1560

Bell Laboratories **Cover Sheet for Technical Memorandum**

Title- **A Set of C Classes for Co-routine Style Programming**

Date- **May 6, 1980**

TM- **80-1273-4**

Other Keywords- **Task**
Queue
Monitor
Timer
Simulation

Author	Location	Extension	Charging Case- **39199**
Bjarne Stroustrup	**MH 2C-521**	**7393**	Filing Case- **39199-11**

ABSTRACT

Some programs are most naturally expressed as a set of relatively independent activities communicating to achieve a common goal. Each activity, here called a *task*, has its own locus of control, a program to execute, and its own private data. Tasks can communicate by message passing, by explicit sharing of data, by using monitors, or by data pipes.

This memorandum describes C classes for a range of styles of multi-programming techniques in a single language, single address-space environment. Class *task* is a base class for representation of an activity in a multi-programmed system. A task can be suspended and resumed without interfering with its internal state.

Class *qhead* and class *qtail* enable a wide range of message passing and data buffering schemes to be implemented simply, and class *monitor* is a facility for controlled sharing of data.

Tasks execute in a simulated time frame presented by the variable *clock*. Objects of class *timer* provide a convenient and efficient facility for using the clock.

The implementation and use of these concepts rely heavily on the idea of derived classes. Familiarity with both the C language [5] and the C class concept [8] would be an advantage for the reader. Classes are used to ensure cleaner interfaces and better type checking than could be achieved in C without classes.

A task can either be declared to have its own C run time stack of specified size, thereby achieving the fastest possible suspension and resumption, or it can be declared to use store less extravagantly at the cost of some run time. Tools for evaluating the space/time tradeoffs for a particular set of tasks are provided.

Pages Text 15	Other 5	Total 20
No. Figures 0	No. Tables 0	No. Refs. 9

DISTRIBUTION
(REFER GEI 13.9-3)

COMPLETE MEMORANDUM TO

CORRESPONDENCE FILES

OFFICIAL FILE COPY
PLUS ONE COPY FOR
EACH ADDITIONAL FILING
CASE REFERENCED

DATE FILE COPY
(FORM E-1328)

REFERENCE COPIES

<AHO, ALFRED V
<BALENSON, CHRISTINE M
<BARCH, ROBERT V
<BECKER, RICHARD A
N,W STANLEY
<C...EY, LORINDA L
<FELDMAN, STUART I
<FRASER, A G
<GLASSER, ALAN L
<GOLDSTEIN, A JAY
<GRAHAM, R L
+HANNAY, N B
<JOHNSON, STEPHEN C
<KEESE, W M
<KERNIGHAN, BRIAN W
<LUDERER, GOTTFRIED W R
<MARANZANO, J F
+MC DONALD, H S
<MC GILL, R
MC.LROY, M DOUGLAS
<MENNINGER, E E
<MORGAN, SAMUEL P
+PRIM, R C
<RALEIGH, T M
<R...OLE, GUY G
<S... INSKI, THOMAS C
T... AY, MILTON P
<WEINBERGER, PETER J
<YANNAKAKIS, MIHALIS
25 NAMES

COVER SHEET ONLY TO

CORRESPONDENCE FILES

4 COPIES PLUS ONE
COPY FOR EACH FILING
CASE

AAGESEN, JOHN
AARONSON, STEVE M
ABATEMARCO, THERESA M
ABATZ, JOSEPH
ACKERMAN, A FRANK
... RMAN, J T
A... OFF, JOHN M
AHRENS, RAINER E

COVER SHEET ONLY TO

<AKHTAR, SUDHIR S
ALBAGLI, V R
ALBERALLA RICHARD J
ALCALAY, D
ALEXIS, A D, JR
ALFONS, FREDERICK
ALLEN, JAMES C, JR
ALLISON, ... E, JR
AMITAI, M
AMMIRATI, J I
AMORY, ROBERT A
AMRON, IRVING
ANDERSON, FREDERICK I
ANDERSON, KATHRYN J
ANDERSON, R E
ANDERSON, R E
ANDREWS, W J
ANTOLICK, DAVID R
ARCHELLI, CHARLES J
APPELBAUM, MATTHEW A
ARCHER, RUSSELL E, JR
ARMSTRONG, J B
ARMSTRONG, J C, JR
ARNOLD, GEORGE J
ARNOLD, JAMES
ARNOLD, PHILLIP
ARNOLD, THOMAS F
ARTHURS, E
ARVILSON, ...
ASELTINE, EDWARD J
ASMUTH, RICHARD L
ASTHANA, ABHAYA
ATAL, BISHNU S
AXELSON, A L
BABU, BRAJESH RATILAL
BAILEY, CATHERINE T
BAKER, BRENDA S
BAKER, DOHN
BAKER, MATTHEW E
BALDWIN, GEORGE L
BALLANCE, ROBERT L
BALLARD, E D, JR
BALLY, S J
BARBATO, ROBERT E
BARNS, R L
BAROFSKY, ALLAN
BARR, DAVID I
BARTAGLIA, FRANCES
BASER, BARBARA M
BAUER, L C
BAUER, BRIAN L
BAUGH, C R
BAVIER, WILLIAM F
BAXTER, LESLIE A
BAYER, D L
BEACH, MILTON
BECERRA, PEDRO D
BECKER, ... LEE A
BECKER, ... L
BECKER, SHELDON J
BECKLEY, KEITH A

