

DTIC FILE COPY

AD-A223 739



Carnegie-Mellon University
Software Engineering Institute

Support Materials for
**Language and System Support for
Concurrent Programming**

Support Materials SEI-SM-25

DTIC
ELECTE
JUL 11 1990
S & B D

DISTRIBUTION STATEMENT A



Carnegie Mellon University
Software Engineering Institute

Support Materials for Language and System Support for Concurrent Programming

.....
Gary Ford, editor
Software Engineering Institute

April 1990

Approved for public release.
Distribution unlimited.

This technical report was prepared for the

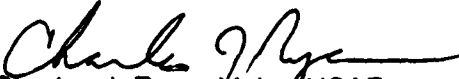
SEI Joint Program Office
ESD/AVS
Hanscom AFB, MA 01731

The ideas and findings in this report should not be construed as an official DoD position. It is published in the interest of scientific and technical information exchange.

Review and Approval

This report has been reviewed and is approved for publication.

FOR THE COMMANDER


Charles J. Ryan, Major, USAF
SEI Joint Program Office

This work is sponsored by the U.S. Department of Defense.

Copyright © 1990 by Carnegie Mellon University.

This document is available through the Defense Technical Information Center. DTIC provides access to and transfer of scientific and technical information for DoD personnel, DoD contractors and potential contractors, and other U.S. Government agency personnel and their contractors. To obtain a copy, please contact DTIC directly: Defense Technical Information Center, Attn: FDRA, Cameron Station, Alexandria, VA 22304-6145.

Copies of this document are also available through the National Technical Information Service. For information on ordering, please contact NTIS directly: National Technical Information Service, U.S. Department of Commerce, Springfield, VA 22161.

Use of any trademarks in this report is not intended in any way to infringe on the rights of the trademark holder.

Table of Contents

Examples of Concurrent Programs	1
<i>Michael B. Feldman</i>	
Dining Philosophers in Ada	3
Dining Philosophers in Concurrent C	5
Dining Philosophers in Co-Pascal	6
Dining Philosophers in occam	8
Sorting Algorithm Race in Ada	11
Sorting Algorithm Race in Concurrent C	16
Sorting Algorithm Race in Co-Pascal	21
Sorting Algorithm Race in Modula-2	25
Sorting Algorithm Race in occam	31
Modula-2 Library Modules for Concurrent Programming	34
Queue Abstract Data Type in Modula-2	37



Accession For	
NTIS GRA&I	☒
DTIC TAB	☐
Unannounced	☐
Justification	
By	
Distribution/	
Availability Codes	
Dist	Avail and/or Special
A-1	

Examples of Concurrent Programs

Michael B. Feldman

The George Washington University

The first example is an implementation in each of four languages (Ada, Concurrent C, Co-Pascal, and occam) of the famous Dining Philosophers problem first stated by Dijkstra¹. In this metaphorical statement of deadlock and resource allocation problems, five philosophers sit around a circular table, in the center of which is a infinitely large bowl of Chinese food. To the left and right of each philosopher is a single chopstick; each philosopher must try to acquire both chopsticks, eat for awhile, then put down the chopsticks and think for awhile; this cycle repeats for some total number of meals. (Dijkstra's original formulation used spaghetti and forks; we prefer the chopstick setting because most people can eat spaghetti with one fork.) The algorithm for chopstick selection must be chosen carefully, otherwise if all philosophers grab, say, their left chopsticks and refuse to yield them, all will starve!

The second example is one we have used with repeated success at The George Washington University, namely a "sort race" in which three different sorting methods are activated as processes. Each sort displays its progress in its "window" (usually a single row) on the terminal; mutual exclusion is necessary to protect the screen, which is a writable shared resource. We have found this example interesting and fun—there is a lot of screen activity, the problem being solved is obvious, and the three independent sorts serve as placeholders for any three independent applications contending for the processor and a shared data structure. In our comparative concurrency seminar, students must implement the sort race in the five different languages, starting from modules like sort subroutines, terminal drivers, process managers, etc., supplied by the teacher.

Machine-readable copies of these programs are available from the Software Engineering Institute. You may request a copy in either of the ways described below. Be sure to specify that you want the "Examples of Concurrent Programs" from support materials package SEI-SM-25.

1. **Electronic Mail.** Send your request to education@sei.cmu.edu on the Internet. The programs will be sent by electronic mail within a few days.
2. **Diskette.** A diskette containing the programs may be ordered from the SEI Software Engineering Curriculum Project. The cost is \$10 and a check must

¹Dijkstra, E. W. "Hierarchical Ordering of Sequential Processes." *Acta Informatica* 1, 115-138.

accompany your order. Two formats are available: IBM PC/AT diskette (5.25", double-sided, high-density, 1.2M byte) and Macintosh diskette (3.5", double-sided, 800K byte). Please specify the desired format.

