

**C++ on a Parallel Machine**

*Thomas W. Doeppner Jr.<sup>1</sup>*

*Alan J. Gebele<sup>2</sup>*

*Department of Computer Science*

*Brown University*

*Providence, RI 02912*

November 17, 1987

Technical Report CS-87-26

---

<sup>1</sup>Doeppner's work was supported in part by a grant from the Encore Computer Corporation, by the Office of Naval Research and Defense Advanced Research Projects Agency under contract N00014-83-K-0146 and ARPA Order No. 4786, and by National Science Foundation Grant MCS8121806.

<sup>2</sup>Gebele is currently affiliated with Bellcore and was supported by Bellcore for this work.

## 1. Introduction

Exploiting a computer with multiple processors can often be difficult because suitable programming-language support for writing parallel programs does not exist. We have integrated C++ with a system called Threads (described below), enabling C++ to take full advantage of parallelism on a parallel computer such as the Encore Multimax<sup>3</sup>. This was accomplished by adding a set of *classes*, the *tasking package*, to the C++ library which the programmer can use to create and synchronize concurrent activity. These classes are based on the system described in [Stroustrup 2], which was designed for a single processor and used a coroutine linkage between concurrent activities (or *tasks*). While extending most of the functionality of Stroustrup's system to a true parallel machine, we have added a class *monitor* to take full advantage of the facilities provided by Threads.

## 2. Overview

The tasking package is based on four C++ classes to be used by the programmer as base classes to build their own classes for parallel computations (see [Stroustrup 1]). The first of these parallel classes is the class *task*. The constructors of any class derived from *task* run as a separate thread of control in the program. The second class, *monitor*, defines a set of routines to make a derived class act as a monitor [Hoare] for the controlled sharing of data between different tasks. A third concept, based on the two classes *qhead* and *qtail*, defines a one-way queue for communication between tasks. *TASKenvironment*, which is not defined directly by the user, is used to set and maintain task-system environment defaults for creating tasks and preemptive scheduling.

### 2.1. The Task Class

The base type *task* creates a separate thread of control for execution of part of a program. Normally, in order to define a task, one defines a class that is derived from *task* which contains only the constructor. It may look like:

```
struct scout : task {
    scout::scout(char*);
};

scout::scout(char *s)
{
    int i = 0 ;
    while (*s) if (*s++ == ' ') i++ ;
    resultis(i);
}
```

This task counts the number of spaces in an input string and is executed in parallel with anything else being done by the program. The routine *resultis()* is used to set a return value for the task so that some other task may get the value. *resultis()* also tells the tasking package to end the execution of the task.

To activate the task, we simply declare an instance of the class *scout* to create a new thread of control:

```
main(int argc, char *argv[], char *envp[])
{
    scout s("a line with four spaces");

    // do some other work.....

    printf("result is %d\n",s.result());
}
```

---

<sup>3</sup>Encore and Multimax are trademarks of the Encore Computer Corporation.

The routine *result()* returns the value from the call to *resultis()* but only after the task has completed. Tasks can be in three states, RUNNING, IDLE, or TERMINATED. Normally, a task is in RUNNING state until it exits either by returning through the normal execution or by a call to *resultis()*, when it becomes TERMINATED. A task is IDLE if it stops execution for some reason; for instance, if it calls *result()* when the task from which it is asking for a value has not yet completed. If a task returns without calling *resultis()*, the value of *result()* is -1.

The *task* class gives other services for synchronization of execution in addition to waiting upon the termination of a task. A task may wait upon another task by calling the *wait()* operation. The waiting task goes to an IDLE state until the other task either calls a *wakeup()* operation, or completes execution and goes to a TERMINATED state. The *wait()* and *wakeup()* operations are based on wait queues associated with a task. Each task has one wait queue by default, whose number is 0; at creation time, a task may allocate more queues by passing a parameter to the task constructor. If no argument is given, calls to *wakeup()* and *wait()* act on queue 0. To specify another queue, give its number as the argument.

