C Threads

Eric C. Cooper
Richard P. Draves

Department of Computer Science
Carnegie Mellon University
Pittsburgh, Pennsylvania 15213

Draft of 11 September 1990

Abstract

The C Threads package allows parallel programming in C under the MACH operating system. The package provides multiple threads of control for parallelism, shared variables, mutual exclusion for critical sections, and condition variables for synchronization of threads.

This research was sponsored by the Defense Advanced Research Projects Agency (DoD), ARPA order 4864, monitored by the Space and Naval Warfare Systems Command under contract N00039-84-C-0467.

The views and conclusions contained in this document are those of the authors and should not be interpreted as representing official policies, either expressed or implied, of the Defense Advanced Research Projects Agency or of the U.S. Government.

1. Introduction

MACH provides a set of low-level, language-independent primitives for manipulating threads of control [3]. The C Threads package is a run-time library that provides a C language interface to these facilities [1]. The constructs provided are similar to those found in Mesa [4] and Modula-2+ [5]: forking and joining of threads, protection of critical regions with mutex variables, and synchronization by means of condition variables.

2. Naming Conventions

An attempt has been made to use a consistent style of naming for the abstractions implemented by the C Threads package. All types, macros, and functions implementing a given abstract data type are prefixed with the type name and an underscore. The name of the type itself is suffixed with `_t` and is defined via a C `typedef`. Documentation of the form

```
typedef struct mutex {...} *mutex_t;
```

indicates that the `mutex_t` type is defined as a pointer to a *referent type* `struct mutex` which may itself be useful to the programmer. (In cases where the referent type should be considered *opaque*, documentation such as

```
typedef ... cthread_t;
```

is used instead.)

Continuing the example of the `mutex_t` type, the functions `mutex_alloc()` and `mutex_free()` provide dynamic storage allocation and deallocation. The functions `mutex_init()` and `mutex_clear()` provide initialization and finalization of the referent type. These functions are useful if the programmer wishes to include the referent type itself (rather than a pointer) in a larger structure, for more efficient storage allocation. They should not be called on objects that are dynamically allocated via `mutex_alloc()`. Type-specific functions such as `mutex_lock()` and `mutex_unlock()` are also defined, of course.

3. Initializing the C Threads Package

3.1. cthreads.h

```
#include <cthreads.h>
```

The header file `cthreads.h` defines the C threads interface. All programs using C threads must include this file.

3.2. cthread_init

```
void
cthread_init()
```

The `cthread_init()` procedure initializes the C threads implementation. A program using C threads must explicitly call `cthread_init()` (typically from `main()`) before using any of the other functions described below. Multiple calls to `cthread_init()` are harmless.

4. Threads of Control

4.1. Creation

When a C program starts, it contains a single thread of control, the one executing `main()`. The thread of control is an active entity, moving from statement to statement, calling and returning from procedures. New threads are created by *fork* operations.

Forking a new thread of control is similar to calling a procedure, except that the caller does not wait for the procedure to return. Instead, the caller continues to execute in parallel with the execution of the procedure in the newly forked thread. At some later time, the caller may rendez-vous with the thread and retrieve its result (if any) by means of a *join* operation, or the caller may *detach* the newly created thread to assert that no thread will ever be interested in joining it.

4.2. Termination

A thread `t` terminates when it returns from the top-level procedure it was executing.[1] If `t` has not been detached, it remains in limbo until another thread either joins it or detaches it; if `t` has been detached, no rendez-vous is necessary.

4.3. cthread_fork

```
typedef ... any_t;
typedef ... cthread_t;
```

The `any_t` type represents a pointer to any C type. The `cthread_t` type is an integer-size ‘‘handle’’ that uniquely identifies a thread of control. Values of type `cthread_t` will be referred to as thread identifiers.

```
cthread_t
cthread_fork(func, arg)
        any_t (*func)();
        any_t arg;
```

The `cthread_fork()` procedure creates a new thread of control in which `func(arg)` is executed concurrently with the caller's thread. This is the sole means of creating new threads. Arguments larger than a pointer must be passed by reference. Similarly, multiple arguments must be simulated by passing a pointer to a structure containing several components. The call to `cthread_fork()` returns a thread identifier that can be passed to `cthread_join()` or `cthread_detach()` (see below). Every thread *must* be either joined or detached exactly once.