Dining Philosophers in Ada

-- Dining Philosophers in Ada
-- Michael B. Feldman, The George Washington University
-- January 1990

```
with TEXT_IO, CALENDAR;
use CALENDAR;
procedure EAT is
  package INT_IO is new TEXT_IO.INTEGER_IO(INTEGER);
  task type CHOPSTICK is
    entry PICKUP;
    entry PUTDOWN;
  end CHOPSTICK;
  task SCREEN is
    entry PUT_LINE(S: STRING);
  end SCREEN;
  subtype NAME is STRING(1..3);
  task type PHILOSOPHER is
    entry GIVE_BIRTH ( ID: NAME; who, one, two : integer );
  end PHILOSOPHER;

  CHOPSTICKS : array (1..5) of CHOPSTICK;
  PHILOSOPHERS : array (1..5) of PHILOSOPHER;
  NAMES : constant array(1..5) of NAME :=
    ("Tony Hoare ",
     "Nicky Wirth ",
     "Eddy Dijkstra",
     "Jean Ichbiah ",
     "Narain Gehani");
  NO_MEALS : integer;
  START_TIME: duration;
  task body SCREEN is
    begin
      loop
        select
          accept PUT_LINE(S: STRING) do
            TEXT_IO.PUT_LINE(S);
          end PUT_LINE;
        or
          terminate;
        end select;
      end loop;
    end SCREEN;

  task body CHOPSTICK is
    begin
      loop
        select
          accept PICKUP;
        or
          terminate;
        end select;

        accept PUTDOWN;
      end loop;
    end CHOPSTICK;

  task body PHILOSOPHER is
    MY_NAME : NAME;
```

```

    first,second,identity : integer;
begin
    select
        accept GIVE_BIRTH ( ID: NAME; who, one, two : integer ) do
            MY_NAME := ID;
            identity := who;
            first := one;
            second := two;
            SCREEN.put_line("T = "
                & integer'image(integer(seconds(clock)-START_TIME))
                & " " & MY_NAME & " living and breathing");
        end GIVE_BIRTH;
    or
        terminate;
    end select;
    for x in 1..NO_MEALS loop
        CHOPSTICKS(first).PICKUP;
        CHOPSTICKS(second).PICKUP;
        SCREEN.put_line("T = "
            & integer'image(integer(seconds(clock)-START_TIME))
            & " " & MY_NAME & " eating with chopsticks"
            & integer'image(first) & " "&integer'image(second) );
        delay DURATION(2*identity);
        SCREEN.put_line("T = "
            & integer'image(integer(seconds(clock)-START_TIME))
            & " " & MY_NAME & " done");
        CHOPSTICKS(first).PUTDOWN;
        CHOPSTICKS(second).PUTDOWN;
    end loop;
    SCREEN.put_line(MY_NAME & " burp");
end PHILOSOPHER;

```

```

begin
    SCREEN.put_line("How many meals do you want to eat?");
    INT_IO.get(NO_MEALS);
    TEXT_IO.NEW_LINE;
    START_TIME := seconds(clock);
    PHILOSOPHERS(2).GIVE_BIRTH(NAMES(2),2,2,3);
    PHILOSOPHERS(5).GIVE_BIRTH(NAMES(5),5,1,5);
    PHILOSOPHERS(3).GIVE_BIRTH(NAMES(3),3,3,4);
    PHILOSOPHERS(4).GIVE_BIRTH(NAMES(4),4,4,5);
    PHILOSOPHERS(1).GIVE_BIRTH(NAMES(1),1,1,2);
end EAT;

```


Dining Philosophers in Concurrent C

```
/* Non-deadlocking Dining Philosophers in Concurrent C
/* Adapted from
   Gehani and Roome, "The Concurrent C Programming Language" by
   Prof. Michael Feldman
   The George Washington University
   February 1990
*/

process spec fork()
{
    trans void pickUp(), putDown();
};
process body fork()
{
    for (;;) {
        accept pickUp();
        accept putDown();
    }
}

process spec philosopher(int id,
                          process fork left,
                          process fork right);
#define LIMIT 10
process body philosopher(id, left, right)
{
    int nmeal;
    printf("Phil. %d: *alive*\n", id);
    for (nmeal = 0; nmeal < LIMIT; nmeal++) {
        /*think; then enter dining room */
        delay 2*(5-id);
        /*pick up forks*/
        right.pickUp();
        left.pickUp();
        /*eat*/
        printf("Phil. %d: *eating*\n", id);
        delay 2*(5-id);
        printf("Phil. %d: *burp*\n", id);
        /*put down forks*/
        left.putDown();
        right.putDown();
        /*get up and leave dining room*/
    }
    printf("Phil. %d: That's all, folks!\n", id);
}

main()
{
    process fork f[5]; int j;

    /*create forks, then create philosophers*/
    for (j = 0; j < 5; j++)
        f[j] = create fork();
    for (j = 0; j < 5; j++)
        create philosopher(j, f[j], f[(j+1) % 5]);
    create philosopher(4, f[0], f[4]);
}
```

