A Multi-threading Architecture for Multilevel Secure Transaction Processing*

Haruna R. Isa
United States Navy
4251 Suitland Rd.
Washington, D.C.
blaze@cts.com

William R. Shockley
Cyberscape Computer Services
1885 Franklin Street
Lebanon, OR 97355
Bill_Shockley@compuserve.com

Cynthia E. Irvine
Computer Science Department, Code CS/Ic
Naval Postgraduate School
Monterey, CA 93943-5118
irvine@cs.nps.navy.mil

Abstract

A TCB and security kernel architecture for supporting multi-threaded, queue-driven transaction processing applications in a multilevel secure environment is presented. Our design exploits hardware security features of the Intel 80x86 processor family. Intel's CPU architecture provides hardware with two distinct descriptor tables. We use one of these in the usual way for process isolation. For each process, the descriptor table holds the descriptors of "system-low" segments, such as code segments, used by every thread in a process. We use the second table to hold descriptors for segments known to individual threads within the process. This allocation, together with an appropriately designed scheduling policy, permits us to avoid the full cost of process creation when only switching between threads of different security classes in the same process. Where large numbers of transactions are encountered on transaction queues, this approach has benefits over traditional multilevel systems.

1 Introduction

Commercial transaction processing (TP) applications generally depend upon a substantial set of services, often provided in the form of *middleware* or as an operating system extension. For the last several years we have been investigating topics relating to the design of a high-assurance security kernel and Trusted Computing Base (TCB) supporting TP requirements. This paper focuses on our approach for providing two related services, transaction queuing and scheduling of multi-threaded processes, that are particularly difficult in a high-assurance security environment.

Aside from the general convenience of assigning each distinct transaction to its own thread, the adoption of multi-threading permits two distinct enhancements to overall throughput [2].

First, it is often the case that the processing of each distinct transaction is quite stereotyped. Instead of creating a new process for each incoming transaction (typically a very expensive operation) a single, multi-threaded process is set up when the system is initialized, with all of the code needed to perform the transaction loaded in advance. An incoming transaction is then queued and allocated to the next free thread of the pre-existing process. Numerous process creation and deletion operations are thereby avoided.

Second, when a thread in execution blocks (e.g., to wait for an I/O or logging operation to finish), the switch to a ready thread in the same process often incurs a much lower performance penalty than a complete context switch to a different process.

The use of multi-threading for systems operating in multilevel secure (MLS) mode has generally been dismissed as inappropriate because the sharing of an address space between two threads handling transactions at different access classes appears to be essentially insecure. If the application program being executed contained an implementation flaw or a Trojan Horse[1], high sensitivity data could easily be transferred to a low sensitivity container, since both data and container would be available in the same address space.

*The views expressed in this paper are those of the authors and do not reflect the official policy or position of the Department of Defense or the U.S. Government.

[1] According to [16] the term was coined by D. Edwards.

The implementation of MLS thread-oriented applications on traditional MLS systems requires processes at each access class for each task in a workflow. Many potential MLS workflows are not restricted to three or four sensitivity levels, but must address the gazillion problem [8], where support of a very large number of sensitivity levels is required. When faced with numerous previously unencountered sensitivity levels, the penalty for process creation may be high.

Our research goal, originally suggested by Shockley, was to investigate the practicality of exploiting specific features built into the Intel 80x86 architecture[2] to obtain some of the benefits of multi-threading in a high-assurance MLS environment. Specifically, the CPU architecture supports two distinct, independently addressable descriptor tables per process [5, 6, 7]. From the perspective of descriptor-based security controls, this means that each process sees two distinct address spaces, not one. Both virtual address spaces are further subdivided into four hardware privilege levels by the CPU architecture. These hardware privilege levels are used to organize all code and data into four tamperproof execution rings using standard techniques [18]. The work presented here is for an architecture and design that can lead to an implementation.

The remainder of this paper is organized as follows: Section 2 gives an overview of traditional transaction processing systems and explores various approaches to transaction processing in an MLS environment. The features of the Intel 80x86 processor family that will be utilized to achieve our security objectives will be described in Section 3. An informal presentation of the security policy to be enforced and the overall system model appears in Section 4. Our security architecture is described in Section 5. Section 6 presents a discussion of the architecture, lessons learned, and areas for future research. Our conclusions are presented in Section 7.

2 Background

In this section we provide an overview of traditional transaction processing and the security-relevant features of the Intel 80x86 family of processors that will be used in our architecture.

2.1 Traditional Transaction Processing

The term *transaction* is unfortunately overloaded in many papers on the subject. We will use the term *transaction* to refer to an application-defined unit of work, realized in our architecture as an enqueued message requesting that a particular type of processing be performed. The message and its frame provide the input and contextual data needed by the application to perform the work, and by our TCB to manage its processing.

In a typical queued transaction processing (TP) system, as illustrated in Figure 1, queues are abstractions around which transaction processing is organized.

The actual execution of a single transaction in our architecture is performed by an execution environment called a task. A transaction is scheduled for execution by placing it on an input queue associated with the task. Whenever a task requests new work, a scheduler is invoked that either provides the task with a handle for the next transaction to be executed or blocks it, letting another task run. As the work is performed, the application or middleware code being executed by the task will typically invoke additional blocking or non-blocking I/O requests. When the work is complete, the task commits or aborts the transaction's effects, and waits for the next transaction. Synchronization between tasks (e.g., if arranged in a pipeline) is implicit in the semantics of the enqueue and dequeue operations. The transaction itself is not actually deleted from the queue until the task has signaled that processing is complete by requesting more work. If the task generated work product in the form of one or more new transactions, the transactional semantics implemented by a higher-level subsystem, the Transaction Manager (TM), will (in effect) perform the deletion of the old transaction and enqueuing of any new transactions as a single, atomic, recoverable event. To the schedule manager, what is seen during the course of execution is a series of requests for blocking or non-blocking I/O operations.

Transaction processing systems must also allow for conditional workflow. Consider the example shown in Figure 3. After TP task A has completed processing a transaction, it may enqueue a new transaction on the queues for both TP tasks C and D. Alternatively, the context of the transaction may result in an enqueue to only one subsequent queue, either that of TP task C or that of task D.

Provision of an efficient high assurance, multilevel secure transaction processing system imposes requirements for isolation of information at various sensitivity levels. Isa [9] has explored support for MLS TP using a variety of traditional MLS architectures. These were deemed insufficient to satisfy goals for management of labeled information, data consistency and support for the gazillion problem. As an alternative to traditional multiprocessing approaches to MLS, we describe a design that uses the hardware features of the Intel 80x86 processor family to support high assurance MLS TP.

[2] Here *80x86* refers to the Intel 80286, Intel386, Intel486, and Pentium processors. Of course, the newer, faster members of this family are the intended target platforms. Intel386, Intel486, and Pentium are trademarks of Intel Corporation.