---

[1]Currently, this is not true of the initial thread executing `main()`. Instead, an implicit call to `exit()` occurs when `main()` returns, terminating the entire program. If the programmer wishes detached threads to continue executing, the final statement of `main()` should be a call to `cthread_exit()`.

4.4. cthread_exit

```
void
cthread_exit(result)
        any_t result;
```

This procedure causes termination of the calling thread. An implicit `cthread_exit()` occurs when the top-level function of a thread returns, but it may also be called explicitly. The result will be passed to the thread that joins the caller, or discarded if the caller is detached.

4.5. cthread_join

```
any_t
cthread_join(t)
        cthread_t t;
```

This function suspends the caller until the specified thread `t` terminates via `cthread_exit()`. (It follows that attempting to join one's own thread will result in deadlock.) The caller receives either the result of `t`'s top-level function or the argument with which `t` explicitly called `cthread_exit()`.

4.6. cthread_detach

```
void
cthread_detach(t)
        cthread_t t;
```

The detach operation is used to indicate that the given thread will never be joined. This is usually known at the time the thread is forked, so the most efficient usage is the following:

```
        cthread_detach(cthread_fork(procedure, argument));
```

A thread may, however, be detached at any time after it is forked, as long as no other attempt is made to join it or detach it.

4.7. cthread_yield

```
void
cthread_yield()
```

This procedure is a hint to the scheduler, suggesting that this would be a convenient point to schedule another thread to run on the current processor. Calls to `cthread_yield()` are unnecessary in an implementation with preemptive scheduling, but may be required to avoid starvation in a coroutine-based implementation.

4.8. cthread_self

```
cthread_t
cthread_self()
```

This function returns the caller's own thread identifier, which is the same value that was returned by `cthread_fork()` to the creator of the thread. The thread identifier uniquely identifies the thread, and hence may be used as a key in data structures that associate user data with individual threads. Since thread identifiers may be reused by the underlying implementation, the programmer should be careful to clean up such associations when threads exit.

4.9. cthread_set_data, cthread_data

```
void
cthread_set_data(t, data)
        cthread_t t;
        any_t data;

any_t
cthread_data(t)
        cthread_t t;
```

These functions allow the user to associate arbitrary data with a thread, providing a simple form thread-specific ''global'' variable. More elaborate mechanisms, such as per-thread property lists or hash tables, can then be built with these primitives.

5. Synchronization

```
typedef struct mutex {...} *mutex_t;

typedef struct condition {...} *condition_t;
```

This section describes mutual exclusion and synchronization primitives, called mutexes and condition variables. In general, these primitives are used to constrain the possible interleavings of threads' execution streams. They separate the two most common uses of Dijkstra's `P()` and `V()` operations into distinct facilities. This approach basically implements monitors [2, 4], but without the syntactic sugar.

Mutually exclusive access to mutable data is necessary to prevent corruption of data. As simple example, consider concurrent attempts to update a simple counter. If two threads fetch the current value into a (thread-local) register, increment, and write the value back in some order, the counter will only be incremented once, losing one thread's operation. A mutex solves this problem by making the fetch-increment-deposit action atomic. Before fetching a counter, a thread locks the associated mutex. After depositing a new value, the thread unlocks the mutex. If any other thread tries to use the counter in the meantime, it will block when it tries to lock the mutex. If more than one thread tries to lock the mutex at the same time, the C threads package guarantees that only one will succeed; the rest will block.

Condition variables are used when one thread wants to wait until another thread has finished doing something. Every condition variable should be protected by a mutex. Conceptually, the condition is a boolean function of the shared data that the mutex protects. Commonly, a thread locks the mutex and inspects the shared data. If it doesn't like what it finds, it waits using a condition variable. This operation also temporarily unlocks the mutex, to give other threads a chance to get in and modify the shared data. Eventually, one of them should signal the condition (which wakes up the blocked thread) before it unlocks the mutex. At that point, the original thread will regain its lock and can look at the shared data to see if things have improved. It can't assume that it will like what it sees, because some other thread may have slipped in and mucked with the data after the the condition was signaled.

One must take special care with data structures that are dynamically allocated and deallocated. In particular, if the mutex that is controlling access to a dynamically allocated record is part of the record, one must be sure that no thread is waiting for the mutex before freeing the record.