Dining Philosophers in Co-Pascal

```
program diners (input, output);

{ This is the Dining Philosophers written in Co-Pascal      }
{ Prof. Michael B. Feldman, The George Washington University }
{ January 1990                                              }

const life = 5;
type semaphore = integer;
var chopsticks: array[0..3] of semaphore;
    room: semaphore;
    screen: semaphore;
    which: integer;

procedure delay(HowLong: integer);
var count: integer;
begin
    count := 1;
    while count < HowLong do
        count := count+1;
    end {delay};

procedure think(WhoAmI: integer);
begin
    wait(screen);
    writeln('Philosopher ',WhoAmI:2,' ..Hmmm...');
    signal(screen);
    delay(10*(WhoAmI+1));

end {think};

procedure eat(WhoAmI: integer; meals:integer);
begin
    wait(screen);
    writeln('Philosopher ',WhoAmI:2,' eating meal ', meals:3, ' ..Slurp slurp...');
    signal(screen);
    delay(100*(WhoAmI+1));

end {eat};

procedure philosopher(WhoAmI: integer);
var meals: integer;
begin
    wait(screen);
    writeln('philosopher ',WhoAmI:2, ' breathing');
    signal(screen);

    for meals := 1 to life do
        begin
            think(WhoAmI);
            wait(room);
            wait(chopsticks[WhoAmI]);
            wait(chopsticks[(WhoAmI+1) mod 4]);
            eat(WhoAmI,meals);
            signal(chopsticks[WhoAmI]);
            signal(chopsticks[(WhoAmI+1) mod 4]);
            signal(room);
        end;
    end;
```

```

wait(screen);
writeln('philosopher ',WhoAmI:2, ' burp');
signal(screen);

end {philosopher};

begin {main}
  room := 3;
  screen := 1;
  for which := 0 to 3 do
    chopsticks[which] := 1;
  cobegin
    philosopher(0);
    philosopher(1);
    philosopher(2);
    philosopher(3);
  coend;
end {diners}.

```

Dining Philosophers in occam

```
--
-- Implementation in occam of the dining philosophers problem.
-- Distributed with University of Loughborough occam for UNIX systems.
-- execute with -c option to get cursor control
--
-- A number of philosophers spend their life either thinking or eating.
-- Unfortunately there is only one bowl of spaghetti and there is only one fork
-- per philosopher, but two forks are needed to eat the food.
-- A philosopher waits for a neighbour to relinquish a fork if needed.
-- The system can deadlock (the philosophers can starve) but it is difficult
-- to prove it.
-- The system is simulated by making the philosophers eat and think for random
-- times, a cursor addressible screen is used for output showing the current
-- status.
--
DEF Enter = 0, Exit = 1 :
DEF Grab = 0, Replace = 1, To.Right = 2, To.Left = 3 :
DEF Grabbed = 0, PutBack = 1 :
DEF Thought = 0, Consume = 1, Queuing = 2 :
--
-- Number of philosophers - may be between 1 and 8
--
DEF number.of.philosophers = 5:
CHAN Door [number.of.philosophers], Request.Fork [number.of.philosophers*2] :
CHAN phil.info [number.of.philosophers], Fork.info [number.of.philosophers] :
CHAN room.info :
EXTERNAL PROC random (VALUE m, VAR n) :
--
-- Sit and think outside the room for a random time interval
--
PROC Think (VALUE n) =
  VAR think.time :
  SEQ
    -- Thinking
    phil.info [n] ! Thought
    random (90, think.time)
    WAIT 40 + think.time
    -- Finished thinking - now waiting to eat.
    phil.info [n] ! Queuing :
  --
  -- Have grabbed two forks - signal eating and wait for a random interval
  --
PROC Eat (VALUE n) =
  VAR eat.time :
  SEQ
    phil.info [n] ! Consume
    random (80, eat.time)
    WAIT 50 + eat.time :
  --
  -- Define action of philosopher - think, enter room, pick up left then
  -- pick up right fork and eat, finally leave the room to think again.
  --
PROC Philosopher (VALUE n, CHAN left, right) =
  WHILE TRUE
    SEQ
      Think (n)
      Door [n] ! Enter
      left ! Grabbed
      right ! Grabbed
```

```

        Eat (n)
        left ! PutBack
        right ! PutBack
        Door [n] ! Exit :
--
-- Room - keep account of how many philosophers
-- there are eating or waiting to eat.
--
PROC Room =
    VAR action,number.in :
    SEQ
        number.in := 0
        WHILE TRUE
            SEQ
                room.info ! number.in
                ALT m = [0 FOR number.of.philosophers]
                    Door [m] ? Action
                    IF
                        Action = Enter
                            number.in := number.in + 1
                        TRUE
                            number.in := number.in - 1 :
--
-- Control of each fork - can be picked up by either side but then must
-- wait until it is put down.
-- Tell the display process the new status of the fork.
--
PROC Fork (VALUE n,CHAN left,right) =
    WHILE TRUE
        ALT
            left ? ANY
                SEQ
                    Fork.Info [n] ! To.Left ; Grab
                    left ? ANY
                    Fork.Info [n] ! To.Left ; Replace
            right ? ANY
                SEQ
                    Fork.Info [n] ! To.Right ; Grab
                    right ? ANY
                    Fork.Info [n] ! To.Right ; Replace :
--
-- Show animated display of what is happening
--
EXTERNAL PROC str.to.screen (VALUE s []) :
EXTERNAL PROC num.to.screen.f (VALUE n,f) :
EXTERNAL PROC Goto.x.y (VALUE x,y) :
EXTERNAL PROC clear.screen :
PROC Display =
    VAR Action,Which,Person,How.Many.In :
    SEQ
        clear.screen
        Goto.x.y (0,2)
        str.to.screen ("Number of philosophers in room : ")
        SEQ n = [0 FOR number.of.philosophers]
            SEQ
                Goto.x.y (0,(n*3)+4)
                str.to.screen ("Philosopher ")
                num.to.screen.f (n,3)
        WHILE TRUE
            ALT
                room.info ? How.Many.In
                SEQ
                    Goto.x.y (33,2)
                    num.to.screen.f (How.Many.In,2)