In the following example, the *wait()* operation is used to ensure that all the threads doing computations are completed before any results are printed. This example multiplies two matrices by computing each member with a separate task. Note that here we do not use the *wakeup()* operation, only *wait()*. Hence the synchronization waits for the task to be TERMINATED. (This use of tasking for matrix multiplication is strictly an illustrative example; even though our implementation of tasking is very inexpensive, it is not cheap enough to make the following example technique practical. However, we are currently developing an even cheaper form of tasking, discussed below, which would be practical for this example.)

```

class matrix {
    int *vec ;
    int len ;
public:
    matrix (int, int);
    ~matrix()                { delete vec ; }
    int get(int r, int c)    { return vec[len*r+c] ; }
    void put(int r, int c, int val) { vec[len*r+c] = val ; }
    int length()            { return len ; }
};

matrix::matrix(int l, int initval = 0)
{
    len = l ;
    vec = new int[l*1] ;
    for (int i = 0 ; i < l*1 ; vec[i++] = initval) ;
}

#define MSIZE  3

matrix A(MSIZE);
matrix B(MSIZE,8);
matrix C(MSIZE);

struct mult : task {
    mult(int, int);
};

mult::mult(int r, int c) : ("mult")
{
    int t = 0 ;

```

```

    for (int i = 0 ; i < A.length() ; i++)
        t += A.get(r,i) * B.get(i,c);

    C.put(r,c,t);
}

main(int argc, char *argv[], char *envp[])
{
    // declare array of mult pointers
    mult *mvec[MSIZE*MSIZE] ;

    // initialize A
    for (int r = 0 ; r < MSIZE ; r++)
        for (int c = 0 ; c < MSIZE ; c++)
            A.put(r,c,(r+1)*(c+1));

    // startup mult tasks
    for (r = 0 ; r < MSIZE ; r++)
        for (c = 0 ; c < MSIZE ; c++)
            mvec[r*c] = new mult(r,c);

    // wait on all threads to complete
    for (r = 0 ; r < MSIZE ; r++)
        for (c = 0 ; c < MSIZE ; c++)
            mvec[r*c]->wait();

    printf("the result is :\n");
    for (r = 0 ; r < MSIZE ; r++)
        for (c = 0 ; c < MSIZE ; c++)
            printf("c[%d,%d] = %d\n", r, c, C.get(r,c));
}

```

Several other useful operations exist within the *task* class. First, the *wakeupall()* operation wakes up all tasks waiting on a queue (not just the first task, as *wait()* does). There are also several operations for checking the state of a task which are often useful in debugging: *print()* prints to stdout a summary of the state of the task, *name()* returns a pointer to the string containing the name given the task at creation time, *status()* returns the running status of the task, and *priority()* returns the current priority of execution the task. The package also provides a method of changing the task's priority by calling *setpriority()*.

All computation in the program, even in the *main()* routine, is done in the context of some task. At the beginning of execution, the program initiates the Threads system and then starts up the first task, the *main()* routine. Actually, at compile time the name of *main()* is changed to *TASKmain::TASKmain()*, which is really a class constructor derived from *task*. Hence the declaration line of *main()* can also be written as:

```
TASKmain::TASKmain(int argc, char *argv[], char *envp) :("TASKmain")
```

This means that the *TASKmain()* can be scheduled and waited upon and can give results just like any other task. (Because the tasking package redefines *main()*, all three arguments to *main()* must be included, even though the program does not necessarily use any of them.)

## 2.2. The Monitor Class

The class *monitor* is designed to help the programmer build classes that act like traditional monitors for data access synchronization. Since C++ does not give syntactic support for monitors, a few

routines are included to make the construction of monitors possible. Each operation in the derived class must first call *enter()* to acquire the monitor, and must call either *exit()* or *signal()* to relinquish the monitor. The monitor can have any number of conditions associated with it. In calling the constructor for the monitor, the second argument tells how many conditions to create with the monitor; the default is 1.

The monitor has the usual condition operations, *wait()* and *signal()*, which have the standard interpretations (note that *signal()* also releases the monitor). In addition to those calls, the tasking package also supports two other condition operations. First, *signalandwait()* allows the programmer to signal on one condition and wait immediately on another condition without having to reacquire the monitor. Second, *waitevent()* is an application of the UNIX system call *select* with the *wait()* operation which allows the task to synchronize on external events along with monitor conditions.