Attempting to lock a mutex that one already holds is another common error. The offending thread will block waiting for itself. This can happen when a thread is traversing a complicated data structure, locking as it goes, and reaches the same data by different paths. Another instance of this is when a thread is locking elements in an array, say to swap them, and it doesn't check for the special case that the elements are the same.

In general, one must be very careful to avoid deadlock. Deadlock is defined as the condition in which one or more threads are permanently blocked waiting for each other. The above scenarios are a special case of deadlock. The easiest way to avoid deadlock with mutexes is to impose a total ordering on the mutexes, and then ensure that threads only lock mutexes in increasing order.

One important issue the programmer must decide is what kind of granularity to use in protecting shared data with mutexes. The two extremes are to have one mutex protecting all shared memory, and to have one mutex for every byte of shared memory. Finer granularity normally increases the possible parallelism, because less data is locked at any one time. However, it also increases the overhead lost to locking and unlocking mutexes and increases the possibility of deadlock.

5.1. mutex_lock

```
void
mutex_lock(m)
        mutex_t m;
```

The `mutex_lock()` procedure attempts to lock the mutex `m` and blocks until it succeeds. If several threads attempt to lock the same mutex concurrently, one will succeed, and the others will block until `m` is unlocked. The case of a thread attempting to lock a mutex it has already locked is *not* treated specially; deadlock will result.

5.2. mutex_try_lock

```
int
mutex_try_lock(m)
        mutex_t m;
```

The `mutex_try_lock()` function attempts to lock the mutex `m`, like `mutex_lock()`, and returns `TRUE` if it succeeds. If `m` is already locked, however, `mutex_try_lock()` immediately returns `FALSE` rather than blocking. For example, a busy-waiting version of the `mutex_lock()` procedure could be written in terms of `mutex_try_lock()` as follows:

```
void
mutex_lock(m)
        mutex_t m;
{
        for (;;)
                if (mutex_try_lock(m))
                        return;
}
```

5.3. mutex_unlock

```
void
mutex_unlock(m)
        mutex_t m;
```

The `mutex_unlock()` procedure unlocks the mutex `m`, giving other threads a chance to lock it.

5.4. condition_signal

```
void
condition_signal(c)
        condition_t c;
```

The `condition_signal()` procedure should be called when one thread wishes to indicate that the condition represented by the condition variable is now true. If any other threads are waiting (via `condition_wait()`), then at least one of them will be awakened. If no threads are waiting, then nothing happens.

5.5. condition_broadcast

```
void
condition_broadcast(c)
        condition_t c;
```

The `condition_broadcast()` procedure is similar to `condition_signal()`, except that it awakens *all* threads waiting for the condition, not just one of them.

5.6. condition_wait

```
void
condition_wait(c, m)
        condition_t c;
        mutex_t m;
```

The `condition_wait()` procedure unlocks `m`, suspends the calling thread until the specified condition is *likely* to be true, and locks `m` again when the thread resumes. Since there is no guarantee that the condition will be true when the thread resumes, use of this procedure should always be of the form

```
mutex_lock(m);
...
while (/* condition is not true */)
        condition_wait(c, m);
...
mutex_unlock(m);
```

Shared variables should be inspected on each iteration to determine whether the condition is true.

6. Management of Synchronization Variables

A mutex or condition variable can be allocated dynamically from the heap, or the programmer can take an object of the referent type, initialize it appropriately, and then use its address.

6.1. Allocation

```
mutex_t
mutex_alloc()

condition_t
condition_alloc()
```

These functions provide dynamic allocation of mutex and condition objects.

6.2. Deallocation

```
void
mutex_free(m)
        mutex_t m;

void
condition_free(c)
        condition_t c;
```

These functions allow the programmer to deallocate mutex and condition objects that were allocated dynamically. Before deallocating such an object, the programmer must guarantee that no other thread will reference it. In particular, a thread blocked in `mutex_lock()` or `condition_wait()` should be viewed as referencing the object continually, so freeing the object ‘‘out from under’’ such a thread is erroneous, and can result in bugs that are extremely difficult to track down.

