PROCEEDINGS

OF THE

FOURTH SEMINAR ON THE DOD COMPUTER SECURITY INITIATIVE

NATIONAL BUREAU OF STANDARDS
GAITHERSBURG, MARYLAND

AUGUST 10 - 12, 1981

DAVID L. GOLBER

SYSTEM DEVELOPMENT CORPORATION

THE SDC COMMUNICATIONS KERNEL

The SDC communications kernel is intended to support secure communications applications, such as secure front ends and terminal access systems. It is a minimal operating system, capability-based, and has a basic structure that we hope will ease the problem of formal specification and verification. [1]

The kernel is oriented towards support of communications systems in that it offers extensive facilities for interprocess communications. Because of its restricted aim, it does not support dynamic changes, such as creation of processes.

The SDC communications kernel has been operational for a number of years in an ARPANET-like DoD system. We feel that the capabilities and speed of the kernel are well-adapted to such a system, and are competitive with other systems not using a kernelized architecture.

The kernel was developed under the primary direction of Dr. Richard Mandell. The design and coding were done by Karl Auerbach, David Clemans, and Jay Eaglstun.

1.0 <u>In General</u>

The SDC communications kernel is a descendant of the UCLA Data-Secure Unix [2] operating system. The SDC communications kernel remains similiar to the UCLA kernel in the following major areas:

a. The SDC kernel is a minimalized operating system. It is a small amount of code which exists to provide environment and services to processes. The processes may be regarded as "application" code; there is no partitioning of the kernel itself into processes. The kernel is the only code in the machine which accesses hardware features of the machine such as memory protection registers, device registers, etc. In a PDP 11/70, the kernel

----

[1] The question of verification is discussed at the end of section 2.

[2] "Unix" is a trademark of Bell Laboratories.

consists of exactly that code which runs in hardware "kernel" mode, the privileged mode of the machine. Processes run in non-kernel hardware mode.

b. The SDC communications kernel is intended to be a verifiable operating system. That is, it should be possible to formally state the services and protections that it supplies and to formally prove that it does what it is intended to do and no more.

c. It is generally felt that operating system code which is interruptable is very hard to verify. Therefore it is preferrable for a verifiable operating system to run with interrupts completely locked out. This is the policy in the case of the SDC communications kernel.

d. The SDC communications kernel is a capability-based operating system. That is, it keeps track of processes' allowed accesses to various objects by maintaining for each process an array of data structures called capabilities, each of which describes an object and an allowed access to that object.

e. The kernel is entered for one of two reasons:

   An interrupt is received from a device. This can only occur while a process is running. Or

   A kernel call (request for some kernel action) is made by some process.

   In either case, the kernel code is entered via a trap or interrupt while a process is running, runs straight through without interruption and then exits. The kernel exits by causing the resumption of the execution of the code of some process (which may or may not be the process which was running when the kernel was entered).

On the other hand, the SDC communications kernel has been modified so as to be appropriate for a communications environment rather than for a general user-support environment. For this and other reasons, the SDC communications kernel differs from the UCLA kernel in a number of important ways:

a. The SDC communications kernel does not provide for the dynamic creation or destruction of processes.

All processes exist from the time that the CPU is booted until it is halted.

b. The SDC communications kernel does not provide for swapping of processes in and out of memory. All processes are permanently resident in memory.

c. The UCLA system runs on a CPU (11/70, 11/45, etc) with three hardware modes: kernel, supervisor and user. The kernel runs in kernel mode, while the supervisor mode contains code called the "unix emulator" which provides an environment very like that of standard Unix to "application" code running in user mode. In distinction, "application" code written for the SDC system runs in supervisor mode and makes kernel calls directly. (User mode is unused.) SDC software thus can run in CPUs with only two hardware modes (11/34 and 11/23). (This is perhaps more a difference in usage than in the kernels themselves. The SDC communications kernel on an 11/70 or 11/45 could support some sort of emulator in supervisor mode, which could in turn provide some sort of standard environment to code in user mode.)

d. The SDC communications kernel incorporates very extensive provisions for interprocess communications.

e. In the UCLA system, a "Scheduler" process is responsible for choosing the next process to run. In the SDC kernel, processes are not swapped out, so scheduling is much simplified and has been made part of the kernel.