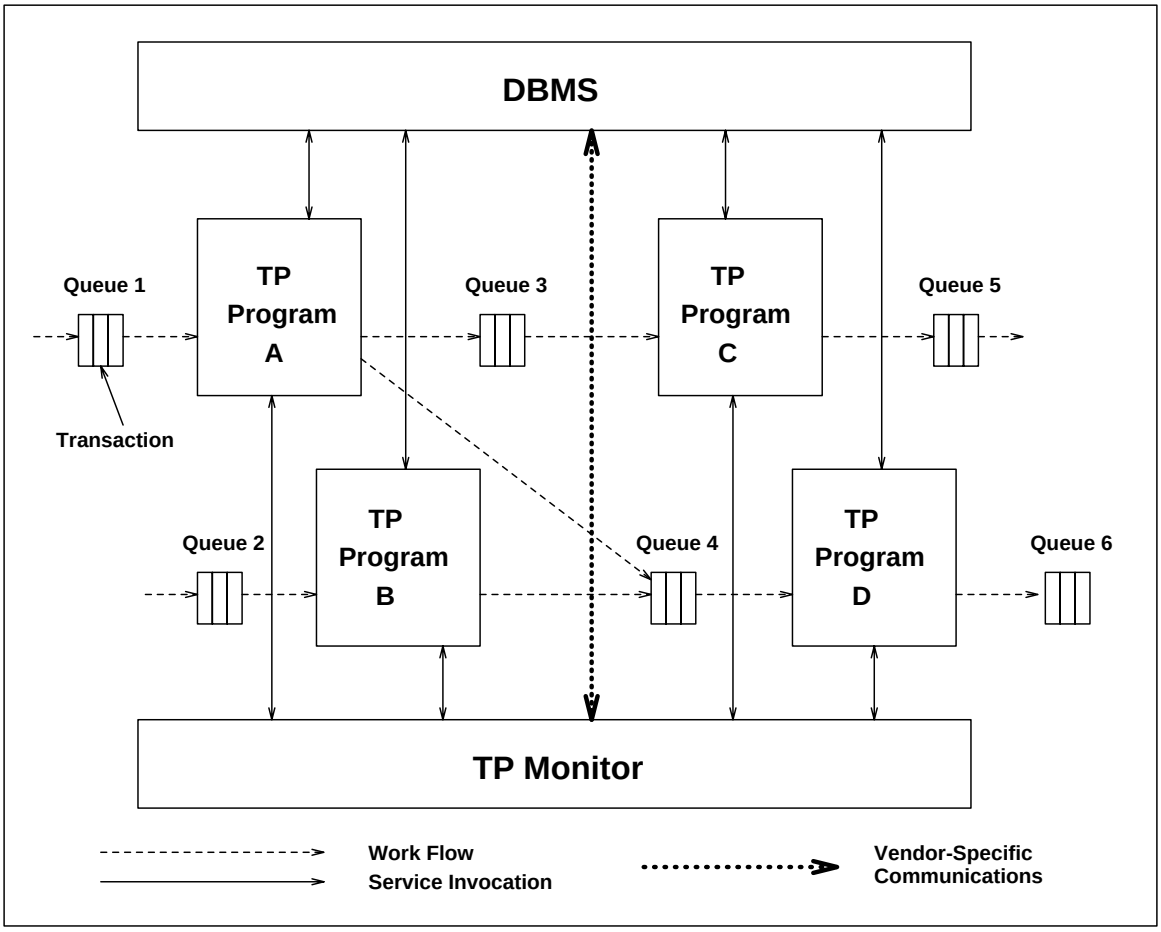

Figure 1. Traditional Transaction Processing

2.2 Intel Processor Features

Our MLS TP system is targeted toward the Intel 80x86 series of microprocessors. This series of processors implements protection features, such as descriptor-based segmented memory and multiple privilege levels [5, 4], that are extremely useful in creating a kernelized, MLS system [16, 17].

Memory is segmented and all access is via descriptors. Descriptors reside in two system descriptor tables; the *local descriptor table* (LDT) and the *global descriptor table* (GDT). A memory address consists of a selector and an offset. The selector is an index into one of the descriptor tables (which describes the segment being accessed) and the offset is the location within the segment that is being accessed. Besides describing the physical address and limits of the memory segment, the descriptors also contain the access modes allowed (read or write), the type of segment (code, gate, etc.), and the privilege level of the segment.

Four hardware privilege levels (HPL) are provided. HPL 0 is the most privileged and HPL 3 is the least privileged. A privilege level is associated with every segment in the address space. Privilege level information is also maintained as part of the hardware-recognized segment descriptors for the GDT or LDT segment. It is a principle of our architecture that the segment descriptors always reflect a kernel-maintained constant ring number attribute for each segment. A *current privilege level* (CPL) is maintained as part of the execution state vector built into the CPU. The CPL is the privilege level associated with the code segment currently being executed. The CPL is used by the hardware to make memory access determinations.

When a process attempts to access memory via a descriptor, the hardware performs several checks. If the CPL is less privileged than the HPL of the segment being accessed or the access is for a mode not allowed (e.g. attempting write access to a read-only segment), a hardware protection fault results. Additionally, since all memory accesses must be

via selectors into one of the two descriptor tables, a process cannot access any segment for which a descriptor has not been loaded. The creation of descriptors and loading of the descriptor tables can only be accomplished by code running in HPL 0.

The address space of a process is all the segments in the GDT and LDT at the CPL or less privileged. A special descriptor, known as a gate, is used to allow less privileged processes to call more privileged routines. The gate descriptor has a HPL equal to or less than the processes that are allowed to call it. The gate provides a controlled entry point to a code segment at a higher privilege level. It can be used to contribute to the tamper resistance of a security kernel and a TCB.

The Intel 80x86 series microprocessors provide two modes of operation; real and protected mode. Real mode is provided for backward compatibility and does not use any of the memory protection required for multitasking, much less MLS, operating systems. Protected mode, however, provides hardware enforcement of memory accesses based upon privilege levels, available descriptors and descriptor attributes.

3 Domain Architecture

The distributed kernel will be associated with hardware privilege level 0 (HPL 0). It will use the three other hardware privilege levels to create a ring abstraction supporting the traditional notion of subjects as $< process, domain >$ pairs. All memory management requests will be serviced by the kernel.

Our architecture uses one descriptor table, the global descriptor table, to support the abstraction of a process. A process is then defined by a GDT image together with an execution point: i.e., a $< CPUstate, GDTimage >$ pair, where the current privilege level is associated with the CPU state. For a multi-tasked application, the intended use of the GDT is to hold the descriptors of all of the segments that are shared by all of the tasks associated with the process. These segments include the entire distributed kernel, which are all assigned to HPL 0, the HPL 1 segments of the process queue management package, the task management package, and any per process application level code and/or data. It is assumed that segments available to less privileged rings are virtualized such that per process code and data is read-only when the execution state of the CPU is in a less privileged ring. From the perspective of an application in execution, the code and data addressed in the GDT looks like ROM: it can be read, but not modified or deleted. The application is not permitted to introduce new descriptors into the GDT.