6.3. Initialization

```
void
mutex_init(m)
        struct mutex *m;

void
condition_init(c)
        struct condition *c;
```

These functions allow the programmer to initialize an object of the `struct mutex` or `struct condition` referent type, so that its address can be used wherever an object of type `mutex_t` or `condition_t` is expected. For example, the `mutex_alloc()` function could be written in terms of `mutex_init()` as follows:

```
mutex_t
mutex_alloc()
{
        register mutex_t m;

        m = (mutex_t) malloc(sizeof(struct mutex));
        mutex_init(m);
        return m;
}
```

Initialization of the referent type is most often used when the programmer has included the referent type itself (rather than a pointer) in a larger structure, for more efficient storage allocation. For instance, a data structure might contain a component of type `struct mutex` to allow each instance of that structure to be locked independently. During initialization of the instance, the programmer would call `mutex_init()` on the `struct mutex` component. The alternative of using a `mutex_t` component and initializing it using mutex_alloc() would be less efficient.

6.4. Finalization

```
void
mutex_clear(m)
        struct mutex *m;

void
condition_clear(c)
        struct condition *c;
```

These functions allow the programmer to finalize an object of the struct mutex or struct condition referent type. For example, the mutex_free() procedure could be written in terms of mutex_clear() as follows:

```
void
mutex_free(m)
        mutex_t m;
{
        mutex_clear(m);
        free((char *) m);
}
```

7. Shared Variables

All global and static variables are shared among all threads: if one thread modifies such a variable, all other threads will observe the new value. In addition, a variable reachable from a pointer is shared among all threads that can dereference that pointer. This includes objects pointed to by shared variables of pointer type, as well as arguments passed by reference in cthread_fork().

When pointers are shared, some care is required to avoid dangling reference problems. The programmer must ensure that the lifetime of the object pointed to is long enough to allow the other threads to dereference the pointer. Since there is no bound on the relative execution speed of threads, the simplest solution is to share pointers to global or heap-allocated objects only. If a pointer to a local variable is shared, the procedure in which that variable is defined must remain active until it can be guaranteed that the pointer will no longer be dereferenced by other threads. The synchronization functions can be used to ensure this.

The programmer must remember that unless a library, like the standard C library, has been designed to work in the presence of reentrancy, the operations provided by the library must be presumed to make unprotected use of shared data. Hence, the programmer must protect against this through the use of a mutex that is locked before every library call (or sequence of library calls) and unlocked afterwards.

7.1. Dynamic Allocation

Dynamic allocation and freeing of user-defined data structures is typically accomplished with the standard C functions malloc() and free(). The C threads package provides versions of these functions that work correctly in the presence of multiple threads.

8. Using the C Threads Package

All of the functions described have been implemented for the MACH multiprocessor operating system. Three implementations of threads are provided, in the form of libraries. Programs need not be recompiled to use a different thread implementation, only relinked. To compile a program that uses C threads, the user must include the file `cthreads.h`. (The directory `/usr/mach/include` should be in the user's `CPATH` search list for the C preprocessor to find this header file.) The program must call `cthread_init()` before using any other C threads features. To link a program that uses C threads, the user must specify on the `cc` command line one of the three libraries described below, followed by the `-lmach` library. (The directory `/usr/mach/lib` should be in the user's `LPATH` search list for the linker to find these libraries.)

8.1. The Coroutine Implementation

The first implementation, `-lco_threads`, uses coroutines within a single MACH task (UNIX process). Scheduling of these threads is non-preemptive, hence `cthread_yield()` should be called within loops that do not otherwise call synchronization procedures. The programmer will typically link with this version while debugging.

This implementation includes versions of the MACH interprocess communication primitives `msg_receive()`, `msg_send()`, and `msg_rpc()`, and a version of the UNIX `select()` system call, that can be called from one thread without blocking the other threads in the program. The other forms of UNIX I/O have not been redefined for use with `-lco_threads`, however. For example, calling `getchar()` from one thread may block all threads in the program, not just the caller. To work around this, the programmer should first call `select()` on the relevant file descriptor to guarantee that the subsequent input operation will not block.

8.2. The MACH Thread Implementation

The second implementation, `-lthreads`, uses one MACH thread per C thread. These threads are preemptively scheduled, and may execute in parallel on a multiprocessor. This is the implementation that should be used in the production version of a C threads program.

The current `-lco_threads` and `-lthreads` implementations allocate large fixed-size stacks for each C thread in virtual memory. The implementations rely on the MACH virtual memory system to allocate physical memory only as needed by the thread.