f. In the UCLA system, a "File Manager" process is responsible for giving capabilities to processes. In the SDC system, most capabilities of processes (for instance, the capability to access a certain peripheral) are assigned statically at the time the system is configured, by a program called the "Superlinker", running under normal Unix. The Superlinker assembles the CPU memory image and gives static capabilities to processes as instructed by the "superlinker control file", which is prepared by a human being. It is this human being who is ultimately responsible for deciding what processes are allowed to communicate, etc. (Some capabilites are given to and taken away from a process dynamically as part of the interprocess communication facilities; this is discussed in more detail below.) A separate File Manager process is not used.

The SDC communications kernel is written in a version of Pascal, augmented to provide certain extensions necessary for the use of Pascal in an operating system. The UCLA Pascal-C translator translates this into C, which is then compiled normally. The code is written in a top-down, highly modular and methodical method, which is intended to facilitate verification, although no verification or formal specification has been done as yet.

2.0 Security Policy ... An Example

The SDC kernel does not itself implement a security policy. In a typical communications system using the kernel, the total security policy would be a result of the properties of various parts of the system, of which the kernel is only one part. The kernel by itself does not guarantee that the security policy is correctly implemented. The kernel is only responsible for maintaining and separating process environments, and providing and regulating interprocess communications. Thus, the properties of the kernel are related to the total security policy as a lemma is to a theorem.

An example may help to make this clear.

Consider a CPU which is to act as a sort of terminal concentrator. The CPU is to support two terminals, one of which is to carry unclassified traffic only, and the other of which is to carry classified traffic only. The TCP and TELNET protocols are to be used to provide services to each of these terminals. In order to provide separation between the classified and unclassified traffic, the TCP and TELNET processes are duplicated. The internal situation in the CPU can be pictured as followed:

```
            -----------------       -----------------
            | Unclassified  |       |  Classified   |
            |   Terminal    |       |   Terminal    |
            -----------------       -----------------
                  ^   |                   ^   |
                  |   |                   |   |
-----------------------------------------------------------------
|                 |   V                   |   V                 |
|           -----------------------------------------           |
|           |                                       |           |
|           |              Line Driver              |           |
|           |         (Part of the kernel)          |           |
|           |                                       |           |
|           -----------------------------------------           |
|                 ^   |                   ^   |                 |
|                 |   V                   |   V                 |
|           -----------------       -----------------           |
|           | Unclassified  |       |  Classified   |           |
|           |    TELNET     |       |    TELNET     |           |
|           |  (A process)  |       |  (A process)  |           |
|           -----------------       -----------------           |
|                 ^   |                   ^   |                 |
|                 |   V                   |   V                 |
|           -----------------       -----------------           |
|           | Unclassified  |       |  Classified   |           |
|           |      TCP      |       |      TCP      |           |
|           |  (A process)  |       |  (A process)  |           |
|           -----------------       -----------------           |
|                 ^   |                   ^   |                 |
|                 |   |                   |   |                 |
|                 |   |                   |   |                 |
|                 |   | ----------------- |   |                 |
|                 |   ->|   Security    |-- |                 |
|                 |     |   MUX/DEMUX   |   |                 |
|                 ----- |  (A process)  |<-----                 |
|                       -----------------                       |
|                             ^   |                             |
|                             |   V                             |
|                       -----------------                       |
|                       |  Net Driver   |                       |
|                       | (Part of the) |                       |
|                       |   (kernel)    |                       |
|                       -----------------                       |
|                              | ^                              |
|                              | |                              |
-----------------------------------------------------------------
                               | |
                               V |
                              To Net
```

In this figure, the "drivers" are collections of subroutines; they are within the kernel, since they must manipulate the physical device registers.

The Security MUX/DEMUX process is a process whose responsibility it is to separate classified and unclassified traffic streams (on reception) and to merge the streams on transmission. We do not speculate here on what basis this is done. But it is clear that this process is performing a highly security-relevant function. Therefore, the code of this process must be appropriately verified. However, it is important to point out that the verification of the functioning of this process is quite distinct from the verification of the kernel. The process is not part of the kernel.

