information and code sharing with no security implications.

Amber was designed to be reliable, maintainable, and portable. Reliability concerns are addressed through dynamic reconfigurability, software consistency checking, and data redundancy. Maintainability and portability are assured through the use of a high-level implementation language and modular programming techniques.

In describing the Amber kernel, this paper covers Amber objects, access control, the storage system, the traffic controller (schedular), interprocess communication, I/O, resource management, modularity, reliability, and portability.

One of Amber's design goals was based on the kernel model, that functions should only be provided by the kernel if they absolutely had to be. This follows the principals of least privilege and least common mechanism [SALTZER75] and aids in keeping the kernel small, understandable, maintainable, and verifiable. The influence of the Multics Kernel Redesign Project [KERNEL] was considerable.

For instance, in most operating systems with hierarchical file systems, the resolution of a textual pathname into a file system object is performed inside the kernel. In Amber, this is not true: the kernel provides operations to list the contents of a directory, and to look up a single name in a directory. User applications may build whatever vvpathname resolution methods they wish out of these operations. As a trivial detail, the character used to delimit the filenames ('/' in Unix, '>' in Multics) can be defined by individual applications.

## Kernel-Provided Objects

The Amber kernel provides seven primitive object types. Users may, of course, build more complicated abstractions out of these primitives. The objects are: segments, domains, tasks, message channels, special segments, dummy objects, and links.

The first six are actual information containers of different sorts; links are a distinct abstraction which is only used in the access control mechanism. They will be described later. All of the real objects have a unique identifier (UID) and share uniform naming, access control, and attributes mechanisms.

## Segments

Segments are simply information containers; the kernel treats them as a vector of bits, and allows users to map them into the address space and then access them directly with host's hardware instructions. Outside the kernel, users can implement abstractions such as executable programs, record-oriented files, databases, stacks, and heaps out of segments.

## Domains

Domains are a fairly complicated object that provides a storage system catalog, a container for holding the attributes of other objects, storage system accounting, and the facilities for implementing protection domains.

First, domains are directories. The storage system is a hierarchy of domains where each domain contains other objects including other domains. Domains maintain the correspondence between names and objects and their representations (if any), and contain the attributes, properties, and access control list for all of the objects cataloged in them. Attributes include items such as object type, creator, date created, and date modified. Object attributes are maintained by the kernel. Amber objects also have a property list of property name, value pairs which can be managed by user programs. The access control list will be discussed below.

Storage system quota allocation is also done on a domain basis. Each domain has quota accounts on various storage system volumes. There is also a limit on the total quota wich can be consumed by a storage system subtree regardless of the allocation to quota accounts on different volumes.

The most interesting feature of Amber domains is their role in the protection scheme. Domains are used as principals in the discretionary and procedural access control schemes detailed below. An object, or a link to that object, must be cataloged within a domain in order for a task contained in that domain to reference the object. The domain also contains the description of the address space shared by tasks contained within the domain. The address space is represented as a list of pairs of segments (or link to segments) and virtual address ranges.

Amber domains are implemented as segments within the kernel. Protection of domains is maintained by never allowing a user task to map a domain into its address space.

## Tasks

Inactive or prototype user tasks are also part of the storage system. Sufficient state, including register values, program counter, and scheduling information, is stored in a domain to allow another task with appropriate access to quickly load and start a task. Tasks may be stopped and stored in a domain for reactivation later, even across bootloads. The containing domain describes the task's address space, and its stack, heap, and linkage information are stored in segments in that address space. There is no information about a quiescent task which is not found in the storage system.

A task executes in its parent domain. It also shares its address space and access rights with other tasks which may be executing in the same domain. The creator of a task must initialize the parent domain so that state necessary to start the new task exists. For instance, the program segment must be mapped into the address space, the stack must be created and mapped, and the initial values for registers in the task's state must be initialized. Once the task begins executing, it can modify its domain as necessary. Because task creation is a frequent operation and the work is somewhat tedious, the kernel provides an operation which 'clones' a template domain and activates tasks contained within it.

### Message channels

Message channels are bidirectional packet-oriented communications paths that are similar to Unix (TM) pipes. A message channel is represented by two file system objects. A message sent through one of the objects is directed to the task listening to the other object. Using distinct objects allows different access to be allowed on either end of the channel, making it easy to implement a user to trusted server communication path.

### Special segments

In order to allow users to access some I/O devices without kernel mediation, Amber supports the special segment object type. A special segment is associated with a portion of the kernel address space. Mapping a special segment into a user's address space allows direct access to kernel memory, including device registers.

### Dummy objects

Dummy objects are simply directory entries in a domain that have no associated representation. They do, however, have property lists, access control lists, and other system-maintained attributes. These objects can be used for modeling extended type objects which require no representation. For instance, a symbolic link can be represented by a dummy object with a 'target_pathname' property.