8.3. The MACH Task Implementation

The third implementation, `-ltask_threads`, uses one MACH task (UNIX process) per thread, and uses the MACH virtual memory primitives to share memory between threads. In most circumstances, the `-lthreads` implementation should be used instead of this one. An exception is when the programmer wishes to use the MACH virtual memory primitives to provide a specialized pattern of memory sharing between C threads.

Users of the `-ltask_threads` implementation should note that capabilities such as MACH ports and UNIX file descriptors are private to the task that creates them, and so cannot be shared. The current `-ltask_threads` implementation also makes stack segments private to each task, so automatic (stack-allocated) variables cannot be shared.

The MACH operating system currently limits the number of tasks (and hence the number of C threads in the `-ltask_threads` implementation) that a user may create. Applications that create large numbers of threads will encounter run-time errors when they exceed this limit. It may be the case that concurrent execution is required to avoid deadlock (for example, in a multi-stage pipeline). For applications with largely independent threads, however, a limited degree of parallelism may still allow correct execution. The following function can be used in such applications.

```
void
cthread_set_limit(n)
        int n;
```

This function limits the number of active threads to `n`. If a newly created thread of control exceeds this limit, it will not begin execution until another thread terminates.

9. Debugging

It is strongly recommended that the coroutine-based implementation (`-lco_threads`) be used for debugging, for the following reasons:

- The order of thread context switching is repeatable in successive executions of the program, so obvious synchronization bugs may be found easily.
- Since the program is a single MACH task, existing debuggers can be used.
- The user need not worry about concurrent calls to C library routines.

9.1. Low-Level Tracing

```
int cthread_debug;
```

Setting this variable to 1 causes diagnostic information to be printed when each C threads primitive is executed. Trace output appears on `stdout`.

9.2. Associating Names with C Thread Objects

```
void
cthread_set_name(t, name)
        cthread_t t;
        string_t name;

string_t
cthread_name(t)
        cthread_t t;

void
mutex_set_name(m, name)
        mutex_t m;
        string_t name;

string_t
mutex_name(m)
        mutex_t m;

void
condition_set_name(c, name)
        condition_t c;
        string_t name;

string_t
condition_name(c)
        condition_t c;
```

These functions allow the user to associate a name with a thread, mutex, or condition. The name is used when trace information is displayed (see above). The name may also be used by the programmer for application-specific diagnostics.

9.3. Pitfalls of Preemptively Scheduled Threads

The C run-time library needs a substantial amount of modification in order to be used with preemptively scheduled threads (`-lthreads` and `-ltask_threads`). Currently the user must ensure that calls to the standard I/O library are serialized, through the use of one or more mutex variables. (The storage allocation functions `malloc()` and `free()` do not require any special precautions.)

The debuggers currently available under MACH cannot be used on programs linked with `-lthreads` or `-ltask_threads`. Furthermore, the very act of turning on tracing or adding print statements may perturb programs that incorrectly depend on thread execution speed. One technique that is useful in such cases is to vary the granularity of locking and synchronization used in the program, making sure that the program works with coarse-grained synchronization before refining it further.