The TELNET and TCP processes are likewise processes, not part of the kernel. Because of the scheme diagrammed above, we can hope to be able to show that the malfunctioning of any of these processes would not be able to violate security constraints. (Note that this diagram represents only one example of a system which might be build on the kernel.)

Note that the drivers handle unseparated data; therefore they too would need to be verified. However, this is true even before we make the observation that they handle unseparated data: They must be verified because they are part of the kernel, and all of the kernel must be verified.

Now we are in a position to discuss the role of the kernel itself. What are the services and protections that the kernel provides?

First of all, the kernel provides and separates the environments of the processes. For example, the kernel sets the machine mapping registers when one process runs so that the code and data of that process are accessed, and so that the code and data of some other process are not accessed.

Second of all, the kernel provides interprocess communications facilities as specified when the system is configured. In the figure above, for example, the various arrows represent interprocess communications mechanisms called "queues". (These will be discussed in more detail below.)

When the system was configured, the responsible person specifies what processes are to exist, and what communications paths between them are to exist.

The tool by which this is done is the "superlinker" mentioned above. The responsible person prepares a "superlinker control file". For instance, for the system pictured above, the superlinker control file will specify that there are to be five processes. Each of these processes has previously been compiled, and its object code is ready and waiting. The control file specifies where this object code is to be found. Furthermore, the control file specifies exactly what queues are to exist in the system, what processes are allowed to place information on a given queue, and what processes are to be allowed to take information off of a given queue.

This superlinker control file is processed by the superlinker program, which is running under whatever development system is in use. (Not under the kernel.) The superlinker prepares the complete memory image of the CPU. In particular, it prepares the kernel tables which establish the existence of the various queues and what processes are allowed to enqueue to and dequeue from each one of them. (This will be discussed in more detail below.)

Now we can describe what it is that the kernel is trusted to do: the kernel is trusted to correctly implement and administer the system described by the superlinker control file. For example, if the superlinker control file describes the system shown in the figure above, then verification of the kernel will ensure that the unclassified TELNET process will not be able to dequeue information from the queue which is shown as leading from the classified TCP to the classified TELNET.

In order to correctly understand the nature of the security policy of the kernel as shown in the example above, it is very important to understand: The MUX/DEMUX process may be described as "trusted" in that it is trusted by the human beings who design, configure and use the system. However, it is inappropriate to describeo this process as "trusted" by the kernel. The kernel does not have a notion of "trusted" process. In particular, there is no "trusted" boolean in the per-process table maintained by the kernel. The kernel knows only what communications paths each process has been authorized to use.

In the example, the queue from the net driver to the MUX/DEMUX process carries both classified and unclassified information, while the queue from the unclassified TCP to the unclassified TELNET carries only unclassified information. Thus, from a security point of view, these queues are very different. However, there is nothing in the kernel corresponding to this difference in the nature of these queues.

We can describe the philosophy here as this: the "real" security policy is executed by the person who prepares the superlinker control file. The kernel is responsible only for seeing that that person's descisions are enforced. Note that this is appropriate for the purpose for which the SDC kernel has been designed. That is, since the system is static, there is no need to burden the kernel with code, algorithms, etc, for making security-related descisions. Instead, these descisions are made beforehand, and the kernel is only responsible for enforcing them.

Note that in the example, the Line Driver software in the kernel handles both terminals. There is no reason to provide two copies of this software; both copies would have to reside in the kernel, so as to access the hardware device registers, and would have be verified to function properly, as is true for any part of the kernel and the kernel as a whole. There would be no hardware separation between the two copies.

Note that as part of its functioning, the driver must take data from the queue from the unclassified TELNET process, and place it on the line to the unclassified terminal. Similarly for the classified terminal and for the other direction of flow. It must be verified that this function is performed correctly; but this is covered by the requirement that all the kernel functioning must be verified to perform correctly.

If the kernel were to be verified, what is it that would be verified? What would be the formal properties that would have to be verified to hold?

The kernel is responsible for

a. Maintaining and separating process environments.

b. Providing and regulating interprocess communications.

c. Operating devices.

Verification of the kernel would require formally stating the nature of these responsibilites. (These statements would probably include formal statements of the effects of the various kernel calls.) Then it would be necessary to formally prove that the kernel code does properly carry out these responsiblites.