COVER SHEET ONLY TO

BELANGER, DAVID
BENCO, DAVID S
BENISCH, JEAN
BENNETT, RAYMOND C
BENNETT, RICHARD L
BENOWITZ, N
BENSLEY, JAMES EDWARD
BERENBAUM, ALAN
BERGLAND, G D
BERKMAN, M A
BERKLEY, DAVID A
BERN, DONALD I
BERNHARDT, RICHARD C
BERNOSKE, BEVERLY G
BERNSTEIN, L
BERNSTEIN, PAULA S
BERZINS, ALEXANDER E
BEYER, JEAN-DAVID
BHATIA, RAJIV
BIANCHI, M H
BICKFORD, NEIL B
BILASH, TIMOTHY D
BILOWOS, R M
BISHER, EMMA S
BISHOP, J DANIEL
BISHOP, THOMAS P
BISHOP, VERONICA L
BITTNER, B B
BITTRICH, MARY A
BLACKMAN, NANCY
BLAKE, GARY E
BLAZARS, S G
BLECHMAN, RONALD
BLEICHER, E
BLEYER, JOSEPH
BLINN, J
BLOSSER, PATRICK A
BLUM, MARION
BOCHICCHIO, EDWARD J
BOCK, NANCY A
BODDIE, JAMES A
BODEN, F J
BOEHM, EARL
BOESE, J C
BOGART, F J
BOGART, THOMAS
BOIVIE, RICHARD H
BOHANNE, ...
BOND, P L
BOND, ...
BORTSON, EILEEN A
BOSALE, SURESH
BOSE, DEBASISH
BOSTON, RONALD A
BOSWELL, PAUL S
BOTHA, HERMAN J
BOURNE, STEPHEN
BOWIES, ...
BOYCE, W E
BOYLE, PHYLLIS
BOZA, ...

COVER SHEET ONLY TO

BRADFORD, EDWARD G
BRADLEY, M HELEN
<BRANDT, RICHARD B
BRAUN, DAVID A
BREILAND, JOHN E
BRENSKI, EDWIN F
BRESLIN, RENEE A
BRIGGS, GLORIA A
BRITT, WARREN D
BROAD, MARTHA M
BRONSTEIN, N
BROOKS, CATHERINE ANN
BROSS, JEFFREY D
BROVMAN, INNA
BROWNING, JASON DAVID
BROWN, ELLINGTON L
BROWN, LAURENCE MC FEE
BROWN, MARK S
BROWN, STUART G
BROWN, W R
BRYANT, DAVID J
BUCK, I D
BULLEY, R M
BURGESS, JOHN T, JR
BURG, F M
BURIC, MILORAD R
BURKE, MICHAEL E
BURKE, T J
BURNETTE, W A
<BURNETT, DAVID S
BURNET, ROSE M
BUROFF, STEVEN J
BUTLETT, DARRELL L
BUTTON, BEVERLY
BYCRICK, ROBERT C
BYRNE, EDWARD R
CALL, PETER F
CALVERT, KENNETH L
CAMPBELL, JERRY H
CAMPBELL, MICHAEL R
<CANADAY, FORD H
CANDREA, RONALD D
CARNETA, PHILLIP E
CAREY, J E
CARREN, JOHN E
CARRIGAN, RAYMOND J
CARR, DAVID C
CARTER, DONALD H
CASPERS, BARBARA E
CATO, H E
CAVINESS, JOHN D
CELLER, GEORGE K
CERMAK, I A
CHAFFEE, N F
CHAI, D T
CHAMBERS, B C
<CHAMBERS, J M
CHANG, JO-MEI
CHAO, C
CHAPMAN, LESLEY D
CHAPMAN, W D, JR

COVER SHEET ONLY TO

CHAPPELL, S G
CHARLES, JOYCE
CHEE, T
CHELLIS, ALICIA L
CHENG-QUISPE, ENRIQUE
CHENG, Y
CHEN, E
CHEN, PING C
CHEN, ROBERT
<CHEN, STEPHEN
CHESSON, GREGORY L
CHE, HER-DAW
CHIN, AUGUSTIN Y
CHI, GOU-CHUNG
CHODROW, M M
CHONG, PHEE
CHOU, PAO-LO P
CHRISTENSON, DENNIS A
CHRIST, C W, JR
CHUNG, MICHAEL
CHUNG, TSUNG-JEN BEN
CICHINSKI, STEVEN
CIEMINSKI, DEBRA F
CIESLAK, THOMAS J
CLARK, C A
CLARK, DAVID L
CLAYTON, D P
CLIFFORD, COURTENAY B
CLINE, LAUREL M I
CLINE, TERRY W
CLOUTIER, J
COATES, KAREN E
COFFEY, JOHN R
COHEN, AARON S
COHEN, DAVID
COHEN, HARVEY
COHEN, HARVEY S
COHEN, KAREN L
COHEN, NEIL B
COLBY, ROBERT W
COLEMAN, ELANE
<COLE, LOUIS M
COLE, MARILYN C
COLLICOTT, R B
CONDON, J H
CONKLIN, DANIEL L
CONNAGHAN, ROBERT
CONNERS, RONALD E
COOK, DIANA
COOK, T J
COOPER, ARTHUR E
COOPER, MICHAEL E
COPP, DAVID H
COTTINGHAM, CHARLES F
COTTRELL, JENNIE L
COVINGTON, RALPH L
COWELL, ARLENE D
CRAGUN, DONALD W
CRAGUN, JOAN
CRAIG, JOHN E
CRISTOFOR, EUGENE
...
1319 TOTAL

+ NAMED BY AUTHOR > CITED AS REFERENCE < REQUESTED BY READER (NAMES WITHOUT PREFIX WERE SELECTED USING THE AUTHOR'S SUBJECT OR ORGANIZATIONAL SPECIFICATION AS GIVEN BELOW)

MERCURY SPECIFICATION ...

COMPLETE MEMO TO:
127-SUP

COVER SHEET TO:
12-DIR 13-DIR 127

COPLSI - COMPUTING/PROGRAMMING LANGUAGES/COMPILATION
UNPLCI - C LANGUAGE

TO GET A COMPLETE COPY:

1. BE SURE YOUR CORRECT ADDRESS IS GIVEN ON THE OTHER SIDE
2. FOLD THIS SHEET IN HALF WITH THIS SIDE OUT AND STAPLE.
3. CIRCLE THE ADDRESS AT RIGHT. USE NO ENVELOPE.
4. INDICATE WHETHER MICROFICHE OR PAPER IS DESIRED.