```

```

ALT m = [0 FOR number.of.philosophers]
  ALT
    phil.info [m] ? Action
    IF
      Action = Thought
      SEQ
        Goto.x.y (20, (m*3)+4)
        str.to.screen ("Thinking ")
      Action = Queuing
      SEQ
        Goto.x.y (20, (m*3)+4)
        str.to.screen ("Waiting ")
      TRUE
      SEQ
        Goto.x.y (20, (m*3)+4)
        str.to.screen ("Eating ")
    Fork.Info [m] ? Which
    SEQ
      IF
        Which = To.Left
        SEQ
          Person := m
          Goto.x.y (50, (Person*3)+4)
        TRUE
        SEQ
          Person := (m+1)\number.of.philosophers
          Goto.x.y (55, (Person*3)+4)
    Fork.Info [m] ? Action
    IF
      Action = Grab
      str.to.screen ("!")
      Action = Replace
      str.to.screen (" ") :
--
-- Define parallel processes
-- There are two channels from philosophers to each fork.
-- The fork process ensures it is in the hand of one philosopher only.
--
PAR
  Room
  Display
  PAR n = [0 FOR number.of.philosophers]
    PAR
      Philosopher (n, Request.Fork [n*2], Request.Fork [(n*2)+1])
      Fork (n, Request.Fork [(n*2)+1], Request.Fork
[ ((n*2)+2)\(number.of.philosophers*2)])

```

Sorting Algorithm Race in Ada

```
WITH TEXT_IO; USE TEXT_IO;
WITH VT100; USE VT100; -- this package is shown after the main program
```

```
PROCEDURE SortRace IS
```

```
--                               SortRace in Ada
--
--                               F. C. Hathorn
--                               CS - 358
--                               5/6/87
```

```
PACKAGE Int_IO IS NEW Integer_IO(Integer);
```

```
MaxLimit: CONSTANT := 34;
Line1: CONSTANT    := 8;
Line2: CONSTANT    := 12;
Line3: CONSTANT    := 16;
```

```
SUBTYPE ValueType IS CHARACTER;
TYPE Vector      IS ARRAY (0..MaxLimit) OF ValueType;
```

```
V:      Vector;
Limit:  Integer;
```

```
TASK Bubble_Sort is
  ENTRY GoAhead;
END Bubble_Sort;
```

```
TASK Insert_Sort is
  ENTRY GoAhead;
END Insert_Sort;
```

```
TASK Heap_Sort is
  ENTRY GoAhead;
END Heap_Sort;
```

```
TASK Screen is
  Entry ClearScreen;
  Entry PutAt(column, row: INTEGER; c: ValueType);
END Screen;
```

```
-----
-- Put Vector
-- This procedure displays a vector on the screen at a given row
--
-----
```

```
PROCEDURE PutVect(S: Vector; Row: INTEGER) IS
BEGIN
  FOR i IN 1..Limit LOOP
    Screen.PutAt(i+1, Row, S(i));
  END LOOP;
END PutVect;
```

```
-----
-- Swap
-- This procedure exchanges two integer variable values.
--
-----
```

```

--
--
-----
PROCEDURE Swap(x,y: IN OUT ValueType; i,j, row: INTEGER) IS
  Temp: ValueType;
  BEGIN
    Temp := x;
    x := y;
    y := Temp;
    Screen.PutAt(i+1,row,x);
    Screen.PutAt(j+1,row,y);
  END Swap;

-----

-- Task Screen --
-- Code to write to the screen. Two entries are provided, ClearScreen --
-- which clears the screen and PutAt which writes one character. --
--
-----

TASK BODY Screen IS

  BEGIN
    LOOP
      SELECT
        ACCEPT ClearScreen DO
          VT100.ClearScreen;
        END ClearScreen;
      OR
        ACCEPT PutAt(column, row: INTEGER; c: ValueType) DO
          VT100.SetCursorAt(column,row); put(c);
        END PutAt;
      OR
        TERMINATE;
      END SELECT;
    END LOOP;
  END Screen;

-----

-- Task Bubble Sort --
-- Code provided by Professor M.B. Feldman and modified slightly to sort --
-- from 1..Limit rather than 0..Limit. --
--
-----

TASK BODY Bubble_Sort IS

  MyV: Vector;
  MyRow: Integer := Linel;
  CurrentBottom: INTEGER;
  AnotherPassNeeded: BOOLEAN;
  Top: INTEGER;

  BEGIN --Bubble_Sort
    Accept GoAhead;
    PutVect(V,MyRow);
    MyV := V;
    Top := 1;
    CurrentBottom := Limit;
    AnotherPassNeeded := TRUE;
    WHILE AnotherPassNeeded AND (CurrentBottom > 1) LOOP
      AnotherPassNeeded := FALSE;
      FOR Current IN Top .. CurrentBottom-1 LOOP
        IF (MyV(Current+1) < MyV(Current)) THEN