As already emphasized, verification of the kernel would be only part of what would have to be done to verify that a given system satisfies some security policy. Various non-kernel parts of the system, as well as various aspects of the total system architecture, would also have to be verified.

Certain parts of the support software used to produce the system would also have to be appropriately verified. Clearly, an important part of this software is the superlinker. The output of the superlinker is source code versions of the kernel tables, which are then compiled, linked, and built into a total memory image. These kernel tables could be human-inspected, but this would be a very difficult task, which itself would use many machine aids. If there were any chance of having to do this tedious human inspection repeatedly, verification of the superlinker would be the proper thing to do instead.

The kernel was developed in a context which emphasized the production of working code in a relatively short time. For this reason, it was decided neither to formally specify the properties of the kernel, nor to attempt to formally demonstrate anything about it. Some such effort may be made in the future.

It is of course the case that code which was not developed from formal specifications may be quite difficult to formally verify after the fact, and will almost certainly have to be modified in order to be verified. This may be true because actual security flaws are found by the formal analysis, or because some aspects of the existing code are particularly unamenable to verification. However, there are some aspects of the existing kernel - the capability orientation in particular - which we hope will ease formal verification.

3.0 The Environment of a Process

To begin with, we emphasize that a "process" is not part of the kernel, but rather an "application" program for which the kernel provides environment and services. No part of the kernel is described as a process.

In a PDP 11/34, the virtual address space of a process comprises 64K bytes - each process produces 16-bit addresses as it accesses memory. These virtual addresses are translated to physical addresses by the memory management hardware. This hardware manages the process' virtual memory space in eight pieces, each of which contains 8K bytes. These pieces are the "pages" of a process' virtual address space.

These pages are used as follows:

a. One page accesses the "library". This is a collection of commonly useful subroutines. A typical routine would be a routine for converting between a machine clock, which might read in seconds past January 1, 1970, to human time (date and time). The library is read-only to all processes.

b. One or more pages are used to access the process' text ... that is, its executable code. This access is read-only.

c. One or more pages are used to access the process' data area ... that is, the area in which initialized variables are kept. This area is normally read-only, but may be made writeable, by special instructions to the superlinker.

d. One or more pages are used to access the process' so-called "bss" area ... that is, the area in which variables which are initially zero are kept. This area is read-write.

e. The last page (page seven) accesses the process' "communications block". This is an area of memory shared by the process and the kernel and used for communications between a process and the kernel.

f. The remaining pages (there are at most three) are free to be used to "map in" blocks of data passed from process to process using the interprocess communications mechanisms described below. These are referred to as mappable pages.

In an 11/70, the situation is similiar, except that an 11/70 has "separate I and D space", and so has twice as many pages for each process as the 11/34.

When an event occurs which affects a process, the kernel posts a notification of the event in the process' communications block, which the process looks at in the course of its main loop, which is described below. (Section

4 discusses traps and interruptions in more detail.)

In the SDC system, programmers write code which makes kernel calls directly. There is no "emulator" to provide the running process with an environment like that of some familiar operating system. (This is in distinction to the UCLA Data Secure UNIX system.) Since the programmer is writing code to run in an environment which is unfamiliar to him, we have taken the approach of providing a standard top-level structure for every process. (This also makes understanding a process written by another programmer considerably easier.)

This standard top-level structure is implemented by providing each programmer with the same "main" routine. (Again: we emphasize that this "main" is part of the process, not part of the kernel.) The entire code of a process consists of subprocedures called from this highest level procedure "main". (In particular, there are no "interrupt handler" or "completion" routines which are initiated directly by the kernel.) The outline of main is as follows:

```
procedure main;
begin
    initialize;
    while (true) do
    begin
        Set "summary" flag in communications block to false.
        while (some external event remains unprocessed) do
        begin
            Call procedure to process that external event.
        end;
        K_SLEEP;
    end;
end;
```

The procedure "main" is caused to begin executing when the system is booted. Main never exits.