## ACCESS CONTROL

A primary Amber design goal was providing for controlled sharing of information and resources. The kernel provides for three types of access control: discretionary access control allows users to specify who may access their objects; procedural access control allows programs to interpretively determine access to an object; and mandatory access control specifies that certain information paths must not be allowed even if the other access methods would allow them.

## DISCRETIONARY ACCESS CONTROL

Traditionally, there have been two models of discretionary access control systems: list oriented and ticket oriented. In the list-oriented model each object has an associated list of subject allowed to access it. Before granting access, the kernel verifies that there is an appropriate entry for the subject on the object's access control list. In the ticket-oriented model, a subject obtains a ticket, frequently called a capability, from the kernel or from some other user process. Possession of a ticket allows access to the object the ticket represents without regard to the identity of the ticket holder.

Both models have advantages and disadvantages: the Amber scheme uses portions of both. The primary discretionary access control mechanism in Amber is the access control list (ACL) associated with each object. Each term on the ACL is a tuple consisting of a @i(principal) and a @i(mode set). A principal is the process or group of processes that is to be allowed to

perform operations on the object. The mode set controls which operations can be performed on the object by the associated principal.

There are associated mechanisms which are derived from a ticket-based scheme. Amber provides a way to pass access from one user to another, both for access to an individual object and for the rights to access other objects. In essence, Amber allows users to distribute tickets obtained using a list model, and to use tickets as principals to be compared against the contents of a list when doing access checks. Before describing the mechanics in detail, the remaining Amber primitive object must be described.

### Links

The Amber version of a ticket is the link. A link is an object which points to another object and contains information about how access to that object was obtained. References to objects not cataloged in a task's parent domain may not be made until the task has obtained a link to that object. Link creation is Amber's ticket-obtaining mechanism.

In order to create a link to an object, a task must have a link to the object's parent domain, the UID of the object, and a list of ways to obtain access to the object. The contents of this list is described below, but it can be thought of as a list of tickets which name principals whose access the task is allowed to use.

In order to start this process, each task's domain must be provided with a link to the root directory by the task's creator. This link, generally called '*root*' in applications, need not actually point to the genuine storage system root. If it points to another domain in the hierarchy, the task will not be able to access objects not in the subtree rooted in that domain (unless there are existing links which point out of that subtree). The secure computing aspects of this capability have been exploited by Gould's UTX/32S using the Unix 'chroot' system call.

A special kind of link, called a seal, allows tickets to be passed to other principals. When creating a link, a task can specify the identity of another principal who will be allowed to access the target of the new link using the access available to the calling task.

### Principals

In systems such as Multics [MULTICS] or Primos [], principals represent users or groups of users to whom access is to be given. Amber principals are represented by domain objects. Any domain in the storage system may be used as a principal. This allows access to be given to users, represented by the domain which is his home directory, or individual processes, represented by their domain of execution. A special principal which denotes the list of all principals is also provided, allowing universal access to be granted.

Amber provides a very powerful extension to the idea of an access principal: groups may be constructed through a 'power-of-attorney' or 'agency' scheme. A domain can be granted 'use' permission to any other domain, allowing it to act as if it possessed the same access as that domain. A set of domains representing many users may all be given 'use' permission on a domain that represents the whole group. That domain, in turn, may be given permission to objects throughout the storage system. A further advantage is that when group membership

changes, only the access to the domain representing the group must be changed.

'Use' permission may be chained together to form structures called @i(access paths). An access path is an ordered list of principals A.B.C...N such that for each component has 'use' permission on its successor. A has 'use' permission on B, B has 'use' permission on C, and so on. The access allowed by the access path to an object is determined by an entry on the object's ACL that matches the access path's final component.

The first element of an access path must be either the current executing domain or its owner. This is equivalent to the statement that any domain has implicit 'use' permission to itself and to its owner; a link is allowed to have a null access path only if it points to either its containing domain or that domain's owner. Note: a task which represents a user has that user's home domain as its owner.

Access paths are themselves represented as links, and are built in the following manner: Let X be the executing task's domain, and let A and B be other domains with appropriate permissions such that X.A.B is a valid access path. The task's owner will have provided with a link to its executing domain called @i(*self*) which has a null access path. The task would first create a link L1 to domain A which requested 'use' permission and specified @i(*self*) as the access path. Next, it would create a link L2 to domain B which requested 'use' permission and specified L1 as as an access path. The access path stored in the link L2 is X.A. If L2 is then used as an access path in a link creation, the resulting link will have an access path X.A.B; this process can continue indefinitely.

In accordance with the principle of least privilege [SALTZER75], an Amber user may dynamically alter her privilege as needed. In most systems, access is completely defined at logon time by the user ID and/or project ID. Some systems provide mechanisms for exercising additional privilege which may be enabled and disabled at runtime. An example of this is the TENEX 'wheel' capability [TENEX] which allows a user to override all protection mechanisms. This provides the capability to minimize the chances of falling prey to a Trojan Horse while providing the convenience of not having to log out and log in to a more privileged account.