As an example, we give a solution to the producer/consumer problem. In this standard synchronization problem, two types of tasks, *producers* and *consumers*, communicate by using a common buffer (of finite size). A producer may attempt to put data into the buffer, a consumer may attempt to take data out of the buffer. If there is no room in the buffer for a producer's data, then the producer will have to wait until space becomes available. If there is not enough data in the buffer to meet a consumer's request, then the consumer will have to wait until the data becomes available. A producer puts as much data into the buffer as possible, then notifies any waiting consumer that more data has arrived and, if it has more data to transfer, puts itself to sleep. A consumer takes out as much data as possible (either the amount it has requested or what is available, whichever is smaller), then notifies any waiting producers and puts itself to sleep if more data is needed. (Note that this solution correctly deals with the case in which a producer has more data to transfer than can be held by an empty buffer). A monitor representing the buffer and the associated produce and consume operations can be defined as:

```
// define buffer monitor.
struct buffer : monitor {
    char *buf ;
    char *in, *out ;
    int nfull, nempty ;
    int alldone ;
    int size ;

    buffer(int);
    ~buffer()      { delete buf ; }
    int consume(char*,int);
    void produce(char*,int);
    void flush();
};

buffer::buffer(int s) : ("buffer",2)
{
    size = s ;
    buf = new char[size] ;
    nfull = alldone = 0 ;
    in = out = buf ;
    nempty = size ;
}

int
buffer::consume(char* s, int n)
{
    int cnt, cnt2 ;
    int totcnt = 0 ;
    char *lin = s ;
```

```

int bytesleft ;

enter();

bytesleft = (int)(size - (out - buf));

while (n > 0) {
    while (nfull <= 0) {
        if (alldone) {
            exit();
            return(totcnt);
        }
        signalandwait(1, 0);
    }

    cnt = (nfull <= n) ? nfull : n ;
    if (cnt > bytesleft) {
        bcopy(out,lin,bytesleft);
        cnt2 = cnt - bytesleft ;
        bcopy(buf,lin+bytesleft,cnt2);
        out = buf + cnt2 ;
    } else {
        bcopy(out,lin,cnt);
        out += cnt ;
        bytesleft -= cnt ;
    }
    nfull -= cnt ;
    nempty += cnt ;
    lin += cnt ;
    totcnt += cnt ;
    n -= cnt ;
}
signal(1);
return(totcnt);
}

void
buffer::produce(char* s, int n)
{
    int cnt, cnt2 ;
    char* lout = s ;
    int spaceleft ;

    enter();

    spaceleft = (int)(size - (in - buf));

    while (n > 0) {
        if (nempty <= 0)
            signalandwait(0, 1);
        cnt = (nempty <= n) ? nempty : n ;
        if (cnt > spaceleft) {
            bcopy(lout,in,spaceleft);

```

```

    cnt2 = cnt - spaceleft ;
        bcopy(lout+spaceleft,buf,cnt2);
        in = buf + cnt2 ;
    } else {
        bcopy(lout,in,cnt);
        in += cnt ;
        spaceleft -= cnt ;
    }
    nempty -= cnt ;
    nfull += cnt ;
    lout += cnt ;
    n -= cnt ;
}
signal(0);
}

```

```

void
buffer::flush()
{
    enter();
    alldone++ ;
    signal(0);
}

```

With this definition of a bounded buffer, writing the routines which actually do the producing and the consuming is quite straightforward. We need not worry whether or not the routines are monitors since all of the details are abstracted away in the buffer.

```

producer::producer(buffer* buf) :("producer")
{
    int cnt ;
    char s[80] ;

    while (1) {
        if ((cnt = read(0,s,80)) <= 0)
            break ;
        buf->produce(s,cnt);
    }
    buf->flush();
}

```

```

// define consumer task.
struct consumer : task {
    consumer(buffer*);
};

```

```

consumer::consumer(buffer* buf) :("consumer")
{
    int cnt ;
    char s[4] ;

    while (1) {
        if ((cnt = buf->consume(s,4)) == 0)
            break ;
    }
}

```

```

    write(1,s,cnt);
    }
}

// Main task.
main(int argc, char* argv[], char* envp[])
{
    buffer *b    = new buffer(80);
    producer *p = new producer(b);
    consumer *c = new consumer(b);
}

```

### 2.3. The Queue Classes

Another method of communicating among tasks is to build a one-way pipeline between tasks for the orderly flow of data from one task to another. In the tasking package there is no class named *queue*: instead, there are two classes, *qhead* and *qtail*, which implement the two ends of a queue. This separation allows the producers and consumers of data to be defined without any dependence on each other. A queue can be created with two calls:

```

qhead qh ;
qtail *qt = qh.tail();

```

*qh.tail()* creates a *qtail* and connects it to *qh*. To move data through this queue, use the *qtail* operation, use *put()* to enter data into the queue, and uses the *qhead* operation, *get()*, to retrieve data from the queue. *put()* and *get()* operate on pointers to items of class *object* or items derived from class *object*. Normally, if *get()* operates on an empty queue, the task which issued the request is suspended until the queue is no longer empty. *put()* behaves similarly when the queue is full.