The process begins by calling an initialization subroutine, and then enters an infinite loop. This loop basically does nothing except process external events. ("External" here means external to the process.) The process detects that there are external events to be processed by examining its communications block. When there are no external events to be processed, the process makes the system call K_SLEEP to give up the CPU until some external event occurs. When an external event does occur, the kernel awakens the process, which resumes execution just as though the K_SLEEP call had returned immediately.

From the ordinary programmer's point of view, writing a process to run under the SDC kernel consists in coding various procedures which are called from main, the procedures which they in turn call, etc.

The "summary" flag in the communications block is used in conjunction with the K_SLEEP call to avoid a possible race condition.

(If the summary flag were not used, the following might be possible:

> A process has processed all previously pending external events, has decided that there is nothing more to do, but has not yet made the K_SLEEP call. Now an external event occurs. The kernel posts a notification of the event in the process' communication block. However, the process has already decided to go to sleep. The process now makes the K_SLEEP call. As far as the kernel can see, the process has disposed of the new event. Thus the process goes to sleep without handling the event, and might even sleep forever.)

The summary flag avoids this race condition as follows: Any time that the kernel posts an external event to a process, it sets the summary flag in the process' communications block to "true". If the summary flag is true when the K_SLEEP call is made, then the process is not put to sleep; the K_SLEEP call returns immediately. It is easy to see that this mechanism, and its usage as in "main" above, avoids the race condition.

4.0 Interrupts and Traps

The kernel operates with all interrupts locked out (PDP-11 priority 7). Thus, if a device wishes to interrupt while the kernel is executing, the interrupt will remain pending until the kernel exits and a process starts to execute. Then that process will be immediately interrupted.

Suppose that an interrupt occurs while a process is executing. The CPU will be interrupted and the kernel will handle the interrupt. When the process resumes executing, it will resume at exactly the place at which it was when the interrupt took place. In this sense, the interrupt is transparent to the process.

If the interrupt implies that some process should be notified of a certain external event, then the kernel posts a notification in the communications block of that process. The process is awakened if it was previously asleep. If the notified process happens also to be the process that was running when the interrupt took place, then the process finds out about the event when it returns to its "main" routine and examines its communications block.

Thus a process runs without interrupts visible to that process. The only possible race conditions that might affect a process are concerned with the reception of notifications of external events. These problems are handled by the summary flag and the provision of a standard "main". Thus, a programmer can produce code for a process without considerations of race conditions, critical areas, etc. This is clearly of great benefit in a security-oriented system which is also production-oriented.

The only trap used in the system is the so-called "EMT" trap, which is used by a process to make a kernel call. The occurence of a trap while in kernel mode would indicate a bug in the kernel code. In this case the kernel halts the machine. A trap other than the EMT trap while a process is running indicates a bug in the code of the process. The kernel handles this by causing the process to be re-entered and restarted at a low virtual process address.

5.0 The Capability List

The kernel maintains for each process a "capability list". This is an array of records, called "capability slots". An index into this array is called a "capability index". A capability slot, if not empty, contains a "capability". A capability names some "object" and describes an allowed "access" to that object. Some examples:

a. A (statically defined) section of a disk is an object. Reading and writing are the two important accesses.

b. The central clock maintained by the kernel is an object. The only access which may be given by a capability to the clock is the ability to set the clock. (Any process is allowed to read the clock without having an explicit capability to do so.)

c. A block of memory is an object. Reading and writing are the two important accesses.

The capability list for each process is maintained by the kernel. Some capabilities are placed in the list by the superlinker at the time that the CPU memory image is prepared, while other capabilities are placed in or removed from the list in response to kernel calls. The process gets no access to its capability list, either read or write.

A capability serves not only to define what accesses a process has to a given object; it serves to actually identify that object. For example, suppose that one process communicates with another process via a "queue", as discussed further below. When enqueueing information to the other process, the process names the queue by giving the capability index to the capability which gives the process' access to its end of the queue.

As another example, suppose that a certain process is to be allowed to set the system clock. The superlinker control file will contain lines instructing the superlinker to set into the process' capability list a capability to set the clock. The superlinker control file, in the part describing the capabilities which the process is to have, will contain a line such as

```
clock capability on 12
```

This specifies that the process is to have a capability to set the system clock, located at index 12 in its capability list. When the process makes the K_SET_TIME system call, one of the parameters will be the number 12. In fact, the call is

```
K_SET_TIME(12, new_time)
```

When this call is made, the kernel will check slot number 12 of the process' capability list to see if it contain a capability to set the clock. Since it does, the kernel will do what the call asks it to do, namely to set the clock. Note that the kernel does not search the capability list of the process for a capability allowing the process to do what it has asked to do.