Because access to an Amber object is determined when a link to that object is obtained, a fine granularity of privilege is available. When a user asks the kernel for a capability to an object, she presents a list of access paths. The kernel creates a link if there is an entry with the requested access for the final component on any path in that list on the object's ACL, or if there is an entry with sufficient access for the universal principal. The contents of the list can vary from call to call depending on what kind of access is requested.

In addition, a user can voluntarily restrict his access by placing an access ceiling mask in a link. A user can prevent accidental writes to a segment to which he has write permission by requesting a link with an access mask which does not permit writes.

All access obtained with links is subject to instant revocation. If the access control list of the target object is modified to delete the entry used to obtain the access, or if 'use' permission between any pair of adjacent elements of the link's access path is deleted. If access is later restored, or even increased, the same link may still be used.

All access-controlled operations except reading, writing, and executing segments are mediated by software, making revocation easy: if permission has been removed, the operation fails. Revocation of read, write, or execute permission to a segment is harder. Revocation caused

by changes to a segment's ACL is accomplished by invalidating all currently valid mappings of that segment, a procedure known as set-faulting [MULTICS]. Subsequent accesses to the segment will fault, and the fault handler will recalculate the executing domain's access to the segment.

Removal of 'use' permission somewhere in an access path used to obtain access must also result in access revocation. This is a somewhat harder problem; it requires remembering how access to segments was obtained. Amber maintains a cache describing the access dependencies of all valid segment mappings. When any 'use' access is deleted, all mappings which depended on that access are set-faulted. The access cache provides the indirection in referring to a capability target that is necessary for access revocation, but has almost no cost. Building the dependency graph is no harder than checking the access, which is a necessary function. The added cost is only incurred when access is deleted, a relatively rare occurrence.

## ACCESS MODES

All Amber objects share a set of generic access modes; each object type has a set of access modes, and some undefined access modes are available for use by protected type managers.

The four generic modes are:

| | |
|---|---|
| get | allows the attributes of an object to be examined |
| put | allows modification of an object's property list |
| listen | allows broadcast events associated with the object to be declared |
| broadcast | allows broadcast events associated with the object to be signaled |

The access modes associated with domains are:

| | |
|---|---|
| find | permits the domain to be searched for an object with a particular name |
| list | permits the contents of a domain to be listed |
| modify | permits the contents of a domain to be changed. Objects may be created and deleted, and their names and ACLs may be changed |
| use | permits another domain to act as an agent of this domain by using access granted to this domain |
| invoke | allows a copy of a domain to created for the purpose of executing a protected type manager in the domain |

The access modes associated with segments are:

| | |
|---|---|
| read | permits the contents of the segment to be read |
| write | allows the contents of the segment to be modified |
| execute | allows the contents of the segment to be executed. Amber does not support execute-only segments, so read permission must also be present |

The access modes associated with tasks are:

| | |
|---|---|
| status | permits reading of a task's state and scheduling information |
| writestate | allows the state of a task to be modified |
| control | permits the task to be started, stopped, or destroyed; it also allows the scheduling paramters to be changed |

The access mode associated with message channels is:

| | |
|---|---|
| transmit | allows a message channel connection and the transfer of packets to or from the channel |

There are no access modes associated with dummy objects, although a protected type manager could use the undefined modes to implement a type that required no special representation but did require access control.

Amber does not permit explicit null access terms on access control lists. If access should be granted to all members of a group save one, it would be too hard for the owner to ensure that the excluded user could not obtain access through the use of another principal. Null access terms could be supported, perhaps as a configuration parameter, if a particular community had a need for them.

## PROCEDURAL ACCESS CONTROL

In many cases, system-provided access modes such as 'read' and 'write' are insufficient: finer granularity is required. For instance, a user might wish to use a segment as an electronic mailbox. Other users must have 'write' access to the mailbox in order to place messages into it, but this also allows them to delete messages placed there by others. A more complex method able to distinguish between adding messages and deleting message is required.

The second access control mechanism provided by Amber, called @i(procedural access control), allows a protected subsystem called a type manager to interpret all accesses to an object. Any primitive object may be @i(sealed) by a domain, making it a protected object. Normal kernel operations such as mapping a segment or transmitting over a message channel may not be performed on a protected object. Only the sealing domain (or its agents) may unseal an object and obtain access to its primitive representation.

By restricting direct access to a primitive object to a single domain, type managers which enforce arbitrary protection policies may be constructed.

Type managers may place any interpretation on the unused access mode bits in access control lists of sealed objects. The presence of a mode bit should allow more operations to be performed than if the mode bit were not present, but the kernel does not require this. In the mailbox example, separate mode bits for reading messages, adding messages, and deleting messages could be used.

A user task can only manipulate a protected object by communicating requests to a type manager task executing in the sealing domain (or in an agent of the sealing domain.) This is commonly done by forwarding a packet containing a capability to the sealed object and a description of the requested operation over a message channel. Any results are transmitted back in the other direction.