Queues have a maximum size arbitrarily set when the queue is created; the default maximum is 10000. There are also other semantics for handling empty and full queues. By setting a mode value for a *qhead* or *qtail*, the operations can return error messages or values which tell the task that called it of the queue's condition.

Let us look at how to handle the producer/consumer problem by using these queues for communication. Note how the type of data item passed through the queue is defined:

```

// define queue data item
struct item : object {
    int last_message ;
    char s[80];
};

struct producer : task {
    producer(qtail*);
};
producer::producer(qtail* qt) :("producer")
{
    for (item *i = new item ; fgets(i->s,80,stdin) ; i = new item) {
        i->last_message = 0 ;
        qt->put((object*)i);
    }
    i = new item ;
    i->last_message = 1 ;
    i->s[0] = 0 ;
    qt->put((object*)i);
}

```

```

struct consumer : task {
    consumer(qhead*);
};
consumer::consumer(qhead* qh) :("consumer")
{
    while (1) {
        item *i = (item*)qh->get();
        if (i->last_message)
            break ;
        printf(i->s);
    }
}

main(int argc, char *argv[], char *envp[])
{
    qhead* qh = new qhead ;
    qtail* qt = qh->tail();

    producer *p = new producer(qt);
    consumer *c = new consumer(qh);
}

```

The Tasking package also supports "splitting" queues in the middle and adding filters to alter the data stream. For example, to change all the lowercase letters to uppercase in the data in the previous example, we can define a task to do this change and add it into the data stream filtering the lowercase letters.

Here is how to split the queue to add the filtering routine. Suppose *qh* and *qt* are already defined:

```

qhead *newqh = qh->cut();
qtail *newqt = qh->tail();

```

*cut()* splits *qh* and returns a pointer to a new *qhead* which is connected to the tail of the old queue. This can be used as the head for the filter. Now the old head *qh* no longer has a tail associated with it, so it makes a new one, and this tail can be used as the output for the new filter. In this way, the queues have been split and a filter put in the data stream without having to notify the tasks currently using the queue.

Queues can also be spliced back together with the *splice()* operation. To undo what was done above, we can write:

```

newqt->splice(newqt);

```

*splice()* deletes the *newqh*, *newqt* and the queue associated with the tail, and connects their respective heads and tails together to make one queue again.

The producer/consumer problem can be changed as follows to add in the filter to make all lowercase letters uppercase. First define the filter task:

```

#include <ctype.h>

struct filter : task {
    filter(qtail*,qhead*);
};

filter::filter(qtail* qt, qhead* qh) :("filter")
{
    while (1) {

```

```

item *i = (item*)qh->get();
    // change all to upper case.
    char c ;
    int t = 0 ;
    while (c = i->s[t]) {
        if (isalpha(c) && islower(c))
            i->s[t] = toupper(c);
        t++ ;
    }
    qt->put((object*)i);
    if (i->last_message)
        break ;
}
}

```

Next, change the main task to split the queue so that the filter can be added, and add the filter:

```

main(int argc, char *argv[], char *envp[])
{
    qhead* qh = new qhead ;
    qtail* qt = qh->tail();

    // split queue to put new filter in it
    qhead* newhead = qh->cut();
    qtail* newtail = qh->tail();

    filter *t = new filter(newtail,newhead);
    producer *p = new producer(qt);
    consumer *c = new consumer(qh);
}

```

The queue structure a variety of applications. For example, it can be used as the basis for message-based systems in which a server accepts messages from client tasks and processes them or create other tasks to do so. For further discussion of how the queues work, see [Stroustrup 2].

### 3. The Threads System

The Threads system cheaply supports the concept of a *thread of control* (or *thread*), which is an independent unit of execution, capable of concurrent execution with other threads. Our implementation is on top of workstations running Berkeley UNIX<sup>4</sup> (it currently runs on Suns, MicroVAXes and Apollos) as well as on a shared-memory multiprocessor – the Encore Multimax. It has proved fast enough to satisfy the needs of several projects at Brown and is about to be distributed to researchers at other institutions. The programming interface provided by Threads insulates the user from such details as the number of processors being used: programs written for uniprocessors normally work correctly on multiprocessors. We provide a standard set of high-level programming abstractions and also provide facilities for the programmer to create his or her own abstractions.