```



```

        Swap(MyV(Current+1), MyV(Current), Current+1, Current, MyRow);
        AnotherPassNeeded := TRUE;
    END IF;
    if (current+1 = currentbottom) THEN
        Screen.PutAt(CurrentBottom+1, MyRow+1, '<');
    END IF;
    END LOOP;
    CurrentBottom := CurrentBottom - 1;
    END LOOP;
    Screen.PutAt(CurrentBottom+1, MyRow+1, '*');
END Bubble_Sort;

```

```

-----
-- Task Insertion Sort                                     --
-- This task performs an insertion sort on the input array. --
--
-----

```

TASK BODY Insert_Sort IS

```

MyV: Vector;
MyRow: Integer := Line2;
j: integer;          --pointer into sorted array
insert: valuetype;   --current key being inserted

begin --Insert_Sort
    Accept GoAhead;
    PutVect(V, MyRow);
    MyV := V;
    MyV(Limit+1) := 'z';          --initialize last + 1th element
    Screen.PutAt(Limit+1, MyRow+1, '<'); --mark last element as sorted
    FOR i IN REVERSE 1..Limit-1 LOOP --insert elements limit-1..1 into
        insert := MyV(i);          --save current key
        j := i + 1;
        WHILE (insert > MyV(j)) LOOP --shift larger keys up
            MyV(j-1) := MyV(j);
            Screen.PutAt(j, MyRow, MyV(j));
            j := j + 1;
        END LOOP;
        MyV(j-1) := insert;          --insert current key in proper place
        Screen.PutAt(j, MyRow, insert);
        Screen.PutAt(i+1, MyRow+1, '<');
    END LOOP;
    Screen.PutAt(2, MyRow+1, '*');
end Insert_Sort;

```

```

-----
-- Task Heap Sort                                         --
-- This task sorts the input key array using the heap sort algorithm. --
-- The input array is treated as a binary tree when building the heap. --
--
-----

```

TASK BODY Heap_Sort IS

```

MyV: Vector;
MyRow: Integer := Line3;

Procedure Adjust(t: IN OUT Vector; root, Lmt: integer) IS
-- adjust is used to adjust a heap whose left and right trees are heaps, but
-- whose root may be smaller than its left or right child

    j: integer;          --child pointer

```

```

key:      ValueType; --key element
done:     boolean := FALSE; --adjustments done flag
BEGIN
  key := t(root);          --save root key
  j := 2 * root;           --calculate child pointer
  WHILE ((j <= Lmt) and not done) LOOP
    IF (j < Lmt) THEN      --find largest child
      if (t(j) < t(j+1)) THEN j := j + 1; END IF;
    END IF;
    IF (key >= t(j)) THEN
      done := TRUE;        --done if child smaller than root
    ELSE
      --otherwise move child up
      t(j / 2) := t(j);
      Screen.PutAt(j / 2 + 1, MyRow, t(j));
      j := 2 * j;
    END IF;
  END LOOP;
  t(j / 2) := key;         --insert root in correct position
  Screen.PutAt(j / 2 + 1, MyRow, key);
END Adjust;

BEGIN
-- main section of code for heap sort
  Accept GoAhead;
  PutVect(V, MyRow);
  MyV := V;
--convert the input array into a heap
  FOR i IN REVERSE 1..(Limit / 2) LOOP
    adjust(MyV, i, Limit);
  END LOOP;
  FOR i IN REVERSE 1..(Limit-1) LOOP --pick off first element n-1 times
    swap(MyV(1), MyV(i+1), 1, i+1, MyRow); --swap with last element
    Screen.PutAt(i+2, MyRow+1, '<');
    adjust(MyV, 1, i); --readjust heap less last element
  END LOOP;
  Screen.PutAt(2, MyRow+1, '*');
END Heap_sort;

BEGIN
  V := " ZzYyXxWwVvUuTtSsRrQqPpOoNnMmLlKkJj";
  V(0) := '<';
  V(34) := '<';
  Screen.ClearScreen;
  Screen.PutAt(1, Line1-3, ' ');
  Put_Line("SORT RACE - in Ada");
  Put("Enter Number of Keys to Sort (3-33): ");
  Int_IO.Get(Limit);
  IF (Limit < 3) OR (Limit > 33) THEN
    Limit := 10;
    Put(ASCII.BEL);
    Put_Line("Sorting 10 keys");
  END IF;
  Screen.PutAt(1, Line1-1, ' ');
  Put_Line("Bubble Sort");
  Screen.PutAt(1, Line2-1, ' ');
  Put_Line("Reverse Insertion Sort");
  Screen.PutAt(1, Line3-1, ' ');
  Put_Line("Heap Sort");
  Screen.PutAt(1, 20, ' ');
  Bubble_Sort.GoAhead;
  Insert_Sort.GoAhead;
  Heap_Sort.GoAhead;
END SortRace;

```

```

with TEXT_IO, MY_INT_IO; use TEXT_IO, MY_INT_IO;
package VT100 is
    use ASCII;
-----
-- Procedures for drawing pictures of the solution on VDU.
-- ClearScreen and SetCursorAt are device-specific
-----

    SCREEN_DEPTH      : constant INTEGER := 24;
    SCREEN_WIDTH      : constant INTEGER := 80;

    subtype DEPTH is INTEGER range 1..SCREEN_DEPTH;
    subtype WIDTH is INTEGER range 1..SCREEN_WIDTH;

    procedure ClearScreen;

    procedure SetCursorAt( A: WIDTH; D : DEPTH);

end VT100;

-- .....
with TEXT_IO; use TEXT_IO;
package body VT100 is
    use ASCII;
-----
-- Procedures for drawing pictures on VT100
-- ClearScreen and SetCursorAt are terminal-specific
-----

    procedure ClearScreen is
    begin
        PUT( ESC & "[2J" );
    end ClearScreen;

    procedure SetCursorAt(A: WIDTH; D : DEPTH) is
    begin
        PUT( ESC & "[" );
        PUT( D, 1 );
        PUT( ';' );
        PUT( A, 1 );
        PUT( 'f' );
    end SetCursorAt;

end VT100;

-- .....