For common operations such as accessing I/O devices or mailboxes, type managers will be created and started by the system. Communications with these type managers will be made over message channels whose names are publicly known. For less frequent operations, or operations whose applicability is not system-wide, a user must create an instance of the type manager. The kernel provides a gate, invoke_domain_, to provide this service.

The invoke_domain_ operation is fairly complex. After checking that the caller has 'invoke' access to the type manager's template domain, the kernel creates a copy of the template domain inside the caller's executing domain. The copy is accessible only to the caller, and only with the modes the caller had on the template. Access to the copy may not be altered, and hence access may not be increased. Typically, the template domain will contain a task and a message channel whose names are known to the invoker. The caller has 'control' access (but not 'status' or 'writestate') to the task, and 'transmit' access to one end of the message channel. After the invoke_domain_ operation is complete, the task may be started and communication over the message channel established. Finally, requests may be sent over the message channel as in the system type manager case.

These type manager tasks are roughly equivalent to Hydra encapsulated procedures [HYDRA] or CAP protected procedures [CAP]. Unsealing of a protected object is the operation which performs rights amplification.

Object sealing has other uses in addition to implementing procedural access control. Sealing an object creates a link which contains [up to] the sealer's access to the object, and this link may be used to pass that access from one domain to another without altering the objects ACL. A practical application of this ability is the implementation of a printer server: on most systems this is a highly privileged process, or one that requires explicit entries on the ACL of the object to be printed. In Amber, a user can pass a seal to the printer server and revoke it when the segment has been printed.

Amber's procedural access control mechanism is not without its problems. Prominent among the problems are efficiency, the implications of control over type manager tasks, and object deletion.

Protected domains, because they require separate processes, are inherently expensive. Amber tries to mitigate the expense by allowing a single instance of a type manager to process many requests and by storing type managers as templates, thus obviating the need to fully

initialize them when they are invoked.

Traditionally, the atomicity of operations on protected objects is preserved by preventing them from being interrupted. File system operations are protected because a process may not be stopped while it is executing in the kernel. On Multics, user interrupts are not processed while a process is executing in an inner ring, thus ensuring the integrity of operations on protected objects such as mailboxes. Because Amber type managers are user tasks, no such protection is offered. A user with 'control' access to a type manager task may stop and destroy that task while it is in the middle of an operation. There is no defense against this: type managers must be designed to mark objects inconsistent before beginning an operation.

A user may also delete the protected object without the intervention of the type manager. Amber provides no mechanism to prevent this, so type managers must be aware of the possibility that their objects could disappear.

## MANDATORY ACCESS CONTROL

At the current time, Amber does not provide for any form of mandatory access control. Future implementations will do so, and the design and implementation of the kernel have made provisions for object labeling as specified in the DoD "Orange Book" [TCSEC]. In addition, covert channel identification has been done.

## STORAGE SYSTEM

Amber segments are the objects used to store data, similar to files on other systems. However, Amber's single-level storage system allows segments to be directly mapped directly into a domain's virtual address space, making traditional file I/O calls unnecessary. In this model, pages of files and directories exist @i(either) in main memory @i(or) on secondary storage, never in both. Changes made to a page in main memory are made to the file; no 'write' system call is necessary. In addition, all tasks accessing a given file see the identical data, modifications made by one program are immediately visible to all observers.

The Amber storage system encompasses both main memory and secondary storage devices. These parts are combined by an operation which creates a mapping between a collection of disk records and a range of virtual addresses. Once this mapping is established, the actual location of data is immaterial, and largely unobservable from outside of the kernel.

When a segment is mapped into an address space, it becomes accessible through the hardware load and store instructions to a range of addresses characterized by a base address and a size. The base address may either be specified by the user or assigned by the kernel; the size must be specified by the user, but it is rounded up to the nearest 2**I 64Kbyte boundary. The same segment may be mapped at multiple locations within a single address space, and may be mapped in multiple address spaces. As a result, both object code and data can be easily shared.

The mapping operation only establishes a connection between a range of addresses and a segment. It does not initiate any read operations, nor will it create disk records or memory pages to hold the segment if sufficient ones have not already been allocated. In fact, there need not even be any spare disk quota at this point.

The Amber philosophy is not to allocate any resources or perform any operation until it is actually needed, on the presumption that many resources and operations will never be required. In the case of segments, pages are not created or disk records allocated until the page is first written, and existing records are not brought into main memory until they are actually referenced.

The most basic unit of storage is the page, defined to be 4K bytes long regardless of the underlying hardware architecture. This is the unit of both main memory and secondary storage allocation, and the granularity of maximum length and storage quota enforcement. Segments are constructed as trees of a 16-page (64 Kbyte) long object called a @i(segmentito).

This structure was originally dictated by the hardware segmentation and paging architecture of the S-1 Mark IIa processor (SMA5), but it has been ported to the S-1 AAP processor which has no paging hardware other than a software-writeable translation lookaside buffer.