If the process by mistake made the call K_SET_TIME(13,new_time), the kernel will look in slot number 13 of the process' capability list. Since this slot does not contain a capability to set the clock, the call will fail. That is, the kernel will give the process a return indicating that the call failed because of a "bad capability" - that is, the capability at the indicated capability index was not what was required. Also, the clock will not be set.

Notice that although the process cannot either read or write its capability list, since that list is maintained entirely by the kernel, the process must know what is in each capability slot. Capabilities are placed in the capability list of a process either statically by the superlinker, like the capability to set the clock, or else as a result of kernel calls made by the process, as in the case of getting a data block as described below. Thus the process can keep track of the entries of its capability list without in fact being able to read it.

6.0 Interprocess Communications

This section describes the major method of interprocess communications under the SDC kernel, namely the enqueueing and dequeueing of blocks. (There are other methods of interprocess communications which are not described here.)

The kernel maintains a pool of free memory blocks. These are blocks of 128 bytes of memory (in our current implementations). The blocks are clear as kept in the free pool. When one process wishes to send a message to another process, the sequence of events is as follows:

a. The first process gets a block, and writes information in it.

b. The first process places the block on a queue to the second process.

c. The second process takes the block off the queue and reads the information from it.

d. The second process returns the block to the kernel, which clears it and puts it back in the free pool.

In more detail, the steps are as follows:

The first process makes a K_GET_DATA_BLOCK kernel call. An argument to this is a capability index. This must be the index to a currently empty capability slot. The kernel will remove a block from the free pool and place a read-write capability to the block in the specified slot.

The process then makes a K_MAP call. This specifies the capability index where the capability to the block is located, and one of the process' virtual pages, which must be unused. The kernel in response sets the memory management hardware to make the block appear at the beginning of that page of the process' virtual address space, giving the process read and write access to the block. This is called "mapping the block in".

The process can now read and write the block, using references to a data structure which is forced to reside at the appropriate location in the process' virtual address space.

Now, this sequence of operations is a natural pair: when a process gets a data block, it will almost certainly want to "map the block in" to access it. Thus, these two calls can be combined for greater efficency. This is in fact what has been done. That is, the K_GET_DATA block call has additional parameters which will allow the calling process to map the block in as part of the call.

The process then makes a K_ENQUEUE call. The parameters here are the capability index naming the block, and the capability index naming the enqueue end of the queue. (The queue is defined, and the capability to the queue is given, by the superlinker.) In response, the kernel removes the capability to the block from the first process' capability list, puts the block on the queue (which is maintained entirely by the kernel), and unmaps the block, so that the process no longer has access to it. It posts a notification to the second process that the queue has something on it, and wakes the second process if it is asleep.

The second process makes a K_DEQUEUE call. The parameters here are a capability index to the dequeue end of the queue, and a capability index to an unused slot in its capability list. The kernel removes the block from the queue, and puts a capability to that block in the specified slot. The normal sequence of events is that a receiving process will first dequeue a block and then map it in, similiar to the situation in the case of the K_GET_DATA_BLOCK call. Therefore the K_DEQUEUE call has optional parameters by which the calling process asks the kernel to map the dequeued block in to a specified virtual page.

The second process can now read the data in the block.

The second process finally makes a K_RELEASE_DATA_BLOCK kernel call, specifying the capability index at which the capability referring to the block is located. The kernel removes the capability, unmaps the block from the process' virtual memory space, clears the block and returns it to the free pool.

The above description is one of the simplest of the interprocess communications mechanisms provided by the SDC kernel. One of the more interesting variations is the ability of the kernel to regulate write-access by a process to the contents of a block on a basis of a finer granularity than the whole block itself.

This facility might be useful if there were a process that should be allowed to modify certain fields in a block, but not other fields. It might be the case that some process receives a block from another via a queue, and should be allowed to modify a "header" field within the block, but no other part of the block.