```

Sorting Algorithm Race in Concurrent C

```

/*
--
--          SortRace in Concurrent C
--
--          F. C. Hathorn
--          CS - 358
--          5/5/87
*/

#define MaxLimit 36
#define Line1 6
#define Line2 12
#define Line3 18
#define SMILE '<'
#define STAR '*'
#define BELL '\\7'
#define VALUETYPE char
#define TRUE 1
#define FALSE 0

VALUETYPE V[MaxLimit] = " ZzYyXxWwVvUuTtSsRrQqPpOoNnMmLlKkJj";
int Counter = 0;
int Limit;

process spec Bubble_Sort( VALUETYPE MyV[36], int MyRow, process Scrn );
process spec Insert_Sort( VALUETYPE MyV[36], int MyRow, process Scrn );
process spec Heap_Sort ( VALUETYPE MyV[36], int MyRow, process Scrn );
process spec Scrn ()
{
    trans void PutAt(int, int, VALUETYPE);
    trans void CheckWinner(int);
};

/*-----
-- Bubble Sort
-- Code Provided by Professor M.B. Feldman and modified slightly to sort
-- from 1..Limit rather than 0..Limit.
--
--*/
process body Bubble_Sort(MyV, MyRow, Screen)
{
    int CurrentBottom;
    int AnotherPassNeeded;
    int Current, Top;

    PutVect(MyV, MyRow, Screen);
    Top = 1;
    CurrentBottom = Limit;
    AnotherPassNeeded = TRUE;
    while ((AnotherPassNeeded) && (CurrentBottom > 1)) {
        AnotherPassNeeded = FALSE;
        for (Current = Top; Current < CurrentBottom; Current++) {
            if (MyV[Current+1] < MyV[Current]) {
                Swap(&MyV[Current+1], &MyV[Current], Current+1, Current, MyRow,
                    Screen);
                AnotherPassNeeded = TRUE;
            }
        }
        if (Current+1 == CurrentBottom)
            Screen.PutAt(CurrentBottom+1, MyRow+1, SMILE);
    }
}

```

```

        CurrentBottom = CurrentBottom - 1;
    }
    Screen.PutAt (CurrentBottom+1, MyRow+1, STAR);
    Screen.CheckWinner(MyRow + 1);
} /* Bubble_Sort */

/*-----
-- Insertion Sort
-- This process performs an insertion sort on the input array.
--
-- */
process body Insert_Sort(MyV, MyRow, Screen)
{
    int j;          /* pointer into sorted array */
    int i;
    VALUETYPE insert; /* current key being inserted */

    PutVect(MyV, MyRow, Screen);
    MyV[Limit+1] = '\177'; /*initialize last + 1 element */
    Screen.PutAt(Limit+1, MyRow+1, SMILE); /*mark last element as sorted */
    for (i=Limit-1; i>=1; i--) { /*insert elements from limit-1..1 */
        insert = MyV[i]; /*save current key */
        j = i + 1;
        while (insert > MyV[j]) { /*shift larger keys up */
            MyV[j+1] = MyV[j];
            Screen.PutAt(j, MyRow, MyV[j]);
            j = j + 1;
        }
        MyV[j+1] = insert; /*ins current key in proper loc */
        Screen.PutAt(j, MyRow, insert);
        Screen.PutAt(i+1, MyRow+1, SMILE);
    }
    Screen.PutAt(2, MyRow+1, STAR);
    Screen.CheckWinner(MyRow + 1);
} /* Insert_Sort */

/*-----
-- Heap Sort
-- This process sorts the input key array using the heap sort algorithm.
-- The input array is treated as a binary tree when building the heap.
--
-- */
process body Heap_Sort(MyV, MyRow, Screen)
{
    int i;

    PutVect(MyV, MyRow, Screen);
    /* convert the input array into a heap */
    for (i=(Limit / 2); i>=1; i--)
        Adjust(MyV, i, Limit, MyRow, Screen);
    /* pick off first element n-1 times */
    for (i=(Limit-1); i>=1; i--) {
        Swap(&MyV[1], &MyV[i+1], 1, i+1, MyRow,
            Screen); /* swap w/ last element */
        Screen.PutAt(i+2, MyRow+1, SMILE);
        Adjust(MyV, 1, i, MyRow, Screen); /* readjust heap */
    }
    Screen.PutAt(2, MyRow+1, STAR);
    Screen.CheckWinner(MyRow + 1);
} /* Heap_sort */

/*-----
-- Process Screen
-- This process controls access to the screen for writing once the sort
--
-- */