### Virtual Memory Management

Amber virtual memory management consists of three levels of abstraction: the segment, the automatically-allocated segmentito (AASEGO), and the demand-paged segmentito (DPSEGO). A DPSEGO is a collection of 16 pages where each page may either be in core, on disk, or non-existent. An AASEGO is an enhanced DPSEGO where the non-existent pages may have special properties. The AASEGO level also enforces disk quota usage and the maximum size of the file. A segment is a collection of AASEGOs.

When AASEGOs are created, each page points to a read-only, zero-filled page. Normally, there is one such page per system, but some cache architectures may require that there be several. An attempt to write the zero page results in a fault which is satisfied by allocating a new disk record and a page frame to hold it. Once allocated, the record is made known to the underlying DPSEGO which is responsible for moving the page between disk and memory. A record is not deallocated until the segment is deleted or explicitly truncated. Incidental zeroing of a page will not cause the corresponding record to be removed.

Pages are brought into memory in response to faults when the corresponding part of the address space is referenced. They are evicted from memory only when the page frame they occupy is required in order to satisfy another page fault. Pages are chosen for eviction from the set of all pages in the system using the clock algorithm approximation of the LRU algorithm. There is no concept of a per-task working set where a single user pages only against himself.

Because this paging behavior is not always desirable, Amber provides ways for individual programs to give advice on memory reference patterns. A call is provided to asynchronously start disk reads on pages which are likely to be used in the near future; another call informs the kernel paging daemon that some pages are unlikely to be used again and are good candidates for being evicted. For time-critical processes which cannot tolerate the page fault penalty, a kernel call whch 'wires' pages into main memory is provided.

To minimize the chance of losing data on system crashes where no recovery is possible, there is a kernel daemon which periodically writes modified pages out to secondary storage devices. There is also a kernel call which performs this function upon request. Neither of these methods evict the page from memory.

## Address Space Management

The remaining portion of virtual memory management is concerned with translating virtual memory addresses into the appropriate pages within segments. Several kernel modules are responsible for maintaining the tables used to perform these translations: the Active Segment Table (AST), the User and Kernel Address Space Descriptor Segments (UDSEGS and the KDSEG), and the tables which contain the non-terminal nodes of segment trees.

The AST contains an entry for each segment which has pages currently in main memory, and for each domain which has an inferior domain with pages in memory. These are known as active segments. The root directory is always active, and always has an entry in the AST.. The AST entry (or ASTE) is the repository for such information as date_time_used and date_time_modified, as well as pointers to the structures describing the segment tree. Segment trees have a branching factor of 16: there are 16 pages in a SEGO; if there are more than 16 pages in a segment, an intermediate structure called a SEGOTE is added which points to 16 SEGOs. Depending on the size of the virtual address space, this tree can grow arbitrarily tall. A height of 4 allows for a 8 Gbyte address space.

Operations on portions of segments are executed by mapping the operation over the subset of SEGOs which contain the requested range. This centralizes the tree-maintenance operations including reading the tree structure from disk and growing or shrinking the tree height.

The Amber kernel address space is shared amongst all tasks on the system, and is distinct from the address space of any of those tasks. A special kernel segment holds the KDSEG, which contains an array of pointers to SEGOs when those are in core, to the ASTEs of the containing segments if the SEGOs are not in core, or nil if the memory is not mapped. Note that any segment mapped in the kernel address space must be active.

Each domain that contains one or more active tasks has an associated address space. Each address space has a UDSEG in the kernel address space. Like the KDSEG, the UDSEG contains pointer to ASTEs or SEGOs as appropriate. Unlike the KDSEG, the UDSEG entries also have chains which connect address spaces which are sharing segments, that is two or more address spaces which both map the same segment. This allows all address spaces to be updated when a fault is satisfied or when a segment is evicted from the AST.

At this point, it is useful to explain what happens when a segment is touched for the first time. The segment fault handler consults the task's containing domain to obtain the UID pathname and allowable access modes. Starting with the Root's ASTE, the handler proceeds down the UID path looking for ASTEs for the intermediate domains, causing them to become active if they are not. A pointer to the ASTE is placed in all of the SEGO slots in the UDSEGs of all address spaces which map this segment for the entire range of addresses mapped to this segment. The segment fault handler then returns.

The reference is retried and faults again. This time the SEGO fault handler is invoked. It calls the segment tree function mapper to read the data for a single SEGO in from disk and build the SEGOTE/SEGO tree as necessary. A pointer to the SEGO is placed in the proper entry in the UDSEGs. The SEGO fault handler returns.

The reference is once again retried. This time, a single page table word in a SEGO is examined. The page is created or brought in from disk as necessary by the page fault handler. The reference will now succeed.

## The Physical Storage System

The physical storage system is the permanent repository for Amber objects on secondary storage devices. It is essentially a direct image of the virtual memory system with all of the pointers replaced by disk record indices. An Amber Storage System Partition (SSPART) consists of three disk areas: the Volume Table of Contents (VTOC), the Volume Segment Management Table (VSMT), and the disk records.