BD CORRESPONDENCE FILES
HO 1A127

TM-80-1273-4
TOTAL PAGES 20

PLEASE SEND A COMPLETE

( ) MICROFICHE COPY ( ) PAPER COPY

TO THE ADDRESS SHOWN ON THE OTHER SIDE.

Bell Laboratories

Subject: A Set of C Classes for Co-routine Style Programming
Case- 39199 -- File- 39199-11

date: May 6, 1980

from: Bjarne Stroustrup

TM: 80-1273-4

MEMORANDUM FOR FILE

Introduction

Some programs are most naturally expressed as a set of relatively independent activities communicating to achieve a common goal. Such activities, here called *tasks*, must be able to execute in parallel with each other and communicate through means convenient to the chosen style of task usage.

Facilities for multi-thread computation can be provided in the semantics of a language, as it is done in Concurrent Pascal [1] and Mesa [6], or a language without such facilities can be augmented using special run-time support systems and library functions, as has been done for BCPL [7] and the C language [3]. The use of C classes represents an intermediate approach by extending the language using features incorporated into language to enable and ease such extensions. This approach was pioneered by Simula67 [2].

The tools presented here provide the basic facilities for several styles of multi-thread programming in a single language, single address-space environment. The underlying facility is a simple and efficient tasking system with non-preemptive scheduling. That is, a task will only be suspended on its own request, so no "system policy" can be enforced without the cooperation of all tasks. In contrast to pure co-routine systems, however, a standard scheduler provides a framework for processor sharing and communication between tasks. A concept of simulated time is implied by this scheduler.

The task system is intented for applications, like event driven simulations, where tasks are used to express a quasi-parallel structure for a single program. The task system is not intented for handling real parallelism of some underlying real-time system. Consequently, no facilities are provided to map interrupts and other real-time events into the concepts provided by the task system. A unit a simulated time can represent any amount of real time, and it is possible to compute without consuming simulated time.

In the following sections the task system will be described in some detail, and examples of its use will be given. The classes used in the task system are presented. This allows a detailed and specific discussion of the concepts involved, but it unfortunately also implies that some concepts cannot be explained in detail where they are first mentioned. For a description of the C class concept see reference 8.

Tasks

The public part of the declaration of class *task* looks like this*:

```
class task : object
{
    void        new(char *, short, short);
public:
    char *      t_name;

    short       rdstate();

    int         result(class task *);
    void        cancel(int);
    void        sleep();
    void        delay(int);

    void        wait(class object *);
    short       waitvec(class object **);
    short       waitlist(class object *, class object *, class object *);

    void        printout(short);
};
```

A task is a locus of control, a virtual processor. It can only be used as a base class. A task is said to execute the program supplied as a derived class's *new* function. The most basic feature of an object of class *task* is that it can be suspended and later resumed so that several tasks can run in quasi-parallel. Most class *task* functions are conditional or unconditional requests for suspension.

A task can be in one of three states:

RUNNING:

The task is executing instructions or it will be scheduled to do so without further intervention from other tasks.

IDLE:

The task is not RUNNING, but it can be transferred to the RUNNING state by some suitable action.

TERMINATED:

The task has completed its work. It cannot be resumed, but its result can be retrieved.

The function *rdstate()* returns a value indicating the state of its task.

Class *task's* first argument is a string of characters used to initialize the *t_name* pointer. This name can be used to provide more readable output and does not affect the behaviour of the task. It is used by the debugging aids and error reporting functions described below. The two other class *task* arguments are tuning parameters and will be described below. If an argument is zero a system default will be used.

---

* The class *object* used in the declaration of class *task* is a simple base class used by all classes in the task system. It contains some of the pointers used by the task system's internal "house-keeping", and also a value indicating the type of the object. Class *object* is presented in appendix A.

A class specifies a data structure and a set of functions which can manipulate it. When an object of a class is created that class's *new()* function will be called with the arguments specified for the new class object. Class objects can be created either by declaring *auto* or *static* class variables, or by generating them on the free store using the *new* operator.

When one class is derived from another, like class *task* from class *object* here, the derived class will have all the attributes of the base class in addition to its own.

The simplest example of the use of tasks is where one task creates another to run in parallel with itself. Later the creator can obtain the result produced by the "secondary" task. For example, a task which counts the number of spaces in a string could be declared.

First a class *spaces* must be declared. This declaration specifies the data structure associated with each object of class *spaces,* and also which functions may operate on this structure.

```
class spaces : public task
{
        short     new(char *);
};
```

In the case of class *spaces* this declaration is trivial. It states that *spaces* is derived from class *task* so that each object of class *spaces* becomes an independently scheduled entity that can be suspended and later resumed. The program for the task is provided by the derived class's *new()* function, in this case *spaces.new().* This use of *new()* resembles the use of *main()* in a C program.

```
short spaces.new(s) char * s;
: (0,0,0)    /* argument list for task.new() */
{
        short i = 0;

        while (*s) if (*s++ == ' ') i++;

        return i;
};
```

A *spaces* task can now be created and used like this:

```
class task * p = new class spaces("a line with four spaces");

 ....

int count = p->result();
```

As the result of the creation of the new *spaces* task its *new()* function is called, and the two tasks now run in parallel. When the "primary" task calls the class *task* function *result()* it will, if necessary, wait for the "secondary" task to complete. *Result()* returns the value returned by *new(),* that is, in this example *spaces.result()* returns the value *4* computed by *spaces.new().*

The *new()* of a class derived from class *task* can either be *void* or return a value. If it is *void* then *result()* will return some arbitrary value, otherwise the returned value will be assigned to an *int* using the cast *(int)* and the resulting value will be the one returned by *result().* When *new()* returns, the state of the task becomes TERMINATED, and any task waiting for it to complete will become RUNNING. A task waiting for another task to complete, for example as the result of executing *result(),* is an example of an IDLE task.