This can be achieved in the SDC kernel as follows: Special instructions are placed in the superlinker control file. These instructions include a specification (namely, a bit-mask) of which bytes of the blocks dequeued from a certain queue the process in question is to be able to alter. The superlinker then configures the kernel's tables in a special way. Now when the process dequeues a block from the queue in question, the process gets a read-only capability to the block. When the process uses the K_MAP call to "map the block in", the kernel sets the hardware mapping registers so that the process gets only read-access to the block. The process sets the fields it is permitted to set by making a K_WRITE_BLOCK call. The parameters to this call are the capability index to the block, along with (the address of) a buffer of 128 bytes in the process' data space. The kernel will then copy from that buffer to the block those bytes which are indicated by the bit-mask supplied to the superlinker by the superlinker control file.

This kind of fine-granularity control must be implemented by the kernel software, since the 11/70 memory management hardware does not have the necessary capabilities.

7.0 Time

The kernel maintains a 48-bit "fast" clock which is incremented every 10 microseconds, using the DEC KW11-P clock device. This can be read by a process, using the K_GET_TIME kernel call.

The kernel also maintains for each process a "slow" clock. This is a counter in the process' communications block which is incremented every half-second. By setting variables in its communications block, a process can arrange for the kernel to give it (the process) an "alarm" notification after a specified number of half-second ticks.

By using the slow clock and the associated alarm mechanism, a process can implement any sort of facilities for maintaining multiple named timers, as it chooses. Note that using the slow clock and the alarm mechanism do not require system calls. The associated system overhead is thus quite low.

The kernel allocates time among processes by time slicing at one-tenth second intervals.

8.0 Implementations and Results

The SDC kernel has so far been implemented on the PDP 11/70, 11/34, 11/23 and 11/03. (The 11/23 and 11/03 implementations are modifications: the 11/23 version allows interrupts, while the 11/03 version is more properly viewed as a kernel emulator.)

The code is written in a modified version of Pascal, as used in the UCLA kernel, with small amounts of assembly language.

The 11/70 version of the code comprises approximately 2500 Pascal statements, including drivers for the DH11, DL11, RX01, RP05, TE16, and other devices. This becomes approximately 30000 bytes of instructions. (Total kernel size, including all tables, is extremely dependent on the system being configured: the number of processes, the sizes of their capability lists, the number of queues, etc.)

The times required for some kernel calls is shown below.

	11/70	11/34	
K_GET_TIME (Read fast clock.)	0.81	1.8	milliseconds
K_GET_DATA_BLOCK (Get block and map it in.)	1.5	2.7	milliseconds
K_ENQUEUE (Put block on queue.)	1.7	3.1	milliseconds
K_DEQUEUE (Get block off queue and map it in.)	1.9	3.5	milliseconds
K_RELEASE_DATA_BLOCK (Clear block and return to pool.)	2.0	4.4	milliseconds

The only one of these calls which has an equivalent in the Unix system is the K_GET_TIME call. The Unix "time" system call takes .31 milliseconds on the 11/70. [3]

We can use these numbers to get estimates of the bandwidth of the enqueue/dequeue interprocess communication path under several assumptions.

First of all, consider the following situation:

```
---------         ---------         ---------
|       |         |       |         |       |
|   A   |------->|   B   |------->|   C   |
|       |         |       |         |       |
---------         ---------         ---------
```

Here, we are supposing that the blocks are prepared by A, processed by B, and consumed and released by C. The total kernel call overhead associated with B receiving and transmitting one 128-byte block is

----

[3] All times discussed below will be for the 11/70, unless specified otherwise.

time for K_DEQUEUE	1.9 ms
time for K_ENQUEUE	1.7 ms
total	3.6 ms

This corresponds to a throughput of about 35K bytes per second.

A second situation is the following:

```
 ---------         ---------
 |       |         |       |
 |   A   |-------->|   B   |
 |       |         |       |
 ---------         ---------
```

Here, we suppose that A gets a block, prepares a message, and enqueues the block to B. B dequeues the message, reads it, and then releases the block. The total kernal call time per 128-byte block here is

time for K_GET_DATA_BLOCK	1.5 ms
time for K_ENQUEUE	1.7 ms
time for K_DEQUEUE	1.9 ms
time for K_RELEASE_DATA_BLOCK	2.0 ms
total	7.1 ms

This corresponds to a throughput of about 18K bytes per second.