The second descriptor table, the local descriptor table, is used to support the abstraction of a task. (We have avoided naming this entity a "thread" because we expect that a genuine TP application may choose to introduce a third level of "multi-threading" in to pick up even more throughput.) A task is defined by a GDT image, an LDT image, and an execution point: i.e., a $< CPU_state, LDT_image, GDT_image >$ triple. A given GDT image is shared among all of the tasks of a given process, but each task has its own LDT image.

The intended use of the LDT is to hold all per task data (e.g. its stack, linkage information, working variables, buffers, etc.), generally with read-write access. Thus, as illustrated in Figure 3, to an executing task the segments addressed in the LDT always look like RAM. Subject to the security constraints imposed by the kernel, the task may freely create, delete, or make known segments in this address space just as an ordinary process would on a conventional kernelized system.

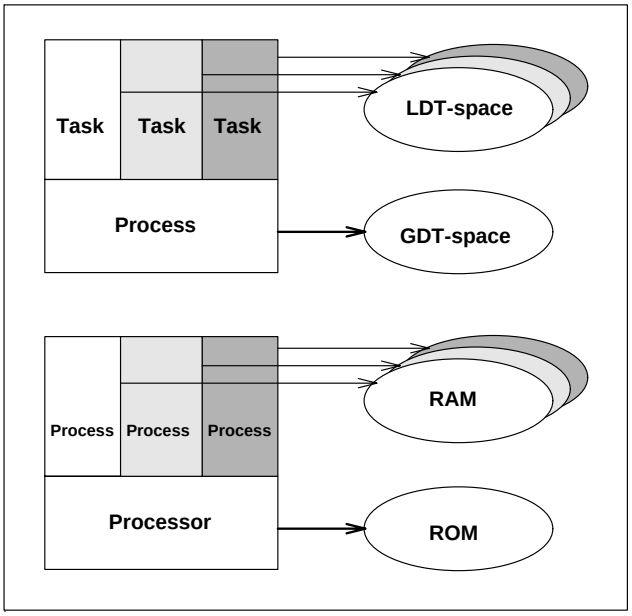

Figure 2. All tasks are managed by the process, which isolates tasks by allocating separate LDT-spaces to each task. This is analogous to the management of processes by a kernel, which virtualizes the processor and allocates RAM to each process, thus isolating them from other processes. All tasks share a common GDT-space as all processes share a common ROM.

4 Security Policy and Model

Security policies relate users and information. Here we will briefly describe the security policy to be enforced by the TP system, first in general terms and then as a technical policy, i.e., in the context of a computer system where the policy is applied to subjects and objects.

The mandatory policy can be related to corporate or government directives an example of which is DoD Directive 5200.28 [12]. Simply stated, the policy declares that only authorized users may have access to sensitive information. User authorization is conveyed through clearances, while information sensitivity is denoted by its classification. Each user will be accorded an access class which will be a combined secrecy class and integrity class. The secrecy class describes the sensitivity of information that the user is trusted not to disclose to unauthorized users, while the integrity class reflects the trust placed in the user to disallow unauthorized modification or contamination of information. More formally stated, our mandatory policy has two components each of which is characterized by a read property and a write property.

Secrecy Component

Read property: Only if a user's secrecy class is greater than or equal to the secrecy class of the information may a user read that information

Write property: A user must insure that information is stored such that it will be inaccessible by users who do not possess the requisite authorization to access the information.

Integrity Component

Read property: A user may not read low integrity information that could potentially corrupt high integrity information to which that user has access.

Write property: A user may only store information at or below the user's integrity class.

When describing a security policy in terms of a computer system, we refer to subjects and objects rather than people and information. Formally, a subject is an active entity operating on behalf of the user and is described as a $< process, domain >$ pair. Here we define a process to be a program in execution that is completely described by its current (and single) point of execution and its address space. The current context of the process, found in the CPU state, describes the domain. It is a subset of the address space and may be represented by a ring number [18, 17]. We can see that a process may have several subjects, but only one subject within the process can be executing at a time. Subjects will be characterized by an access class that will be at or below the clearance of the user, i.e., some subset of the user's total set of authorizations. The subject's domain of access will be further restricted by its ring number.

The kernel creates the notion of processes. Each process manages its tasks and is a trusted subject, i.e., a subject having two security classes: a read-class and a write-class. The read-class defines the highest access class that the subject can read, while the write class places a lower bound upon the access class that the subject can write.[3] In theory, the kernel can set up a process with a particular read-class/write-class range and different processes within our system could have different ranges. The access classes of all tasks managed by a process must fall within its read-class/write-class range. Tasks will correspond to untrusted subjects, in this case, subjects with a degenerate read-class/write-class range. In both cases, subjects are defined by an $< execution_point, ring, LDT_image, GDT_image >$ quadruple. The kernel mediates all accesses to segments by the process and insures that all segments within the address space of the processes are within the read-class/write-class range of each process.

The distributed kernel manages its own code and data. All kernel segments are associated with the most privileged ring, Ring 0. Only when the processor is executing in HPL 0 is it possible for the kernel's address space to be modified.

Processes are Ring 1 subjects. Only these subjects may successfully call the kernel to introduce or delete Ring 1 segments from the GDT. Thus they are responsible for the management of GDT-space. The intent is that this will be done only when the system is booted (i.e., emulating ROM) or under administrator control (i.e., emulating "reburning" the ROM.) This approach insures that tasks will be unable to use the GDT as a resource supporting covert channels.

To subjects in Rings 2 or 3, the GDT-space is static. It follows that in order to be readable by all Ring 2 or 3 subjects of a task, a segment in the GDT must be "process-low" in sensitivity. One can see that this is precisely what is needed: what one finds in here is "secrecy-low" application code (read only) in rings 2 and 3, task management and task scheduling code in ring 1, and kernel code in ring 0. Of course "internal databases" for the kernel and task scheduler are made unreadable by rings 2 and 3. It is also important that even ring 2 and 3 data in the GDT be unwritable to avoid channels (even if the policy would allow it to be written by some task.)

This allocation of segments to rings creates an architecture that stands up to the litmus test established by the reference monitor concept [1]: that the reference monitor be

[3] Using these notions, we see that a *trusted subject* is defined to be a subject that is not constrained by the confinement property within its write class-to-read class range.