Each segment has a VTOC entry, called a VTOCE, which contains the attributes of a segment and pointers to inferior levels of the segment tree. The inferior levels (non-terminal nodes of the segment tree) are kept in VSMT entries (VMSTEs). Direct VSMTEs contain record indices of the disk pages making up the segment. Indirect VSMTEs contain VSMT indices of subordinate indirect or direct VSMTEs below them in the tree.

As follows from this description, individual segments must reside entirely in one SSPART, which may not be larger than a single physical storage device. A single device may contain multiple SSPARTS. A storage system volume consists of an arbitrary number of SSPARTS.

The following figure depicts the on-disk and in-core versions of a segment tree. In the in-core version, not all of the segment is present in main memory.

**An Active Segment with In-core Pages**

| ASTE | SEGOTE 1 | SEGO 1 |
|---|---|---|
| ptr to segote 1 | addr of vsmte 4 | page frame 1 |
| addr of IND VSMTE 2 | ptr to sego 2 | page frame 2 |
| ptr to segote 3 | tr to sego 3 | disk addr 3 |
| nil | | |
| nil | | |

**A segment on disk**

| VTOCE | INF VSMTE 1 | DIR VSMTE 4 |
|---|---|---|
| IND VSMTE 1 | DIR VSMTE 4 | disk addr 1 |
| IND VSMTE 2 | DIR VSMTE 5 | disk addr 2 |
| IND VSMTE 3 | DIR VSMTE 6 | disk addr 3 |
| invalid | | |
| invalid | | |

From these diagrams, it is clear that an ASTE is nothing more than an encached VTOCE, a SEGOTE is simply an encached indirect VSMTE, and a SEGO is an encached direct VSMTE.

## Directory Control

The portion of the kernel responsible for managing domain contents is known as directory control, which indicates that is deals solely with the directory portion of a domain's function.

These modules reduce the domain abstraction to their underlying segments, and operate

on the domain contents as simple data. They implement operations such as object creation and deletion, access control list manipulation, property list operations, and address space mapping.

## TRAFFIC CONTROLLER

The Amber traffic controller was designed to be an efficient multi-processor scheduler providing inexpensive tasks and fast task switching between user, kernel, and real-time tasks. Additional goals included tunability, extensive metering, and allowance for site-specific scheduling policies. The traffic controller manages all tasks and schedules them to run on individual processors. It also provides interprocess communication primitives for tasks executing in the kernel and procedures which create, delete, and alter the parameters of tasks.

The traffic controller is split into two portions along the lines suggested by Reed [REED]. The lower level, called the wired traffic controller (WTC), supports the virtual processor abstraction. WTC is needed to support virtual memory, and therefore cannot depend on virtual memory. The upper level, called the paged traffic controller (PTC), uses virtual memory to provide support for a large number of tasks.

The WTC level can be viewed as a single-processor scheduler which only considers a subset of the total tasks active on the system: namely, those tasks which have been @(loaded) on that processor. Loading a task on a processor entails reading its state into wired memory (WTC is not allowed to take page faults) and binding it to that processor such that it may only run on that processor. This makes better use of per-processor data caches and allows gang scheduling and device-specific I/O drivers to guarantee execution on a particular CPU.

WTC uses a simple round-robin scheduling algorithm within different priority levels. Tasks give up the CPU when their scheduling quantum has expired (a timer interrupt), when they are preempted by a higher priority task becoming ready to run (a preempt interrupt), or when the voluntarily relinquish control of the processor by blocking.

The PTC level is a per-system scheduler. It is paged to make tasks less expensive consumers of valuable system resources and multiplexes more expensive loaded slots amongst arbitrarily many cheap, unloaded tasks. PTC also performs interprocessor load-leveling as it is the entity which performs the task load and unloads on specific CPUs. It is also responsible for implementing the site's scheduling policy.

### Tasks

A task is a single, separately executable, control point in a program. Each task consists of processor state, including processor status word, register contents, etc., scheduling parameters, and a pointer to a containing domain where its address space may be found.

The task state is kept in two containers: a small Task Control Block (TCB) and a larger Task Wired or Paged Storage (TWORP). The TCB is present in wired memory whenever the task is active. This allows the WTC level to deliver interprocess communication wakeups even when the task is not loaded. The TCB contains minimal information: the task's ID, current state, list of pending wakeups, several flags, and a pointer to its TWORP. All other information about a task is kept in the TWORP As its name implies, this is the storage which

is moved between wired and paged memory when the task is loaded and unloaded.

There are six states that an Amber process can be in at any given time. A task is @i(inactive) when its state resides solely in a domain and is not known to the traffic controller. A task may be made active but not allowed to run, for instance while a debugger is examining its state, by placing it in the @i(stopped) state. Once a task is allowed to run, it may either be @i(ready), @i(waiting), @i(runnable), or @i(running). A task is ready when it is waiting for an event in the relatively distant future, usually more than a minute. A ready task is usually unloaded. A waiting task is paused waiting for an event expected to occur in the near future, such as the completion of a page read. A runnable process is able to run, but is awaiting the availability of a processor, which are all being used by running tasks.