10. Examples

The following example illustrates how the facilities described here can be used to program Hoare monitors [2]. The program would be compiled and linked by the command

```
cc hoaremonitor.c -lthreads -lmach
```

```
/*
 * Producer/consumer with bounded buffer.
 *
 * The producer reads characters from stdin
 * and puts them into the buffer.  The consumer
 * gets characters from the buffer and writes them
 * to stdout.  The two threads execute concurrently
 * except when synchronized by the buffer.
 */
#include <stdio.h>
#include <cthreads.h>
typedef struct buffer {
    struct mutex lock;
    char *chars;      /* chars[0..size-1] */
    int size;
    int px, cx;       /* producer and consumer indices */
    int count;        /* number of unconsumed chars in buffer */
    struct condition non_empty, non_full;
} *buffer_t;
void
buffer_put(ch, b)
    char ch;
    buffer_t b;
{
    mutex_lock(&b->lock);
    while (b->count == b->size)
        condition_wait(&b->non_full, &b->lock);
    ASSERT(0 <= b->count && b->count < b->size);
    b->chars[b->px] = ch;
    b->px = (b->px + 1) % b->size;
    b->count += 1;
    condition_signal(&b->non_empty);
    mutex_unlock(&b->lock);
}
char
buffer_get(b)
    buffer_t b;
{
    char ch;
    mutex_lock(&b->lock);
    while (b->count == 0)
        condition_wait(&b->non_empty, &b->lock);
    ASSERT(0 < b->count && b->count <= b->size);
    ch = b->chars[b->cx];
    b->cx = (b->cx + 1) % b->size;
    b->count -= 1;
    condition_signal(&b->non_full);
    mutex_unlock(&b->lock);
    return ch;
}
void
producer(b)
    buffer_t b;
{
    int ch;
```

```
    do buffer_put((ch = getchar()), b);
    while (ch != EOF);
}
void
consumer(b)
    buffer_t b;
{
    int ch;
    while ((ch = buffer_get(b)) != EOF)
        printf("%c", ch);
}
buffer_t
buffer_alloc(size)
    int size;
{
    buffer_t b;
    extern char *malloc();
    b = (buffer_t) malloc(sizeof(struct buffer));
    mutex_init(&b->lock);
    b->size = size;
    b->chars = malloc((unsigned) size);
    b->px = b->cx = b->count = 0;
    condition_init(&b->non_empty);
    condition_init(&b->non_full);
    return b;
}
#define BUFFER_SIZE 10
main()
{
    buffer_t b;
    cthread_init();
    b = buffer_alloc(BUFFER_SIZE);
    cthread_detach(cthread_fork(producer, b));
    cthread_detach(cthread_fork(consumer, b));
    cthread_exit(0);
}
```

The following example shows how to structure a program in which a single master thread spawns a number of concurrent slaves and then waits until they all finish. The program would be compiled and linked by the command

```
        cc masterslave.c -lthreads -lmach
```

```
/*
 * Master/slave program structure.
 */
#include <stdio.h>
#include <cthreads.h>
int count;          /* number of slaves active */
mutex_t lock;       /* mutual exclusion for count */
condition_t done;   /* signalled each time a slave finishes */
extern long random();
init()
{
    cthread_init();
    count = 0;
    lock = mutex_alloc();
    done = condition_alloc();
    srandom(time((int *) 0));  /* initialize random number generator *
}
/*
 * Each slave just counts up to its argument,
 * yielding the processor on each iteration.
 * When it is finished, it decrements the global count
 * and signals that it is done.
 */
slave(n)
    int n;
{
    int i;
    for (i = 0; i < n; i += 1)
        cthread_yield();
    mutex_lock(lock);
    count -= 1;
    printf("Slave finished %d cycles.\n", n);
    condition_signal(done);
    mutex_unlock(lock);
}
/*
 * The master spawns a given number of slaves
 * and then waits for them all to finish.
 */
master(nslaves)
    int nslaves;
{
    int i;
```

```
    for (i = 1; i <= nslaves; i += 1) {
        mutex_lock(lock);
        /*
         * Fork a slave and detach it,
         * since the master never joins it individually.
         */
        count += 1;
        cthread_detach(cthread_fork(slave, random() % 1000));
        mutex_unlock(lock);
    }
    mutex_lock(lock);
    while (count != 0)
        condition_wait(done, lock);
    mutex_unlock(lock);
    printf("All %d slaves have finished.\n", nslaves);
    cthread_exit(0);
}
main()
{
    init();
    master((int) random() % 16);  /* create up to 15 slaves */
}
```

References

[1] Brian W. Kernighan and Dennis M. Ritchie.
The C Programming Language.
Prentice-Hall, 1978.

[2] C. A. R. Hoare.
Monitors: An Operating System Structuring Concept.
Communications of the ACM 17(10):549-557, October, 1974.

[3] Robert V. Baron, Richard F. Rashid, Ellen Siegel, Avadis Tevanian, and Michael W. Young.
MACH-1: A Multiprocessor Oriented Operating System and Environment.
In Arthur Wouk (editor), *New Computing Environments: Parallel, Vector, and Symbolic*. SIAM, 1986.

[4] Butler W. Lampson and David D. Redell.
Experience with Processes and Monitors in Mesa.
Communications of the ACM 23(2):105-117, February, 1980.

[5] Paul Rovner, Roy Levin, and John Wick.
On Extending Modula-2 for Building Large, Integrated Systems.
Technical Report 3, DEC Systems Research Center, January, 1985.

Table of Contents