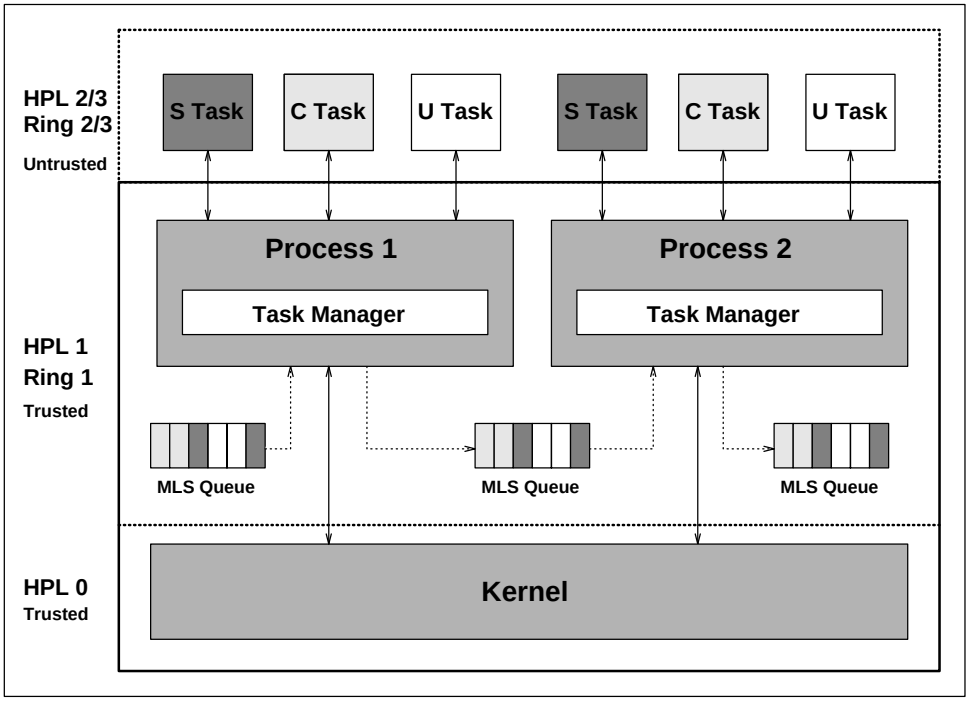

Figure 3. Multilevel Secure Transaction Processing Architecture. Each transaction is in a separate segment (not illustrated). The multilevel queues are ring 1 internal databases, respectively, each contains pathnames for the segments that comprise their members.

tamperproof or self protecting and that it be non-bypassable. The hardware protection mechanisms of the Intel 80x86 processors permits us to create protection domains that can be managed using hardware features and we benefit from the assurance provided by carefully implemented hardware mechanisms.[4] Our architecture is conceptually simple, thus lending itself to analysis for assurance. The use of segments permits us to begin with hardware objects that may be augmented with additional attributes by the kernel. These attributes include the segment's access class and ring number and, from the perspective of less privileged subjects, may be considered immutable. Although less privileged subjects are able to request the creation and deletion of segments from both the GDT (emulating ROM) and the LDT (emulating RAM), all modifications ultimately are mediated by the kernel.

The policy for LDT-space is similar to that associated with other high assurance MLS systems which rely upon segmentation to provide process isolation and thereby isolation of access classes [16, 17]. Each subject in Ring 2 is provided with a separate LDT. When the task manager schedules a task the LTD descriptor register will be loaded with the value of the LDT of the task to be executed. The Intel hardware insures that no segments are accessible to the ring 2 subject other than those visible via the LDT and the GDT. As we have noted earlier, not all of the segments contained in the GDT will be accessible by ring 2 or 3 subjects. Since each subject in ring 2 is single level, its address space will be restricted to segments for which the subject meets the requirements of the model described earlier.

We accomplish task switching by using a call gate, ensuring that the Intel-designed context switching mechanism is invoked to enter the more privileged Ring 1 domain where LDT-to-task mapping is managed. This means that the cost of switching from one task to another is roughly the same as the cost of switching from one process to another. Our architecture does not save time by substituting task for process switches, but by reducing the total number of switches performed.

5 System Design

Our three-tier architecture is illustrated in Figure 3.

[4] It is recognized that hardware mechanisms may contain exploitable flaws [19]; however, flaws are less numerous than in comparable commodity software [11].

At the core of the system is the security kernel. The kernel manages memory, processes and its eventcounts [15]. The processes are each designed to handle a specific transaction type. There would be as many processes as there are transaction types. The processes are multi-level and consist of a task manager and process queue manager. The task manager within the process creates and manages single level transaction processing tasks that actually process the transactions. The intent is to have a covert channel-free model and then to not introduce storage channels during implementation.

The incoming transactions of a given type would be sent to the process of the correct type and then to the task of the appropriate level. The tasks are the outer layer of the system and are single level entities which process a given transaction type of a given classification.

Figure 4 illustrates the system layering and dependencies.

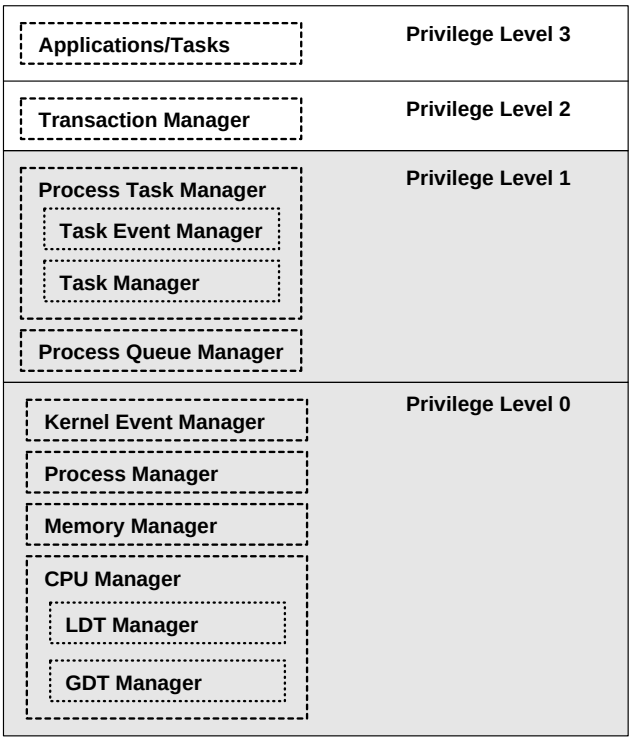

Figure 4. Selected System Layers.

5.1 Security Kernel

The security kernel consists of several distinct subsystems. There is a CPU Manager, which manages the various local descriptor table (LDT) images (used by the individual tasks within each process space), the global descriptor table (GDT) and adding or removing segments from any of the descriptor tables. There is a Memory Manager which is responsible for allocating and deallocating memory. There is a Process Manager, which is responsible for creating, scheduling and destroying processes. Finally, there is a Kernel Event Manager, which manages eventcounts used by the processes and the MLS queues. These eventcounts are used for synchronization by all modules in the system.

5.1.1 CPU Manager

All code and data manipulated by a task is contained wholly in the LDT and all code and data used by the kernel, the MLS queues, the processes are contained wholly in the GDT. As such, a task switch involves only an LDT switch. The CPU Manager can be viewed as being logically divided into two distinct subsystems; one which manages the LDT and one which manages the GDT. Whenever a new task is created, a new LDT image is created to hold the descriptors for its address space. Associated with each LDT image is an identifier which is made available to the less privileged ring.