Applying the *delete* operator to a pointer to a non-TERMINATED task causes a run-time error*. This avoids some potentially nasty bugs, but occasionally there is a need to stop a running task. For example, the result of a task could have been found not to be needed after all, or tasks may be created which simply produce a steady stream of output until they are explicitly stopped.

Cancel() puts its task into the TERMINATED state from which it cannot be resumed. Pointers to the task will remain valid, however, so that its public functions, for example *result(),* can be used to retrieve information from it. When *result()* is executed for a cancelled task it will return the value of *cancel's* argument.

* The handling of run time errors will be described below.

When a task enters the TERMINATED state through the use of *cancel()* it will wake up all tasks waiting for it to complete just as if a normal completion (return from *new()*) had occurred.

The class *task* function *sleep()* suspends the task unconditionally without specifying what is supposed to cause it to be resumed.

The *extern* class *task* pointer *thistask* denotes the currently active task. If no tasks have been created its value is *0*. It is illegal to assign to *thistask*. The use of *thistask* enables the class *task* functions to be used from *extern* functions without explicit passing of the current task's *this* pointer.

It is not possible to have only one task. Therefore, when the first task is created in a program another task is implicitly created. Its *new()* is *main()*, and its name is "*main*". It can be suspended and resumed like any other task. Please remember that a return from *main()* terminates a *C* program. If the "*main*" task should be terminated when there are other tasks which should be left running then *cancel()* can be used. For example,

```
thistask->cancel(0);
thistask->sleep();
```

can be executed in *main()* to ensure that it will execute further. The program will then run on until no more tasks are or can become RUNNING.

The task system does not provide a garbage collector. It is left to the programmer to ensure that pointers to deallocated store are not used.

Queues

A *queue* is a type of storage that is organized so that objects are retrieved from it in the order in which they were inserted into it. A queue has a *head* from which data is retrieved and a *tail* to which data is added. With a little elaboration this basic type of data structure makes an excellent inter-task communication facility.

There is a function *put()* which adds an object to the *tail* of a queue and a function *get()* which retrieves an object from the *head* of a queue. In this context an object is an object of any class derived from class *object*. To allow explicit separation between the source of data and the recipient of data there is no class "queue" available to a user, rather two classes *qhead* and *qtail* provide the services needed. The public part of the declaration of class *qhead* looks like this:

```
class qhead : object
{
        void            new(short,short);
public:
        class object *  get();
        void            putback(class object *);

        short           rdcount();
        short           rdmode();
        short           rdmax();
        void            setmode(short);
        void            setmax(short);

        class qtail *   tail();

        class qhead *   cut();
        void            splice(class qtail *);

        void            printout(short);
};
```

A queue has a private variable called *max* that specifies the maximum number of objects which can be on the the queue at a time. A *qhead* has a variable called *mode* which determines what happens when *get()* is executed on an empty queue. The two arguments to class *qhead* specifies initial values for these variables.

```
class qhead * h = new class qhead(WMODE,10);
```

creates a new queue and *h* points to its *qhead*. To obtain a *qtail* for an existing queue execute the *tail()* function for the *qhead:*

```
class qtail * t = h->tail();
```

the tail will initially have the same mode as the head. The queue identified by *h* and *t* could now be used as a one way inter-task communication channel by giving them as arguments to two new tasks:

```
class producer(t);
class consumer(h);
```

The task *producer* can now *put()* objects to its qtail *t* and *consumer* can *get()* those objects from its qhead *h*.

As mentioned above, the *mode* argument to class *qhead* controls what happens when *get()* is executed on an empty queue. In EMODE this causes an run time error. In WMODE a task executing a *get()* on an empty queue will wait on that queue, that is become IDLE until the queue becomes non-empty.

Typically one task will *put()* into a queue and another *get()* from it, but a queue can also be used with multiple tasks inserting and/or removing objects. A queue can also be used by a single task for its own private purposes.

The public part of the declaration of class *qtail* is similar to that of class *qhead*. The two classes complement each other, and together they provide a representation of the general idea of a queue:

```
class qtail : object
{
        void            new(short,short);
public:
        void            put(class object *);

        short           rdspace();
        short           rdmax();
        short           rdmode();

        void            setmax(short);
        void            setmode(short);

        class qhead *   head();

        class qtail *   cut();
        void            splice(class qhead *);

        void            printout(short);
};
```

A *qtail's mode* controls what happens on queue overflow in the same way as *qhead's mode* controls what happens on queue underflow. For example, when a task executes *put()* on a full queue where the *qtail* is in WMODE, then that task will be suspended waiting for a *get()* on the head.

The mode of a *qhead* or a *qtail* can be inspected by *rdmode()* and changed at any time by *setmode()*. The *modes* of a queue's *qhead* and *qtail* need not be the same.

Similarly the maximum number of objects which can be on a queue can be examined by *rdmax()* and changed by *setmax()*. Decreasing the *max* below the current number of objects on the queue is legal. Doing this simply implies that no new objects can be *put()* on the queue until the queue has been drained below the new limit.

The *qhead* function *rdcount()* returns the current number of objects in a queue, and the *qtail* function *rdspace()* returns the number of objects which can be inserted into a queue before it becomes full.

The objects inserted and deleted from a queue can be objects of any class derived from class *object*. It is up to the programmer to define types of objects suitable for each application*. For example, one could define a class *message* as follows:

```
class message : object
{
public:
        short           r_operation;
        short           r_arg1;
        short           r_arg2;
        class qtail *   r_reply;
};
```

A message, that is an object of class *message*, describes an operation *r_operation* that is to be performed by the recipient of the message. Arguments for this operation can be passed as *r_arg1* and *r_arg2*, and the result of the operation is to be returned as a message on the queue denoted by *r_reply*.