The first version of Threads was completed in September 1985, and versions have been used by researchers other than the implementer since then. The first multiprocessor version of Threads was completed in March of 1987, two months after we acquired our Encore. This version has been used by others since April of 1987. A detailed tutorial on the use of the system is available [Doeppner 1]. [Doeppner 2] describes the design of the system.

---

<sup>4</sup>UNIX is a trademark of AT&T Bell Laboratories.

The notion of cheap concurrency is often known as “lightweight processes.” An early operating system that was based on this notion was Xerox’s Pilot system [Redell]. Other UNIX-based lightweight process implementations have been discussed in the past few years [Binding, Kepecs]. The system that comes closest to ours is Eric Cooper’s C-Threads [Cooper]. Recently an operating-system-supported notion of lightweight process, also known as a thread, has been implemented and used extensively at CMU as part of the Mach system [Tevanian].

What differentiates our system from all of the other UNIX-based systems is its complete support for systems concepts, including I/O, interrupts and exceptions. What differentiates our system from all of the other approaches to inexpensive concurrency is its support for concurrent programming abstractions – both the design of the particular abstractions we have implemented and how we allow the programmer to define new abstractions.

For a highly concurrent style of programming to be practical, threads, which support the concurrency, must be very inexpensive; the overhead required for creating, synchronizing and scheduling threads must be very low. A thread is not a traditional operating-system process, which are typically very expensive to create and very expensive to synchronize. One reason for this expense is that processes are much more than just threads of control. They entail (usually) a separate address space and other protection boundaries which are time-consuming to set up. Another reason for the expense of processes is that they are managed by the operating system kernel; requests to perform operations such as synchronization must be passed to the kernel over a user-kernel boundary that is typically fairly expensive to cross (for example, for Berkeley UNIX running on a MicroVAX<sup>5</sup> II, the overhead for a trivial system call is 200 microseconds).

In the Mach system [Tevanian], threads are supported in the kernel, but are cheap enough to qualify as lightweight processes (Mach threads are a lower-level abstraction than our threads). We are very interested in combining our approach with that of Mach, building our threads on top of Mach threads.

Currently all of our Threads system runs as user-mode code. This has resulted in a minimal overhead due to system calls and has allowed our system to be ported fairly easily to other UNIX systems.

Our implementation consists of two layers. The bottom layer, which is built on top of the UNIX process, implements the basic notion of a thread as an independent entity. A thread at this level presents a fairly low-level procedural interface which allows it to be manipulated directly. In the next layer, the thread abstraction is extended to supply the functionality needed to give the programmer the types of programming constructs expected in a high-level language; this layer can be thought of as the runtime library for such a language. A user of the Threads system may add additional layers, building on top of the lower layers by supplying additional procedures and adding additional fields to the thread data structures.

The functionality defined in the bottom layer includes scheduling and context switching, low-level synchronization, interrupt processing, exception handling and stack handling. In addition, this layer provides the routines employed for protection from interrupts and for the locking of data structures when used on a multiprocessor.

In the second layer we build up a set of programming constructs from the low-level interface of the bottom layer. These constructs allow threads to synchronize their execution (using semaphores or monitors [Hoare]), to perform I/O, and to respond to exceptions and interrupts. Exception handling is integrated with synchronization so that, for example, when a thread is forced out of a synchronization construct by an exception, the state of the synchronization construct is “cleaned up” so that it will continue to operate correctly. The programmer may choose a variety of ways of dealing with interrupts. For example, a thread may be created in response to an interrupt or an interrupt may cause an exception to occur in a specified thread.

---

<sup>5</sup>MicroVAX is a trademark of the Digital Equipment Corporation.

Recently we have added support for a different type of thread, called a “microthread,” which is an extremely low-overhead thread that is useful for such fine-grained applications as parallelizing do loops. They are allocated in groups, so that a set of microthreads will jointly invoke a “parallel function”, each of them determining what subtasks they are responsible for and then performing these subtasks. In our Encore implementation, approximately 185 microseconds are required to create a standard thread (vs. 10 – 20 milliseconds to create a process in UNIX) and approximately 5 microseconds are required to start a set of 10 microthreads. We have achieved 90% speedup using microthreads to solve an edit-distance problem. The integration of microthreads into C++ is nearly complete and will be discussed in a subsequent paper.

#### 4. Use of Threads in the Tasking Package

The concept of parallel tasks in the tasking package maps directly to threads of control in the Threads system. Threads offers all of the constructs needed to create, schedule, synchronize, and maintain tasks. The rest of this section will discuss how threads are used to implement the different parts of the tasking package, including, tasks, queues, monitors, and task wait queues.