### Kernel Interprocess Communication

The WTC module provides a wait/notify protocol which is similar to Reed and Kanodia eventcounts [REED AND KANODIA]. Events are represented by arbitrary event ids. A task requests to be notified of the occurrence of an event by calling the addevent entry in WTC and then goes to sleep by calling the wait entrypoint. When the event occurs, the task signaling the completion of the event calls the notify routine which wakes up all tasks which requested notification of that event.

Notify is the equivalent of the advance operation, although the events being reported are unordered. Wait is the same as await, although a task is not allowed to wait on multiple events in the kernel, nor on the n'th occurrence of a given event.

In contrast to semaphores, this scheme has only one writer which makes it easier for distributed processing and in solving the confinement problem. Since a task performing the P operation cannot observe other tasks which are also performing P, covert channels are eliminated. Since enqueuing and waiting on an event are separate events, you can perform the simultaneous-P semaphore operation[REED AND KANODIA].

The method used inside the kernel to wait for short events is represented by the following code fragment:

```
loop
  wtc.addevent (event_id);
exit if <event_has_occurred>;
  wtc.wait_no_timeout ();
exit if <event_has_occurred>;
end;
```

Note that the event is checked between the calls to addevent and wait in order prevent race conditions. Once addevent has returned, the process will not go to sleep if the event is posted before the task reaches the wait entry.

The traffic controller also provides interprocess wakeups, an additional simple communications mechanism for use both inside and outside of the kernel. As part of its wired state, each task as a small set of wakeups which are monitored by the kernel. When a task receives

a wakeup, it is interrupted unless it has masked wakeups. In the latter case, all wakeups that were received while the task was masked will be delivered as a group when the task unmasks. If more than one identical wakeup was delivered while the task was masked, knowledge of all but the first one will be lost.

Interprocess wakeups are used by the broadcast event and message channel mechanisms. The kernel also provides clock-time and CPU-time alarms which send wakeups to tasks when they time out. Possession of 'control' access to a task allows wakeups to be sent directly to that task. Finally, tasks which control I/O devices may receive wakeups when that device sends an interrupt to a processor.

### Interprocess Communication

The simplest and highest bandwidth IPC mechanism supported by Amber is shared memory. Multiple tasks, including at least one writer, map the same segment. Changes to the segment are instantly visible to all readers. Loop-waiting for a memory location to change is inefficient, however, so broadcast events are frequently used for synchronization.

Broadcast events are very similar to Reed and Kanodia event counts as used in Reeds VP2, and can easily model semaphores in the same fashion as the WTC-level wait-notify protocol does. An event is a tuple consisting of an Amber object and an arbitrary event identifier. Unlike the wait-notify kernel protocol, a task can declare that it is waiting for an arbitrary number of events. The declaration includes a wakeup to be sent when the event occurs. When a task broadcasts the occurrence of an event, all tasks which declared the event will receive the wakeups they requested.

Amber requires listen and broadcast permission to the object associated with an event to declare and broadcast that event. This security mechanism addresses the confinement problem as regards interprocess communications.

### Message channels

Message channels provide packet-oriented bidirectional pathways between a pair of tasks. Each channel consists of two nodes, called the user node and the server node, which are functionally symmetric. Each node may be controlled by only one task at any given time. When a task sends a packet into one node, it becomes available to the task in control of the other node.

Tasks must have 'transmit' access to a node in order to establish a connection and to send or receive information over it. The kernel also provides a way for a task to identify the owner of the task which controls the other node. Thus, a server task can perform interpretive access control if appropriate.

A message packet is a variable-length record which corresponds to the amount of information sent in a particular send operation. Packets are delivered in the order in which they are sent. To send a packet, a task passes a pointer and a length into the kernel, which then adds the send request to a queue. When the receiver issues a read request, the next available packet is copied directly from the sender's address space into the receiver's.

As far as the kernel is concerned, tasks need not wait at the send or receive operations for the transfer to occur. If the send request is issued before the read request, the sender may continue to execute. In this case, the actual transfer will occur when the receiver issues the read request. If a read request is issued first, the transfer will occur when the send request is made. In either case, both sender and receiver will be notified (by sending a interprocess wakeup) when the transfer is complete.

The kernel provides additional mechanisms to aid in implementing higher-level protocols: a task may request a wakeup when a packet becomes available, and it may interrogate the kernel as to the presence of pending send requests and the size of the first packet in the queue. More complicated protocols can easily be synthesized by user routines. For instance, a remote procedure call protocol can be implemented by a procedure which sends a request packet, waits for it to be read by the server, issues a read request for the response, waits for the response, and returns any return parameters included in the packet received from the server.