A task serving requests presented as messages on a queue can be defined as follows:

```
class server : task
{
        void     new(class qhead *);
};
```

The message queues are assumed to be in WMODE.

* The task system use objects of types TASK, QHEAD, QTAIL, MONITOR, and TIMER.

```
void server.new(name,in)
char * name;
class qhead * in;
: (name,0,0)    /* arguments for server's base class task */
{
        for(;;) {
                class message * req =   in->get();
                class queue * reply =   req->r_reply;
                short res =             VALUE;
                short val;

                switch (req->r_operation) {
                        case PLUS:
                                val = req->r_arg1 + req->r_arg2;
                                break;
                        case MINUS:

                        ...

                        default:
                                res = ERROR;
                };
                req->r_operation = res;
                req->r_arg1 = val;
                reply->put(req);
        };
};
```

This style of server has proved useful in many context. In particular, it is the backbone of many "message-based systems". In this particular example a server, that is an object of class *server*, and the queue on which it depends can be declared:

```
class qtail * rq = new class qtail(WMODE,10);
class qhead * rply = new class qhead(WMODE,10);
class qtail * reply_to = rply->tail();

class server * ser = new class server("plus and minus",rq->head());
```

Other tasks can now communicate with this particular server through the queues. For example:

```
for (;;) {
        class message mess;

        mess->r_operation = nextop();
        mess->r_arg1 = nextarg1();
        mess->r_arg2 = nextarg2();
        mess->r_reply = reply_to;

        rq->put(&mess);
        mess = rply->get();

        if (mess->r_operation == ERROR) error();
};
```

The *qhead* function *putback()* puts its argument back at the head of the queue, that is

```
class qhead qh(WMODE,10);

class object * oo = qh.get();
qh.putback(oo);
oo = qh.get();
```

will assign the same object to *oo* twice. *Putback()* has proved to be a useful function in many systems in the past, and it also allows a *qhead* to operate as a stack. When *putback()* is used the task executing it competes for queue space with tasks using *put()* on the queue's tail. A *putback()* to a full queue causes a run time error regardless of the mode of the *qhead*.

Waiting

Functions like *task.result()*, *qhead.get()*, and *qtail.put()* each provides a way of waiting for one single specific event to happen. More general facilities are sometimes needed. The class *task* function *wait()* provides a way of waiting on an arbitrary object. For example, if *taskp* is a pointer to a task then

```
wait(taskp);
```

will suspend the task executing it until the task denoted by *taskp* becomes TERMINATED.

Each class derived from class *object* which is ever going to be "waited on" must have some rules associated with it specifying under which conditions a task executing a *wait()* for it will be resumed. The rules for class *task*, for class *qhead*, and for class *qtail* have been stated.

The conditions for wakeup are reflected in state changes in the objects, and are not just transitory unrecorded signals. For example, if a task executes a *wait()* for a non-empty *qhead* it will immediately continue, that is the condition for returning from a *wait()* for a *qhead* is that the queue is non-empty, not a brief state change from empty to non-empty. Rules of this type simplify programming considerably by eliminating race conditions.

The class *task* functions *waitvec()* and *waitlist()* suspend a task waiting for one of a list of objects, for example to wait for messages to arrive on one of a number of *qheads*. *Waitlist()* takes a list of object pointers terminated by a zero as argument, for example:

```
class qhead * q1;
class qhead * q2;

 ....

short who = waitlist(q1,q2,0);
```

will suspend the task executing it until either *q1* or *q2* is non-empty. If either is non-empty when *waitlist()* is executed the task will continue immediately.

The value returned is the position in the list of the object that caused the return from the wait, that is if *q2* caused the task to resume the value *1* will be assigned to *who*. Positions are numbered starting from *0*. *Waitlist()* can take any number of arguments. The degenerated example

```
waitlist(0);
```

causes unconditional suspension of the task executing it without any guarantee of later resumption. It is equivalent to *sleep()* and *wait(0)*.

Please note that one should not assume that because *waitlist()* returns a particular value indicating one object as the cause of resumption none of the other objects are "ready". The value returned by *waitlist()* only indicates what is known to have happened, and it does not exclude other independent possibilities. On the other hand, even if *waitvec()* indicates a particular object, that object cannot in all circumstances be assumed to be "ready". For example, if two tasks are taking objects from the same *qhead* each using *waitvec()* to wait for several objects then returning from *waitvec()* with an indication that the queue has become non-empty does not guarantee that it is still non-empty.

Because every class in the task system allows non-blocking examination of the conditions which might lead to suspension using the three wait functions the value returned by *waitvec()* can always be ignored. The information it conveys can always be obtained by direct inquiry. In many cases, however, the value returned can be trusted and used to write simpler, more efficient programs.

Waitvec() takes the address of a vector holding a list of object pointers, for example:

```
class object * vec[] = { q1, q2, 0 };

short who = waitvec(vec);
```

is equivalent to the previous example.

System Time and Timers

The *long* variable *clock* measures simulated time. It is initialized to zero. It is illegal to assign to *clock*.

The *task* function *delay* suspends a task for a specified time. That is,

```
long t = clock;
delay(n);
actual_delay = t - clock;
```

will assign the value *n* to *actual_delay*. *Delay()* is useful for representing service delays in simulations. While a task is delayed in this way its state is still RUNNING, but it will not be affected by the actions of other tasks except if *cancel()* is used on it.

A *timer* provides a facility for implementing time-outs and other time dependent phenomena.