```

```

-- processes have been activated
-----*/
process body Scrn()
{
    for (;;)          /* loop forever */
    select
    {
        accept PutAt(column, row, c)
        {
            SetCursorAt(column, row);
            putchar(c);
        } /* PutAt */
    or
        accept CheckWinner(row)
        {
            int i;
            Counter = Counter + 1;
            SetCursorAt(Limit+4, row);
            switch (Counter) {
                case 1: printf("WINNER!!!");
                        break;
                case 2: printf("SECOND!!");
                        break;
                case 3: printf("THIRD!");
                        SetCursorAt(1, Line3+4);
                        break;
            }
            for (i=Counter; i < 4; i++) putchar(BELL);
        } /* CheckWinner */
    or
        terminate;
    }
} /* Scrn */

main()
{
    VALUETYPE v1[MaxLimit], v2[MaxLimit], v3[MaxLimit];
    int i;

    process Scrn monitor;      /* screen monitor */
    process Bubble_Sort s1;
    process Insert_Sort s2;
    process Heap_Sort s3;

    ClearScreen(), SetCursorAt();

    V[0] = '\0';
    for (i=0; i<MaxLimit; i++)
        {v1[i] = V[i]; v2[i] = V[i]; v3[i] = V[i]; }
    SetCursorAt(1,1);
    ClearScreen();
    printf("SORT RACE - in Concurrent C\n");
    printf("Enter Number of Keys to Sort (3-33): ");
    scanf("%d%c", &Limit);
    if ((Limit < 3) || (Limit > 33)) {
        Limit = 10;
        putchar(BELL);
        printf("Sorting only 10 Keys\n");
    }
    SetCursorAt(2, Line1-2);
    printf("Bubble Sort");
    SetCursorAt(2, Line2-2);
}

```

```

printf("Reverse Insertion Sort");
SetCursorAt(2, Line3-2,);
printf("Heap Sort");

/* start the screen monitor first */
monitor = create Scrn();

/* start the 3 sort processes */

s1 = create Bubble_Sort(v1, Line1, monitor);
s2 = create Insert_Sort(v2, Line2, monitor);
s3 = create Heap_Sort(v3, Line3, monitor);
} /* main */

ClearScreen()
{
    putchar('\033'); putchar('[');
    putchar('2'); putchar('J');
} /* clearsreen */

SetCursorAt(column, row)
int column, row;
{
    static ASCIIOffset = 48;
    putchar('\033'); putchar('[');
    putchar((row / 10) + ASCIIOffset);
    putchar((row % 10) + ASCIIOffset);
    putchar(';');
    putchar((column / 10) + ASCIIOffset);
    putchar((column % 10) + ASCIIOffset);
    putchar('H');
} /* SetCursorAt */

/*-----
-- Put Vector                                     --
-- This procedure copies the input vector into a local vector of the --
-- calling task and displays that vector on the screen                --
-------*/
PutVect(InV, row, Screen)
VALUETYPE InV[];
int row;
process Scrn Screen;
{
    int i;
    for (i = 1; i <= Limit; i++) Screen.PutAt(i+1, row, InV[i]);
} /* PutVect */

/*-----
-- Swap                                           --
-- This procedure exchanges two integer variable values.             --
-------*/
Swap(x, y, i, j, row, Screen)
VALUETYPE *x, *y;
int i, j, row;
process Scrn Screen;
{
    VALUETYPE temp;
    temp = *x;
    *x = *y;

```

```

*y = temp;
Screen.PutAt(i+1,row,*x);
Screen.PutAt(j+1,row,*y);
} /* Swap */

```

```

/*-----
-- Adjust
-- adjust is used to adjust a heap whose left and right trees are heaps,
-- but whose root may be smaller than its left or right child
--
-----*/

```

```

Adjust(t, root, Lmt, MyRow, Screen)
VALUETYPE t[];
int root, Lmt, MyRow;
process Scrn Screen;
{
    int j;                /* child pointer */
    VALUETYPE key;        /* key element */
    int done = FALSE;     /* adjustments done flag */

    key = t[root];        /* save root key */
    j = 2 * root;         /* calculate child pointer */
    while ((j <= Lmt) && !done) {
        if (j < Lmt) {    /* find largest child */
            if (t[j] < t[j+1]) j = j + 1; }
        if (key >= t[j])
            done = TRUE;  /* done if child smaller than root */
        else {            /* otherwise move child up */
            t[j / 2] = t[j];
            Screen.PutAt(j / 2 + 1, MyRow, t[j]);
            j = 2 * j;
        }
    }
    t[j / 2] = key;       /* insert root in correct position */
    Screen.PutAt(j / 2 + 1, MyRow, key);
} /* Adjust */