GDT Manager

The GDT component of the CPU Manager is responsible for managing the global descriptor table. The specific functions it provides include:

create_gdt - create a new GDT image with the specified access class range

destroy_gdt - destroy the specified GDT image

add_to_gdt - add a segment to a specified GDT image

remove_from_gdt - given a valid GDT selector, the associated descriptor is removed from the GDT and the current process GDT image.

switch_gdt - switch GDT segments

Associated with each process is a handle to a GDT image. On a process switch the process portion of the GDT must be saved to the currently running process' GDT image storage segment while the GDT image of the new process must be restored to the process portion of the GDT.

The call to switch a GDT segment comes from the Process Manager which actually performs the process switch.

The GDT Manager is supported by one data structure:

GDT Database used to keep track of the GDT images for the various processes

LDT Manager

The LDT component maintains a database of LDT images. Each of these LDT images is associated with a given TP task. However, the mapping between a given LDT images and a specific task is not maintained by the kernel, but instead is maintained by the Task Managers. The LDT component provides functions to:

create_ldt - creates and returns a new LDT image at a specified access class

destroy_ldt - destroys an LDT image

add_to_ldt - given a valid segment and access class, the descriptor for the segment is added to the current LDT

remove_from_ldt - given a valid selector for a segment in the current LDT, the associated descriptor is removed from the LDT.

switch_ldt - make a given LDT image the current LDT

When a task requests the addition of an item to its LDT, the Task Manager brokers the request. It adds an access class and the task's LDT image number to the arguments and invokes the kernel. The kernel LDT Manager validates the access class of the specified LDT image against the access class provided by the Task Manager and the range designated for the process before adding the designated entry. Note that the Task Manager must provide the actual access class of the task as the kernel only knows the range of access classes managed by the Ring 1 subject.

When a task manager creates a new task, a request is made for the kernel to create a new LDT image. It is then up to the task manager to associate with that LDT image a particular task.

The LDT component is supported by one data structure:

LDT Database used to keep track of LDT images and their addresses

5.1.2 Memory Manager

The Memory Manager manages virtual memory. It is responsible for those functions that allocate memory for the segments that are currently in use. It provides information on every segment currently in memory. It enforces the mandatory security policy basing its decision upon a comparison of process ranges, requested attributes, and segment attributes. A key attribute cached for each segment by the Memory Manager is the access mode of the segment. The Memory Manager functions include:

make_known - determines whether a segment can be moved into the virtual memory and, if so, provides a handle to the segment

terminate - removes the segment from the virtual memory

swapin - insert a segment into the process virtual address space

swapout - delete a segment into the process virtual address space

list_memory - return the attributes associated with a segment

The Memory Manager is responsible for one data structure:

KST used to keep track of all segments currently in the address space of the process and their attributes (security label, descriptors, etc.)

5.1.3 Process Manager

The Process Manager, as the name implies, manages the processes, which equate to different transaction types. The Process Manager implements several operations to manage processes.

create_process - creates a new process and adds it to the ready list

switch_process - suspend the current process and run the next ready process

destroy_process - destroys a process

change_process_status - moves a currently running process to either ready or blocked status

get_current_process - returns an identifier for the current process

Associated with each process is a global descriptor table image. These GDT entries are where each process stores the code and data the task manager needs to perform its functions. Upon a process switch, this GDT image is switched to reflect the new process' entries. The process manager also performs scheduling of processes.

The Process Manager is supported by one data structure:

Process table holds the process information (including GDT image) needed to manage processes

The process table allows the kernel to map a given process to its GDT image. A process switch also results in a switch of the contents of the process portion of the GDT. Processes can be in one of three states: ready, running or blocked on an event.

5.1.4 Kernel Event Manager

The Kernel Event Manager provides eventcounts and sequencers, called tickets, for use by modules in Ring 1: processes and the MLS queues. Sequencers are monotonically increasing integers (perhaps with a modulus), initialized to 0. Eventcounts are also initilaized to 0 and monotonically increasing. The *advance* operation increments the value of the eventcount and indicates the highest ticket number that should be serviced.

The Kernel Event Manager provides the following functions:

k_create_evct - returns a new kernel eventcount with a specified access class range

k_destroy_evct - deletes a specified kernel eventcount

k_await - causes a wait on a specified kernel eventcount value

k_advance - advances a kernel eventcount

k_read_evct - inspect the current value of a specified kernel eventcount

k_ticket - get a kernel ticket

The MLS queues use kernel eventcounts and sequencers to keep track of the number of items in a queue. A call to *get_work* from an empty queue becomes a wait call on a kernel eventcount and leads to a process change. When new items are added to this queue, the associated kernel eventcount is advanced at which point the blocked process is moved from the blocked process list to the ready process list making it eligible to be scheduled. Process scheduling is determined by the transaction flow through the MLS queues.

The Kernel Event Manager is supported by one data structure:

KED the kernel event database, is used to track the values of the eventcount-sequencer pairs

5.2 Process Queue Manager

The Process Queue Manager is the entity of the system which managers the MLS queues. It is layered between the process (Task Manager) and the kernel. The Process Queue Manager provides functions which allow processes to:

create_queue - create an MLS queue

destroy_queue - destroy an MLS queue

enqueue - enqueue an item on an MLS queue

dequeue - dequeue an item from an MLS queue

copy_to_queue - make a copy of an item and place it on an MLS queue. This permits an item to be enqueued as a "carbon copy" to an MLS queue.

get_work - get and item from an MLS queue (without dequeuing it)

Initial input transactions are added to the system and are put on the MLS queues (usually by type) by some trusted input process. This process would enqueue an incoming transaction on the MLS queue associated with the process that handles transactions of that type. The transactions are of varying access classes. The Task Manager maintains single level tasks which process the transactions from the MLS queue (based on access class).

The Task Manager makes a *get_work* request to the Process Queue Manager while providing a preferred access class. The Task Manager seeks to keep running the same TP task (at a specified access class) as long as work exists for that task to process. This minimizes task switches and provides maximum throughput of transactions through the system. The Process Queue Manager will return an item at the requested access class or an item at a different access class if: (1) there were no items at the requested access class or (2) there is a transaction with a higher priority than the next transaction of the requested access class. The access class of the item returned by the Process Queue Manager determines whether the current task remains running or whether a new task will have to be scheduled.

A call to *get_work* from an empty queue blocks (being translated into a wait call on a kernel eventcount). The process of the calling Task Manager would thus be blocked and a new process scheduled by the kernel. When the blocked process has an item enqueued to it (which also involves a call to advance the appropriate eventcount), the process will be moved to the ready list and could be scheduled to run.

All the functions of Process Queue Manager are exported to the Task Manager.

The Process Queue Manager is supported by one data structure:

PQD the process queue database, is used to keep track of information about the various MLS queues

5.3 Task Management

Here we describe the two task management subsystems pertinent to our security architecture. They are the Task Manager, which is responsible for resource allocation to the tasks. The other subsystem is the Task Event Manager, which implements eventcounts to be used for inter-task communication and scheduling.