Class *timer* has this declaration:

```
class timer : object
{
        void    new(int);
public:
        short   rdstate();
        int     result();

        void    reset(int);
        void    cancel(int);

        void    printout(short);
};
```

A timer is quite similar to a task with *new()* function consisting of the single statement

```
delay(d);
```

that is, when a timer is created it simply waits for the number of time units given to it as its argument, and then wake up any tasks waiting for it.

A *timer's* state can be either RUNNING or TERMINATED. This state can be inspected by using *rdstate()*.

A common use of timers is to wait for a task and a timer, for example, to wait for the completion of a task handling an input operation and also on a timer. The timer ensures that the waiting task will eventually be resumed even if the input operation is never completed*:

```
class timer * tt = new class timer(15);   /* 15 time units delay */

short res = waitlist(io_ptr,tt,0);

switch (res) {
        case 0:
                /* normal completion of i/o */

                ...

                break;
        case 1:
                /* time out occurred */

                ...

                break;
        default:
                error(IMPOSSIBLE);
};
```

The class *timer* function *result()* is very similar to *task.result()*. They differ only in that the value returned by *timer.result()* undefined unless *cancel()* was used. In the same way *timer.cancel()* is identical to *task.cancel()*.

The function *reset()* re-sets the timer delay to the value of its argument. This makes repeated use of timers possible. A timer can be *reset()* even when it is TERMINATED.

* In a quasi-parallel system this will only be true provided no infinite loop without task system calls exists. Such a loop constitutes an error that only a system with true parallelism can recover from.

Deleting a RUNNING timer causes a run time error.

A unit of simulated time can be used to represent any unit of real time. Only use of *delay()* causes the *clock* to advance.

More about Queues: Cutting and Splicing

One of the most convenient and powerful ways of using tasks involves tasks defined to do a transformation on a data stream. Such a task is called a filter. It has an input queue and an output queue and tasks at the "other ends" of these queues tend to view these queues plus the filter as one entity. The data source simply sees an output queue that is being emptied at some rate, and the task at the receiving end sees an input queue being filled. In other words, a task sees only its input and output queues and cares little about the "internal organization" of the programs that operate on the other ends of those queues.

For example, one task could produce a stream of lines of characters, that is objects of class *line*, and another expect an input stream consisting of words, that is objects of class *word*. A filter that handles the conversion could be defined and used like this:

```
class line_to_word : task
{
        void            new(class qhead *, class qtail *);
        class word *    next_word(class line *);
};

void line_to_word.new(in_q,out_q);
char * name;
class qhead * in_q;
class qtail * out_q;
: (0,0,0)      /* use default arguments for class task */
{
        class line * l;
        class word * w;

        for(;;) {
                l = in_q->get();
                while(w = next_word(l)) out_q->put(w);
        }
}

class qhead * line_q = new qhead(WMODE,10);
class qhead * word_q = new qtail(WMODE,50);

class producer * prod = new class producer(line_q->tail());
class consumer * cons = new class consumer(word_q->head());
class line_to_word * filt = new class line_to_word(line_q,word_q);
```

In this way the filter *filt* is programmed into the path between *cons* and *prod* using two queues to separate *filt's* input from its output.

This is a fairly static use of a filter. Often one would like to insert a filter into an existing data path. For example, a macro-based text formatting program could be organized as a sequence of filters - each doing its small part of the common task. First some filters re-arrange the input into a form suitable for the formatter proper, then the "input independent" formatter does its job producing output of a standard form, and last some output filters adjust this output to a form suitable for physical output. The task *filt* is an example of such a filter. In this scenario it would be useful to have each macro defined as a filter which the formatter proper inserts just in front of itself when the macro expansion is needed and which removes itself when it is not needed any more. Assuming that data streams are represented by queues, this can be achieved by using the class *qhead* functions *cut()* and *splice()*

When the task *formatter* recognizes a call to the macro "foo" it creates a new task of class *macro* to handle a macro of type FOO and diverts its own input through it. This is done by first "cutting" the input queue to create a place to insert the new filter, and then creating the filter giving it the new *qhead* and *qtail* as arguments:

```
class qhead * newhead = input_queue->cut();
class qtail * newtail = input_queue->tail();

class macro * f = new class macro(FOO,newhead,newtail);
```

Cut() splits the queue to which it is applied into two. *Newhead*, the pointer returned from *cut()*, denotes the *qhead* for the original queue and has the same *mode* as the original *qhead*. The original *qhead* is now attached to a new empty queue with the same *max* as the original.

Put()'s to the original *qtail* will therefore place objects on the filter's input queue, and *get()'s* from the original *qhead* will retrieve objects from the filter's output queue.

The result of these operations has been to insert a filter with an input and an output queue into a queue without changing the appearance of that queue to anyone using it without halting the flow of objects through that queue. In our example the macro expansion filter *foo* will *get()* the input which would otherwise have gone to the formatter, interpret it as macro arguments, and output the expanded input as its output.

The filter can be removed again by splicing its input and output queues together with *splice()*:

```
newhead->splice(newtail);
```

Splice() deletes the *qhead* to which it is applied, the *qtail* given to it as an argument, and the queue denoted by that *qtail*.

Deleting the filter completes the cleanup:

```
delete filt;
```

Typically a filter would remove itself when its task was completed, because the task that inserted it would not be programmed to be aware of the presence of the filter it inserted. The sequence of operations which enables a task to remove itself without a trace is:

```
cancel(any_value);
delete this;
```

This will work because *cancel()* does not imply immediate suspension, only a guarantee that the task cannot be resumed.

The *qtail* functions *cut()* and *splice()* are similar to *qhead's*, but they operate on the other end of the queue.

Monitors

Monitor is a base class that guarantees mutual exclusion for tasks executing functions from a class derived from it. That is, only one task at a time can be executing a function from a class derived from class *monitor*. A task calling a function from a class derived from class *monitor* while another task is executing a function for the same class object is suspended on that object waiting for the completion of the first task's call. At this time all waiting functions are re-started and will again try to enter the derived class*.