##### 4.1. Task Creation and Execution

A thread consists of a stack and a thread control block which is scheduled on the run queue in the Threads system. For a task to be created, the data structure which controls the task must be associated with the thread’s stack and control block. This association is done at the time the constructor for a class with a base class *task* is called. In the constructor, space for a stack and thread control block is allocated and the calling history on the current task’s stack is modified. The new stack is set up so that new thread of control will execute the derived constructor of the task class being created. The current thread’s stack is also modified so that it will return directly to the operation which called for the creation of the new task.

All execution in a program using the Tasking Package is done inside the context of some task, even the *main()* routine. The *main()* that is defined by the programmer is actually made into its own task by being called by a library-supplied *main()* routine which also sets up the environment for the Threads system and calls the Threads startup routine *Threadgo*. At startup time, the Threads environment can be changed by the user by passing arguments on the command line which are interpreted in library-supplied *main()*. These arguments primarily affect the members of the class *TASKenv* which defines certain default values and aspects of the Threads environment.

All scheduling issues, including, priorities and preemptive scheduling is handled at the Threads level with no intervention from Tasks. Any changes in the behavior is handled by making Threads calls, such as *THREADsetpriority()* for changing priority and *THREADstopclock()* for turning off preemptive scheduling.

##### 4.2. Task Queues

Task queues are a straightforward use of the Threads queues facility. When a task does a *wait()* on another task it uses Thread’s *THREADmovetowaitq()* routine to suspend the thread and put it on a queue. Likewise, the *THREADqueueenext()*, *THREADpullfromq()*, and *THREADmovetorunq()* routines are used to wake up waiting tasks. Also, whenever any of the queues associated with *qhead* or *qtail* need to suspend tasks because of over/underflow conditions, the same Threads routines are used to suspend the tasks.

##### 4.3. The Monitor Class

The *monitor* class is built directly from the monitors of the Threads system: each of the calls in the class map directly to a corresponding Threads call. We would prefer not to require the programmer to supply the *enter* and *exit* calls, since these should be supplied implicitly as part of the abstraction, but we found no convenient method for doing this. However, users of classes derived from the *monitor* class can still view the class as having all of the properties associated with a monitor.

## 5. Conclusion

The tasking package is a useful set of C++ classes which gives programmers a set of abstractions for creation, maintenance, and synchronization of separate threads of control in a parallel programming environment. The Threads run-time system provides a rich set of constructs for building parallel tasks in a single UNIX process, making implementation of the tasking package an easily manageable job.

## 6. References

- [Binding] Binding, C., "Cheap Concurrency in C," *SIGPLAN Notices*, Vol. 20, No. 9, September 1985.
- [Cooper] Cooper, E.C., Draves, R.P., "C Threads," Draft of Carnegie Mellon University/Computer Science Report, March 1987.
- [Doepfner 1] Doepfner, T.W. Jr., "A Threads Tutorial," Computer Science Technical Report CS-87-06, Brown University, March 1987.
- [Doepfner 2] Doepfner, T.W. Jr., "Threads – A System for the Support of Concurrent Programming," Submitted for publication, also Computer Science Technical Report CS-87-11, Brown University, June 1987.
- [Hoare] Hoare, C.A.R., "Monitors: An Operating System Structuring Concept", *Communications of the ACM*, Vol. 17, No. 10 (1974).
- [Kepecs] Kepecs, J., "Lightweight Process for UNIX/Implementation and Applications," *USENIX Association Summer Conference Proceedings*, June 1985.
- [Redell] Redell, D.D., Dalal, Y.K., Horsley, T.R., Lauer, H.C., Lynch, W.C., McJones, P.R., Murray, H.G. and Purcell, S.C., "Pilot: An Operating System for a Personal Computer," *Communications of the ACM*, Vol. 23, No. 2, February 1980.
- [Stroustrup 1] Stroustrup, B., *The C++ Programming Language*, Addison-Wesley, 1986.
- [Stroustrup 2] Stroustrup, B., "A Set of C++ Classes for Co-routine Style Programming", AT&T Bell Laboratories Computer Science Technical Report, Available with Release notes for 1.2.1 C++.
- [Tevanian] Tevanian, A., Rashid, R.F., Golub, D.B., Black, D.L., Cooper, E. and Young, M.W., "Mach Threads and the UNIX Kernel: The Battle for Control," *USENIX Association Summer Conference Proceedings*, June 1987.