5.3.1 Task Manager (TKM)

This module is responsible for resource allocation to tasks.

Each process contains a Task Manager which manages the single level TP tasks for each transaction type. The Task Manager creates, schedules and destroys the individual tasks. The Task Manager is very similar to a typical process manager, however, instead of managing a process table as a kernel might, it manages a per process task table. It manages a database that contains an identifier for each task, per task state information, a handle to the transaction in the process queue and an eventcount associated with the task. Effectively, the each task is associated with a queue of length one.

The Task Manager is responsible for managing the single level tasks within each process space. The operations supported by the Task Manager include:

create_task - create a new single level task

destroy_task - destroy a single level task

activate_task - schedules a new task

switch_task - changes from one single level task to another

The Task Manger exports the following functions

tm_get_work - returns a transaction for processing by the current task

The Task Manager implements scheduling of single level tasks within its process space. Besides directly managing the TP tasks, the Task Manager also serves as an intermediary for task access to kernel memory management functions (add/remove from LDT). The Task Manager maintains a data structure that associates each task with its access class and LDT image. When a tasks attempts to add or remove a segment from its LDT image, the Task Manager passes the request on to the kernel after adding the access class of the requesting task and the identifier for its LDT image.

Additionally, the Task Manager interfaces with the Process Queue Manager to retrieve and insert items into MLS queues on behalf of the tasks. When a task makes a request for a new transaction, the Task Manager makes a call to *get_work* providing the appropriate access class as a parameter. If the returned item is of the requested access class, the Task Manager returns it to the current task which continues to run. If the Process Queue Manager should return an item of a differing access class, the Task Manager will suspend the current task and begin running the task of the access class associated with the returned transaction. This will result in making the segment containing the transaction known in the address space of the task by requesting a call to the kernel to add the segment to the LDT of the task. An attempt to *get_work* from an empty queue will block and not return until there are items available in the queue.

The Task Manger will insure that tasks blocked on I/O calls are run after I/O completes. In general, tasks will execute asynchronously within a process: the relative execution speeds of two tasks cannot be determined *a priori*. It is anticipated that tasks may need to synchronize their activities in order to communicate.

It is expected that there will be a per-process task-pool of tasks for the most common security classes associated with that process. This pool would be administratively configured prior to system boot and the tasks associated with this bank would be non-deletable. In addition, the administrator would allot resources to dynamically create and delete tasks at less common security classes. At runtime both common and uncommon classes would be treated as a cache. Tasks in the task-pool would never be removed from the cache, while those associated with less common security classes would be managed using a standard caching algorithm, such as least recently used.

Our belief is that a "pool" of intelligently selected, statically allocated tasks will significantly reduce the first cost described in the introduction, i.e., for environments with a small number of known access classes, it avoids unnecessary process or task creation. Where the gazillion problem arises, adding a pool of dynamically allocated, cached tasks managed in ring 1 should permit significant savings if the cache is large enough and there is good *locality of class*. Even in the worst case, where incoming work is a random selection from a gazillion possible classes, the overhead of full process creation is avoided, as one need only set up a new LDT for the new access class. The GDT, which is common to all tasks, is known *a priori* to be readable to the new task, and does not need to be revalidated.

It is important to note that the advantage of our approach is not the avoidance of a task switch. That, in fact, is required to insure no flow of information through the registers or stack used by tasks at each sensitivity level. The advantage comes through the support of a large number of sensitivity levels within a process and the consequent avoidance of process creation to support transactions at unusual access classes. The kernel-level scheduler might only run another process when all of the work on the queue of the current process has been exhausted.

The Task Manager is supported by one data structure:

TD the task database, used to maintain information on the current single level tasks being managed by the Task Manager

5.3.2 Task Eventcount Manger (TEM)

This module implements eventcounts and sequencers (viz. tickets) to be used for synchronization and scheduling of

tasks. These are not kernel eventcounts, but Ring 1 abstractions. The Task Eventcount Manager exports two synchronization primitives the following operations:

t_create_evct - returns a new TEM eventcount with a specified access class range

t_destroy_evct - deletes a specified TEM eventcount

t_wait - causes a wait on a specified TEM eventcount value

t_advance - advances a TEM eventcount

t_read_evct - inspect the current value of a specified TEM eventcount

t_ticket - get a TEM ticket

The advance operation will advance the specified eventcount by 1 and may cause tasks waiting on that eventcount to be awakened. The await function causes return to the calling task to be delayed until the eventcount attains a particular value.

The Task Event Manager is supported by one data structure:

TED the task event database, is used to track the values of the eventcount-sequencer pairs

5.4 Tasks

The tasks in the outermost layer of the architecture are the untrusted applications. It is these tasks that actually do whatever work is required by the transactions. Each task is at a single level and might coexist with copies of itself at different sensitivity levels within the same process space. However, through Task Manager manipulation of the LDT images, each task has its own distinct address space (perhaps sharing a read-only code segment.) Since the task manager sets up an LDT image for a task when it is created, a task switch simply requires changing the current LDT.

Our objective is to maximize transaction throughput. Thus the scheduling policy entails completely processing all entries in a given queue before moving on to the next queue. So, in general, a process continues to execute so long as transactions remain in the MLS queue it is waiting on. Likewise, within the process, a task continues to execute so long as transactions of the appropriate access class remain in the MLS queue of its controlling process. In this manner, we minimize the number of task switches within a process and minimize the number of processes switches within the kernel.

5.5 Input/Output

The currently defined architecture does not yet incorporate specific functions supporting the direct use of I/O devices by Ring 2 or Ring 3 tasks. This omission reflects our initial research focus on the queue abstraction and multitasked processes. It is possible, however, to sketch our general intent with respect to I/O services.

Our Obuective is to encourage the creation of Ring 3 transaction servers that view input and output in terms of a set of logical queues, representing streams of incoming and outgoing events and data. Ring 2 middleware (not elsewhere discussed in this paper) maps logical queues to actual queues under system management control.

Usually, Ring 2 device server processes (device demons) will allocate and manage physical devices. Consider, for example, a multilevel printer required to print jobs of various classifications. We would allocate such a printer to a dedicated print spooler process and use the multitasking architecture described in this paper to structure the work. In Ring 1, we would augment the Task Manager to not only remove work from the input queue and assign it to a task, but to manage as well device-dependent functions associated with changes of current device class (e.g., printing a separation banner, reinitializing the printer to a known state, etc.) Most of the device-dependent code would reside in Ring 2 and execute in a single-level environment. As Ring 2 must invoke Ring 1 functions to do actual I/O, Ring 1 can intervene when necessary to do additional security-critical functions, such as page labeling.

As the printer example shows, our architecture still requires the inclusion of device-specific code within the TCB. It does, however, provide a useful framework for separating security-critical device-dependent functions from non-security-critical functions.

5.6 Transaction Management Requirements

We allocate transaction management (TXM) and other middleware system software to Ring 2. In particular, we would expect most transactional resource managers (e.g., a DBMS) to be placed in this ring. Support for specific programming environments, including multithreading (e.g., a Java interpreter) would also exist in this ring.