The public part of class monitor's declaration looks like this:

```
class monitor : public object { };
```

A *monitor* provides a facility for implementing a data structure shared between several tasks while maintaining a well-defined interface to it. Updates to data in a class derived from class *monitor* by functions from the derived class are atomic. For example:

```
class counter : monitor
{
        void            new(class qhead *);
        class qhead *   qq;
        short           nn;
public:
        void            put(class object *);
        short           read();
};

void counter.new(q) class qhead * q;
{
        qq = q;
        nn = 0;
};

void counter.put(x) class object x;
{
        nn++;
        qq->put(x);
};

short counter.read()
{
        return nn;
};
```

Where *nn* is intented to hold the number of of objects which have been *put()* into *qq*. Had *counter* not been a monitor a task could have *read()* while the counter was suspended waiting to *put()* into *qq* and an incorrect value would have been returned.

Monitors are less useful in a non-preemptive system than in a system with preemptive scheduling simply because any sequence of operations which does not suspend the current task is an atomic action. However, because a programmer is not necessarily aware of possible suspension operations buried in standard functions like *qtail.put()* monitors may still be useful. A monitor based system may also be chosen simply for stylistic reasons, or because the task system is used to simulate a monitor based system.

* There are, however, no guarantee that the monitor can be entered when a previously waiting task re-tries to enter. The task may be suspended again. A *monitor* does therefore not conform to Hoare's original definition of a monitor [2], but the total system of tasks should on average run more efficiently when this guarantee is not given [4].

Implementation Details

The following sections contain many implementation-dependent details. The implementation described is the PDP11/Unix version. Implementation-dependent information is unfortunately often necessary to allow tuning and ease debugging.

Task Stack Allocation

The two arguments *mode* and *stacksize* allows the user to guide the systems handling of the task. Their exact interpretation is implementation dependent. Users who are not interested in implementation details and/or want a more portable program should set them both to zero. The system will then choose (hopefully reasonable) implementation-dependent default values.

The *stacksize* argument indicates the maximum amount of stack storage that the task is allowed to use. Using more is an error. It will be expressed in a unit of store suitable for stack allocation on the host system. The stack is the one which is supported by the standard compiler and operating system.

The *mode* provides additional information. Two values are valid for the PDP11/Unix implementation.

The value SHARED indicates that the stack space should be taken from the stack space of the parent task, that is the task which created the new task. Where SHARED stacks are used the active part of the stack is copied to a save area when a task is suspended, and copied back when it is resumed. Since stack locations are not dedicated to a single task pointers to local variables should not be passed to other tasks. The time needed to suspend and resume a task with SHARED task is approximately proportional to the amount of stack space actually used at the time of suspension.

If, on the other hand *mode* is DEDICATED then a new and separate stack area is allocated, and no copying of stack space will occur.

Scheduling

Functions of a system class, known as the scheduler, are invoked as the result of any function of class *task* which causes the suspension of a running task, and may be invoked by any function from the standard classes described here. The scheduler selects the next task to run. When the scheduler finds no more tasks to run it examines the pointer variable *exit_fct*, and if this is non-zero the scheduler will call the function denoted by it.

Whenever *clock* is advanced the scheduler examines the pointer variable *clock_task*, and if it denotes a task then that task will be resumed before any other task. The *clock_task* must be IDLE when resumed by the scheduler. The class *task* function *sleep()* is useful to ensure this.

Debugging and Tuning Aids

The task system has been designed under the assumption that a typical use of tasks may involve hundreds of tasks and need tuning to achieve an acceptable time-space tradeoff. The task of debugging such a system can safely be assumed to be non-trivial.

Classes were used in the implementation of the task system largely because they allow the scope of data and functions to be explicitly restricted to the object to which they belong. This allows better type checking of a multi-threaded program than could be achieved by a function based implementation. The classes which constitute the task system were designed to allow quite strong type checking of programs using them.

A number of run time errors are detected by the task system, for example it is illegal to *delete* an object on which tasks are waiting. When such a run time error is detected the task system function *task_error* is called with the number of the error as argument. Appendix B is a list of run time errors. *Task_error()* will in turn examine the pointer *error_fct*, and if this is non-zero call the function denoted by it with the error number as argument. Otherwise *task_error()* will call the system function *exit()* with the error number as argument.

All task system classes have a *printout()* function which can be used to print the contents of their objects. The *printout()* functions all take one argument of type *short*. The higher the argument is the more verbose *printout()* becomes.

The current stack size is one of the pieces of information given by *task.printout()*. If the function *hwm()* has been executed then *printout()* will also give an estimate of the maximum amount of stack space ever used by the task, the stack's "high water mark". For tasks that share a stack the high water mark printed will be the high water mark of most greedy task.

The function *print_stacks()* will print out information about the stacks of all tasks in the task-system.

The function *print_tasks()* will call *printout()* for a specified set of tasks and timers. For example:

```
print_tasks(n,RUNNING|IDLE);
```

will call *task.printout(n)* for all RUNNING and IDLE tasks and *timer.printout(n)* for all RUNNING timers.

The output of these functions is implementation-dependent and hopefully self-explanatory.

The *extern short po_level* is used as the argument to *printout()* by *task_error()*.

Overheads and Performance

The store used for representing a class object in addition to the user specified data is:

```
object       6 bytes
timer       12 bytes
task        24 bytes + stacksize
queue       18 bytes (including the qhead and the qtail)
monitor      8 bytes
```

The time needed to execute some of the task system functions are:

```
C procedure call + return       1 unit
task suspend + resume           7 units (using result())
put                             2
get                             2
monitor call + return           3
wait, waitvec, or waitlist      3
```

The last four actions can all cause a task to be suspended. When this happens add 6 units of time.