## I/O

As part of the Amber philosophy, most I/O operations are performed outside of the kernel. The functions of the traditional file system are subsumed by the single-level storage system described above. Many peripherals can be controlled through the use of special segment objects. Access to these devices can be performed by user tasks executing in privileged domains outside the kernel. User tasks wishing to use those devices communicate with the appropriate server using message channels. This technique is used for terminals, tape drives, and network connections.

For security and reliability reasons, storage system volumes are not directly accessible outside of the kernel. This requires all of the disk driver software to be inside the kernel.

## Resource Management

Scarce resources, such as page frames, page tables, and virtual processors, must be multiplexed among many users with as small overhead as possible. Amber has a kernel daemon charged with maintaining a free stock of every resource. When any task notices that the stock for a particular resource is below a certain level, defined by a tuning parameter, it notifies the appropriate daemon. In nearly all cases, the daemon will be able to make more resources available before the free stock completely disappears. This mechanism allows tasks to share resources without having to block waiting for those resources to become available, except in infrequent times of heavy load.

## Dynamic Reconfiguration

Amber allows physical resources to be added and deleted from the system at any time. Processors, memory, and peripherals (with some exceptions) may be added or deleted without having to reboot the system. Exceptions would include the disk containing the file system root and the system console.

## Modularity

Software engineering practices dictated the use of modularity in the construction of the Amber kernel to simplify the kernel, ensure correctness, and aid in maintainability and portability. The division of labor between modules was done based on abstraction-creating design decisions rather than on sequences of execution. It is better to separate the abstractions of virtual memory and virtual processors into separate modules because of the different data they process, than it is to have them in the same module because common operations such as page fauls operate upon both: modules are viewed as controlling things, not processes.

Amber modules can be placed in a partially-ordered graph; there are no circular dependencies. If module A depends on (calls) module B, then neither module B nor any module that it depends on can ever depend on module A. A practical consequence of this requirement is that a kernel module which depends on virtual memory need not be memory resident, because no module which is incapable of taking page faults will ever call it. A further advantage is deadlock avoidance: the locking hierarchy is isomorphic to the module hierarchy, and no locks held by different modules can ever be involved in a deadlock.

To be consistent with this view of the modularity hierarchy, traps and exceptions (e.g., page faults) as subroutine calls on behalf of the user, and so at the top of dependency diagrams. This allows, for instance, the directory control module to take page fault, during which the trap handler will invoke the access control module which depends on the directory control module.

Overall, this strict modularity rule reduces the overall complexity of the kernel by eliminating complex, hard to debug interdependencies between the many parts of the kernel.

The other benefits of modularity include being able to prove system correctness by proving that each module correctly fulfills its documented contract. No programs outside of a module are permitted to depend on any side effect of a module not documented in its published contract. Because only code within a module is allowed to directly access data belonging to that module, bugs can be more easily localized. Finally, those portions of the kernel which are machine dependent are placed in their own modules. When Amber is ported to a new hardware platform, only those modules need to be changed to implement that contract on a different machine. All other modules need only be recompiled.

## Reliability

The Amber kernel was also designed to be reliable, though not fault-tolerant. Many modules in the Amber kernel continuously check for error conditions. When one is detected, redundant information is used to correct the error and ensure that it is written to disk. In particular, the contents of domains are closely examined during all directory-manipulating operations.

It is true that most of these errors will cause the system to crash to limit the spread of the error. However, Amber is designed to reboot quickly without the time-consuming file system consistency checking done by such things as the Unix *fsck* program. A particularly damaging crash, such as a power failure where the contents of kernel tables in main memory are lost, will not prevent a quick reboot. Amber maintains a partial consistency which allows benign

errors on permanent storage, but prevents malicious ones.

When the kernel needs to obtain a free store of disk records, it consults the free record table on the disk. Before any records are placed in use, the entire free stock obtained is marked as used on the disk. Thus, if the system crashed at this point, there would be records marked used which were in fact free. However, there cannot be records marked free which are in fact used. This is the meaning of partial consistency. An online program known as the volume scavenger can recover those records which are inadvertently marked as used.

In general, the kernel performs operations with careful attention to order, so that the crash recovery procedure, emergency shutdown, is able to figure out which operations need to be completed and which may safely be aborted.

### Implementation and Portability

The Amber kernel consists of 130 separate compilation units totaling about 60,000 lines of Pastel code, not including device drivers. Pastel is a local dialect of Pascal, extended to support separate compilation and systems programming constructs. About twenty of the source modules contain machine-dependent code.

Amber has been implemented on two members of the S-1 family of uniprocessors. It was originally written for the S-1 Mark IIa, and many of the design decisions, particularly the virtual address translation scheme, were heavily influenced by that machine's architecture. A port to the S-1 AAP was completed, and was awaiting completion of the AAP hardware. A port to a MIPS R2000-based processor was underway before the S-1 project ended.

*Technical Information Department* · Lawrence Livermore National Laboratory
University of California · Livermore, California 94551