This calculation does not allow for the time necessary to switch processes so as to allow A and B to both run, and thus may seem unduly optimistic. However, it is actually quite realistic. When a system is heavily loaded, as is the case of interest for throughput calculations, process A would typically have a number of external events to process when it wakes up. A will then process all of these events, producing a number of blocks which it enqueues to B, before it (A) goes to sleep, letting B run. B then will process all these input blocks at one scheduling. Thus the time required to switch from A to B is divided among this number of blocks, and so does not greatly affect the throughput.

The time required to switch processes is, however, of some interest. The experiment

```
          ---------         ---------
          |       |         |       |
          |   A   |-------->|   B   |
          |       |         |       |
          ---------         ---------
```

was re-run in such a way as to force B to be scheduled every time A sent one block. The following figure shows the times used to send one block. Note that both processes used the standard "main"; the test did not separate out the time in main from the time actually in the kernel.

```
<-----  Process A ----->              <----- Process B ----->

          Enqueue             Kernel          Dequeue
             |                  |                |
   Get       |       Exit       |      Enter     |     Release
  block      |       main       |      main      |      block
    |        |        |         |        |       |        |
    V        V        V         V        V       V        V
|-------|-------|-------|-----------|-------|-------|-------|

 < 1.5 > < 1.7 > <---------  10  ----------> < 1.9 > < 2.0 >


 <------------------------  17  ------------------------->
```

This shows a worst-case time of 17 milliseconds for a 128-byte block. This corresponds to a throughput of 7.5K bytes per second.

It should be remembered that these throughputs are based on the use of 128-byte blocks, as in our current implementations. The use of larger blocks would be a minor change, and would result in proportionally larger bandwidths, since kernel call times are independent of the size of the block. [4] For example, if 256-byte blocks were used, the throughputs above would nearly double, giving values of 70K, 36K, and 15K bytes per second.

In considering these speeds and throughputs, it should also be pointed out that the SDC kernel, although it has been in use for some time, has not been extensively worked over to increase its speed. Effort in this area would undoubtedly pay off.

---

[4] with the exception of the K_RELEASE_DATA_BLOCK call.

In comparison, the throughput of a Unix pipe, on an otherwise-idle 11/70, is about 25K bytes per second. This is the rate when a sending process sends in units of 128 bytes. Increasing the send unit to 1280 bytes leaves the throughput rate approximately unchanged.

9.0 Denial of Service

The SDC kernel does not attempt to deal with denial-of-service threats. That is, a malicious process could cause CPU usefulness to be so degraded that the CPU could perform no useful work. For example, a process could (potentially but improbably) get and keep a large number of blocks. (This threat is somewhat limited: a process cannot get more blocks than it has slots in its capability list. This is a per-process parameter in the superlinker control file.)

Facilities could be added to the SDC kernel to address some denial-of-service issues, but it should be pointed out that it is consistent with the objects of the SDC kernel not to worry about denial-of-service issues. The reason is that there are no "optional" processes in a communications processor of the kind that the kernel was constructed to support. That is, the correct functioning of each process is necessary for the system to provide correct service. If any process is not performing its tasks correctly, service will be denied, and the kernel cannot do anything about it. However, security is preserved regardless of service denials.

10.0 Current Status

The SDC kernel was coded several years ago. It is currently operational for the Department of Defense on a number of CPUs functioning as special communications controllers and network front ends for ARPANET-like packet network terminal and host interfaces. Our experience so far shows that the resulting system provides throughput which is competitive with other systems not using a kernelized architecture.

11.0 References

Kampe, M., et al. The UCLA Data Secure Operating System. Tech. Rep., UCLA, July 1977.

Popek,G., and Farber, D. A Model for Verification of Data Security in Operating Systems. Comm. ACM, 21 9 (Sept 1978) 737-749. (Contains other pertinant references.)

Walton, E. The UCLA Kernel. Master's Th., Comptr. Sci. Dept., UCLA, 1975.