The task system use about 8K bytes of store for program and data.

Acknowledgements

The task system is in many ways a descendent of A.G. Fraser's set of C functions described in reference 3. M. D. McIlroy acted as "midwife" for many parts of the design.

MH-1273-BS

Bjarne Stroustrup

Att.:
Appendix A
Appendix B
Appendix C

References

[1] Brinch-Hansen, Per
The Architecture of Concurrent Programs
Prentice Hall 1977

[2] Dahl, D.J., Myrhaug, B., and Nygaard, K.
SIMULA Common Base Language
Norvegian Computing Center, S-22 (Oct.1970)

[3] Fraser, A. G.
C Language Routines for Multi-Thread Computation
TM 79-1273-4, May 7, 1979

[4] Hoare, C. A. R.
Monitors: An Operating System Structuring Concept
Comm. ACM vol 17 no 10 (Oct.1974) pp 549-557

[5] Kernighan, B. W. and Richie, D. M.
The C Programming Language
Prentice-Hall 1978

[6] Lampson, Butler W. and Redell, David D.
Experience with Processes and Monitors in Mesa
Comm. ACM vol 23 no 2 (Feb.1980) pp 105-117

[7] Richards, Martin
The BCPL Programming Manual
The Computer Laboratory, University of Cambridge, England
(April 1973)

[8] Stroustrup, Bjarne
Classes: An Abstract Data Type Facility for the C Language
Bell Laboratories Computer Science Technical Report CSTR-81
(April 1980)

[9] Unix Programmer's Manual
Bell Telephone Laboratories (January 1979)

Appendix A: Objects

The task system as described above is implemented using a lower level of abstraction based on the direct use of the class *object*. Class *object* can also be used as a base for other (user defined) abstractions, but beware, it is an implementation tool, that is not intented to be used directly.

Class *object* is a base class for all classes in the task system and also the most basic facility for inter-task communication. The declaration of class *object* looks like this:

```
class object
{
        void            new(short);
        void            delete();
        class link *    o_link;
public:
        short           o_type;
        class object *  o_next;
        void            remember(class task *);
        void            forget(class task *);
        void            alert();

        void            printout(short);
};
```

The task system implements objects of type TASK, QHEAD, QTAIL, MONITOR, and TIMER.

A task can be added to the set of tasks "remembered" by an object by executing *remember()* and a task can be removed from this set by executing *forget()*. Executing *alert()* has the effect of transferring all IDLE tasks remembered by the object to the RUNNING state. A task can be "remembered" by several objects or several times by the same object without any bad effects. *Forget()* will insure that its argument is not "remembered" any more, and it causes no bad effects when used for an object that does not "remember" its argument task. No record is kept of how many *alert()* operations has been executed on an object. *Alert()* does not cause an object to *forget()* tasks. Executing a *remember()* does not suspend a task. Applying *alert()* to an object that does not remember any tasks is legal, but has no effect. Caveat Emptor!

The class *object* functions *remember(), forget(),* and *alert()* provide a simple, efficient, but unstructured and therefore error-prone, communication mechanism.

Appendix B: Run Time Errors

When an error is detected at run time *task_error()* is called. This function will print the error number on the standard error output stream *stderr*. It will then examine *error_fct*, and if this variable denotes a function that function will be called with the error number as argument. The value returned by the function denoted by *error_fct* will be given as argument to *exit()*. If no *error_fct* is supplied by the user *exit()* will be called with the error number as argument.

[11] Attempt to delete object which remembers a task.

[12] Attempt to delete an object which is still on some chain.

[21] The *clock_task* was non-IDLE when *clock* was advanced.

[23] An object of unknown type found by the scheduler.

[24] Attempt to schedule a TERMINATED task

[25] Negative time increment - go look for the fountain of youth.

[26] Attempt to schedule a task which is already on some other queue.

[31] Task mode corrupted at deletion.

[32] Unknown mode given to new task.

[37] Attempt to delete a non-TERMINATED task.

[39] Run time stack overflow.

[41] Attempt to delete a non-TERMINATED timer.

[52] *get()* from an empty queue with the qhead in EMODE.

[54] Attempt to *putback()* an object which is already on another queue.

[55] You cannot *putback()* to a full queue.

[62] Attempt to *put()* an object which is already on another queue.

[63] *put()* to a full queue with the qtail in EMODE.

[99] No more free store - NEW failed

Appendix C: A Program using Tasks

The program below is a simple example of a program using tasks and queues. A task of class *pc* uses *get()* to read objects from its input *qhead* and copies what it gets to its output by using *put()* on its *qtail.* Twenty *pc's* are created and joined in a circle. Then an object is *put()* to one of them. This object will now pass round the circle forever.

```
#include <class_task>

classdef pc;

class pc : task
{
        void        new(short,class qtail *,class qhead *);
};

void pc.new(n,t,h)
char * n;
class qtail * t;
class qhead * h;
: (n,0,0)
/*
        producer and consumer
*/
{
        class object * p;

        printf("\nnew class pc(%s,%o,%o)",n,t,h);

        while (TRUE) {
                p = h->get();
                printf("\ntask %s",n);
                t->put(p);
        };
} /* new */
```

```
#define NPC     20
#define TOKEN   11

main()
{
        class qhead * hh = new class qhead(WMODE,2);
        class qtail * t = hh->tail();
        class qhead * h;

        short i;

        printf("\nmain:");

        for (i=0;i<NPC;i++) {
                char * n = new char[2];
                n[0] = 'a' + i;
                n[1] = 0;
                h = new class qhead(WMODE,2);
                new class pc(n,t,h);
                printf("\nloop");
                t = h->tail();
        };
        new class pc(NPC,t,hh);

        printf("\nmain, here we go");

        t->put(new class object(TOKEN));

        printf("\nexit from main");
        thistask->sleep();
} /* main */
```