Notionally, Ring 2 manages any system-wide discretionary access controls (DAC). This allocation interposes a hardware-enforced protection boundary between "system software" and "application." Our architecture does not preclude the use of memory isolation techniques to isolate threads. However, our assumed hardware base provides only one LDT, which we must use either in Ring 1 for task isolation, or in Ring 2 for thread isolation. We chose to use it for task isolation because this suited our particular

research goals. For systems not required to support MAC, one could choose instead to use the techniques we discuss to protect individually per user threads by giving them distinct LDT images.

We have not yet undertaken a detailed architecture for the Ring 2 middleware. In this section, we report therefore conclusions from selected design studies undertaken in support of the Ring 1 architectural effort.

5.6.1 Concurrency control

We continue to provide eventcounts and sequencers in Ring 2 for both interprocess and intertask synchronization and control. Without them, it would be impossible for the Ring 2 programmer to design device demons or shared resource managers. However, Ring 1 also exports the useful abstraction of a queue, and provides a framework for custom-programmed multilevel queues. For those choosing to use this abstraction, concurrency control for the queue is implicit: that is, one is guaranteed that a given element will be provided to one, and only one, task instantiation, and queue functions are already "task and process safe" without explicit user-level synchronization.

In Ring 3, TP application programmers will typically use the TP services provided by Ring 2 rather than explicit synchronization.

5.6.2 Write-up

Lattice-oriented mandatory policies allow subjects to write to objects of access classes that dominate that of the subject itself. Typically, providing for the unrestricted use of such a capability eases attack by denial of service, while prohibiting the capability altogether precludes the construction of otherwise valid application systems.

Our architecture for Ring 1 supports two kinds of "write-up" for untrusted Ring 2 and 3 tasks:

1. One may (if one knows its name) advance a higher-level eventcount. This potentially unblocks the tasks or processes (if any) waiting on that particular advance. There is no improper backward channel because the low-level advancer is not shown whether any waiting processes existed. In effect, this provides a primitive "signal up" capability: one informs the higher level that some event has occurred.

2. One may copy a queue element onto a higher level input queue. (The "copy" operation takes as arguments both the name of the queue, and the access class wanted for the copy). Return from this operation shows that "the system" has successfully enqueued the element but provides no indication whether any higher-level task received it. This service is a higher-level version of "advance up": the low-level subject may, in effect, attach a message to an event notification.

To prevent a back-channel while preserving control of all queue elements, Ring 1 must always do something useful with the copied element. To block resource allocation channels, we expect that ordinary user queues will have static size constraints. If Ring 1 cannot deliver the copy to the designated queue, it will instead log the event. (The log entry will, of course, include a complete copy of the undelivered element.) In the unlikely event that the system operator has failed to replenish the log media, an option will be available to suspend processing until the media is replenished. A Ring 1 demon will periodically awaken to attempt to "redeliver" such transactions. In any event the system operator will be kept informed of the count of undelivered transactions (if any) existing in the log. Of course, Ring 1 does not inform the low-level subject invoking the copy operation which alternative occurred, as that would be a channel.

5.6.3 Deadlock control

The design of an adequate system for managing deadlocks in a multi-level, distributed TP environment would seem, at first glance, to require a significant collection of trusted code within the TCB because, by definition, deadlock is a global condition that may involve multiple nodes. We argue that this is not, in fact, the case.

The key observation is this: deadlock is defined as a "circular wait": i.e., one has a set of tasks arranged in a cycle, each waiting for a lock to be released that is held by the next.

However, it is simple to prove that any such circular wait is single-level (i.e., all lock-holding transactions are of the same level). The proof follows directly from the fact that the set of access classes is partially ordered. The analog of "waiting for a lock" is "waiting for an eventcount to reach a prescribed value" and of "holding a lock" is "not having advanced an eventcount yet". Tasks are only allowed by the TCB to wait on eventcounts of the same or lower class. Suppose a task is "waiting down". It cannot, then, be part of a deadlock cycle! Suppose the contrary; then there must be cycle in the set of access classes. Since no such cycle can exist, a task that is "waiting down" is not part of a circular wait. To be clear, the task **holding** the wait may be part of the circular wait, but the task waiting down is not: it is blocked by the deadlock, but not participating in it.

It follows that if one looks for deadlocks one access class at a time, one finds them all. This means that one can schedule for a given access class a "single-level" deadlock maintenance routine and be assured, when it completes, that any deadlocks at that class have been cleared. There is no need to synchronize deadlock maintenance among levels.

5.6.4 Scheduling Policy

The module design described previously carefully distinguishes between synchronization and scheduling. Synchronization is implicit in the semantics of an eventcount "wait": a processing entity (task or process) has finished its assigned work and will not be resumed if or until more is available. Upon return from a "wait," the task or process may be sure there is more work available on its input queue.

The scheduling issue is: given a set of ready entities (processes, tasks), which one should be resumed? The algorithms used in Rings 1 and 2 collectively define a scheduling policy. We have therefore carefully placed these algorithms in independent submodules so that we can test different ideas.

Our initial design is deliberately simple, emphasizing maximum throughput. With static allocation of prebuilt applications, it is anticipated that obvious timing channels can be appropriately monitored. Although we have provided a field for a transaction-specific priority, our initial scheduler ignores it. We expect to investigate more sophisticated scheduling policies in later projects.

For many TP applications, the key goal is maximum throughput. This reflects an economic environment where one is paid for each transaction processed. Since costs are fixed (i.e., per node) any additional transactions that can be processed in a unit of time are "pure profit." A small increase in throughput is leveraged into a larger profit margin. We have therefore chosen, in our initial design, to emphasize throughput as a goal.

Our approach has been to choose a scheduling policy that avoids process or task switches whenever possible. Therefore, the kernel chooses to resume the current process (if not blocked for I/O) as long as there is work queued for it. The Process Queue Manager schedules work within a process batched by access class, so that the task scheduler can resume the current task as long as there is work at the same level available for it.

We have also tried to design the system to reduce ring crossings, as these are more expensive than ordinary procedure calls. In particular, by re-implementing intertask eventcounts in Ring 1 (rather than using kernel eventcounts) we avoid calls to the kernel during task scheduling and synchronization.

Clearly, in most applications we must provide for priority processing. There will also be system requirements for high-priority processes (those serving high-performance I/O ports, for example.) Accordingly, we have provided both for a "priority" attribute for kernel processes, and for individual transactions. We have not yet added priority to our scheduling policy, as our initial intent is simply to demonstrate the use of LDTs to support multi-tasked processes.

6 Discussion

In this section we compare our architecture with some alternatives, relate some lessons learned, and discuss future research possibilities.

6.1 Comparison with Alternative Architectures

There are several possible alternatives to the MLS transaction processing system we present. The first is to create a system composed of several single level TP systems. Each single level system within this conglomeration would process transactions of a given level. Some type of trusted interconnection device, such as the Naval Research Laboratory Pump [3], would be used between these single level systems. This solution becomes unwieldy, if not impractical, when faced with the gazillion problem and a complex, conditional workflow.

The second alternative would be to implement a TP system on top of an already developed MLS system, such as the XTS-300 [13] or the GTNP [14]. However, the need to maintain separation between transactions of differing levels would require the creation of a separate process not only for each workflow node, but for each access class as well. Although this approach might be successful for a few access classes and task types, it fails when confronted with the gazillion problem. In addition, since this traditional architecture is not optimized for TP, performance enhancements such as the task-pool and the queues would be hard to implement without significant modification to the underlying system.

6.2 Lessons Learned

At the start of this effort, we believed that support for MLS transaction processing would require a security kernel design significantly different from traditional systems, e.g. [10, 14]. In particular, it was thought that multilevel queues would have to be managed by the kernel. In our final design, the queues are managed outside of the kernel, but within the TCB. Queue management would have added considerable complexity to the kernel, so the resulting design supports minimization objectives for a high assurance reference validation mechanism.

6.3 Further Research

In a previous section, we suggested that additional performance benefits could be achieved by superimposing thread management on tasks. We would place the thread management software in Ring 2, providing services to application threads in Ring 3. The thread manager would be similar to the Task Manager, simplified by the fact that all

threads within a task execute at the same mandatory security class. A natural choice is to associate a new thread with each incoming transaction. Traditional discretionary access controls (DAC) could then be enforced with respect to objects visible at the Ring 2 interface.

Providing isolation between threads belonging to different users remains a problem, primarily because the CPU architecture supports only two virtual address spaces. We haved defined (in general terms only) the following possible approaches:

1. Ring 2 maintains per thread images for a section of the LDT and swaps them in and out when the thread changes – in effect, providing a ”virtual” second LDT.
2. Ring 2 provides a closed, Java-like interpretive environment (not necessarily high-performance, but attractive to some potential users).

7 Conclusions

We have described a system architecture intended to support multilevel transaction processing. Hardware protection features of the Intel 80x86 family of processors are used in an innovative design which supports tasks at multiple levels within a single process. The design starts with a relatively conventional security kernel executing in hardware privilege level 0, which is responsible for enforcing the mandatory access control policy, process management, memory management, multilevel queue management, and, within each process, enforcing a ring mechanism.

In Ring 1 a task manager executes as a trusted subject that creates virtual single-level queues and manages tasks. Tasks having frequently encountered access classes may be administratively assigned to a static task pool, while an appropriate scheduling algorithm can be used to maximize throughput for less common access classes. This design allows us to avoid creation of new processes to handle tasks at new access classes.

The tasks in Ring 3 may contain a conventional thread manager and may include either software or hardware-based support for the enforcement of discretionary access control policy. The latter is achieved through the further virtualization of the segments in the local descriptor table so that a thread is only able to access segments for which it has appropriate discretionary access. Alternatively, to achieve the former, Ring 3 can contain the application itself, which yields the kind of DAC enforcement typical of current commercial products.

8 Acknowledgements

We would like to thank the sponsors of the Naval Postgraduate School Center for INFOSEC Studies and Research for their support of the ongoing system security program. We also thank Nelson Irvine for his comments on our manuscript.

References

[1] J. P. Anderson. Computer Security Technology Planning Study. Technical Report ESD-TR-73-51, Air Force Electronic Systems Division, Hanscom AFB, Bedford, MA, 1972. (Also available as Vol. I,DITCAD-758206. Vol. II, DITCAD-772806).

[2] P. Bernstein and E. Newcomer. *Principles of Transaction Processing for the Systems Professional* . Morgan Kaufmann Publishers, Inc., San Francisco, 1997.

[3] J. Froscher, M. Kang, J. Mcdermott, O. Costich, and C. E. Landwehr. A practical approach to high assurance multilevel secure computing service. In *Proceedings 10th Computer Security Applications Conference*, pages 2–11, Orlando, FL, December 1994.

[4] J. L. Hennessy and D. A. Patterson. *Computer Architecture a Quantitative Approach*. Morgan Kaufmann, San Francisco, CA, 1996.

[5] Intel. *Intel Architecture Software Devloper's Manual, Volume 1: Basic Architecture*. Intel Corporation, Santa Clara, CA, 1997.

[6] Intel. *Intel Architecture Software Devloper's Manual, Volume 2: Instruction Set Reference* . Intel Corporation, Santa Clara, CA, 1997.

[7] Intel. *Intel Architecture Software Devloper's Manual, Volume 3: System Programming Guide* . Intel Corporation, Santa Clara, CA, 1997.

[8] C. E. Irvine. A Multilevel File System for High Assurance. In *Proceedings 1995 IEEE Symposium on Security and Privacy*, pages 78–87, Oakland, CA, May 1995. IEEE Computer Society Press.

[9] H. Isa. *Utilizing Hardware Features for Secure Thread Management*. M.S. thesis, in preparation, Naval Postgraduate School, Monterey, CA, December 1998.

[10] P. A. Karger, M. E. Zurko, D. W. Bonin, A. H. Mason, and C. E. Kahn. A VMM Security Kernel for the VAX Architecture. In *Proceedings 1990 IEEE Symposium on Research in Security and Privacy*, pages 2–19. IEEE Computer Society Press, 1990.

[11] T. Lewis. Joe Sixpack, Larry Lemming, and Ralph Nader. *IEEE Computer*, 31(7):107–109, 1998.

[12] National Computer Security Center. *Department of Defense Trusted Computer System Evaluation Criteria*, DoD 5200.28-STD, December 1985.

[13] National Computer Security Center. *Final Evaluation Report of HFSI XTS-200*, CSC-EPL-92/003 C-Evaluation No. 21-92, 27 May 1992.

[14] National Computer Security Center. *Final Evaluation Report of Gemini Computers, Incorporated Gemini Trusted Network Processor, Version 1.01*, 28 June 1995.

[15] D. Reed and R. Kanodia. Synchronization with eventcounts and sequencers. *Communications of the A.C.M.*, 22(2):115–123, 1979.

[16] J. H. Saltzer and M. D. Schroeder. The Protection of Information in Computer Systems. *Proceedings of the IEEE*, 63(9):1278–1308, 1975.

[17] R. Schell, T. F. Tao, and M. Heckman. Designing the GEMSOS security kernel for security and performance. In *Proceedings 8th DoD/NBS Computer Security Conference*, pages 108–119, 1985.

[18] M. D. Schroeder, D. D. Clark, and J. H. Saltzer. The Multics Kernel Design Project. *Proceedings of Sixth A.C.M. Symposium on Operating System Principles*, pages 43–56, November 1977.

[19] O. Sibert, P. A. Porras, and R. Lindell. The Intel 80x86 Processor Architecture: Pitfalls for Secure Systems. In *Proceedings 1995 IEEE Symposium on Security and Privacy*, pages 211–222, Oakland, CA, May 1995. IEEE Computer Society Press.