```


Sorting Algorithm Race in Co-Pascal

```
PROGRAM SortRace(INPUT,OUTPUT);
```

```
{ Sort Race - written by Roshan Thomas  
                  The George Washington University  
                  CSci 358 - Spring 1989
```

```
Tested under Co-Pascal version 3.0 for IBM-PC.  
Be sure ANSI.SYS is installed before compiling this.
```

```
demonstrates a concurrent sort race using Bubble Sort, Linear Insertion,  
and a non-recursive version of QuickSort }
```

```
CONST Limit = 32;
```

```
TYPE ValueType = CHAR;  
    semaphore = INTEGER;  
    Vector = ARRAY[0..Limit] OF ValueType;
```

```
VAR V: Vector;  
    i, Won: INTEGER;  
    A: CHAR;  
    Screen: semaphore;
```

```
PROCEDURE ClearScreen;  
BEGIN  
    Write(CHR(27)); Write('[');  
    Write('2'); Write('J')  
END {ClearScreen};
```

```
PROCEDURE SetCursorAt(column, row: INTEGER);  
BEGIN  
    WriteLn;  
    Write(CHR(27)); Write('[');  
    Write(row:1);  
    Write(';');  
    Write(column:1);  
    Write('H');  
END {SetCursorAt};
```

```
PROCEDURE WriteAt(column, row: INTEGER; C: CHAR);  
BEGIN  
    WAIT(Screen);  
    SetCursorAt(column, row);  
    Write(C);  
    SIGNAL(Screen);  
END {WriteAt};
```

```
PROCEDURE WriteVect(V: Vector; Row: INTEGER);  
VAR i: INTEGER;  
BEGIN  
    FOR i := 0 TO Limit DO BEGIN  
        WriteAt(i+1, Row, V[i]);  
    END;  
    WriteLn;  
END {WriteVect};
```

```
PROCEDURE CopyVect (VAR Dest: Vector; Source: Vector);
```

```

    VAR i: INTEGER;
BEGIN
    FOR i := 0 TO Limit DO BEGIN
        Dest[i] := Source[i];
    END;
END {CopyVect};

PROCEDURE Swap(VAR x,y: ValueType; i,j, row: INTEGER);
    VAR Temp: ValueType;
BEGIN
    Temp := x;
    x := y;
    y := Temp;
    WriteAt(i+1,row,x);
    WriteAt(j+1,row,y);
END {Swap};

PROCEDURE Bubble(MyV: Vector; MyRow: INTEGER);

    VAR
        CurrentBottom: INTEGER;
        AnotherPassNeeded: BOOLEAN;
        Top: INTEGER;
        Current: INTEGER;

BEGIN
    Top := 0;
    CurrentBottom := Limit;
    AnotherPassNeeded := TRUE;
    WriteVect(MyV,MyRow);
    WHILE AnotherPassNeeded AND (CurrentBottom > 0) DO BEGIN
        AnotherPassNeeded := FALSE;
        FOR Current := Top TO CurrentBottom-1 DO BEGIN
            IF MyV[Current+1] < MyV[Current] THEN BEGIN
                Swap(MyV[Current+1],MyV[Current],Current+1,Current,MyRow);
                AnotherPassNeeded := TRUE;
            END;
        END;
        CurrentBottom := CurrentBottom - 1;
    END;
    IF Won = 0 THEN
        BEGIN
            WAIT(Screen);
            Won := 1;
            SetCursorAt(8,6);
            WRITELN('BUBBLE SORT HAS WON, SURPRISINGLY');
            SIGNAL(Screen);
        END;
    END {Bubble};

PROCEDURE LinearInsertionSort(LV: Vector; Lrow: INTEGER);

    VAR
        NewArrival: ValueType;
        Top: INTEGER;
        Bottom: INTEGER;
        CurrentBottom: INTEGER;
        current: INTEGER;
        position: INTEGER;

    BEGIN
        Top := 0;

```

```

Bottom := Limit;
FOR CurrentBottom := Top+1 TO Bottom DO BEGIN
  FOR current := CurrentBottom DOWNTOP Top+1 DO BEGIN
    IF LV[current] < LV[current-1] THEN
      Swap(LV[current], LV[current-1], current, current-1, Lrow);
    { END;}
  END;
END;

IF Won = 0 THEN
BEGIN
  WAIT(Screen);
  Won := 1;
  SetCursorAt(8,11);
  Writeln('Linear Insertion Sort Has Won, Interestingly');
  SIGNAL(Screen);
END;

END;

PROCEDURE QuickSort(QV:Vector; Lrow: INTEGER);

CONST m = 20;
VAR
  i, j, l, r : INTEGER;
  x, w       : ValueType;
  s          : INTEGER;
  stack: array [1..40] of
    RECORD l,r: INTEGER END;

BEGIN
  s := 1;
  stack[1]. l := 0; stack[1]. r := Limit;
  REPEAT (take top request from stack)
    l := stack[s]. l; r := stack[s]. r; s := s-1;
    REPEAT (split QV[l]...QV[r])
      i := l;
      j := r;
      x := QV[(l+r) div 2];
      REPEAT
        WHILE QV[i] < x DO i := i+1;
        WHILE x < QV[j] DO j := j-1;
        IF i <= j THEN
          BEGIN
            Swap(QV[i], QV[j], i, j, Lrow);
            i := i+1; j := j-1;
          END;
        UNTIL i > j;
        IF i < r THEN
          BEGIN (stack request to sort right partition)
            s := s+1; stack[s]. l := i; stack[s]. r := r;
          END;
        r := j;
      UNTIL l >= r
    UNTIL s = 0;
    IF Won = 0 THEN
      BEGIN
        WAIT(Screen);
        Won := 1;
        SetCursorAt(8,16);
        Writeln('QuickSort has WON!!!!, PREDICTABLY');
        SIGNAL(Screen);
      END;
    END;
  END;

```

```

END;

BEGIN
  V[0] := 'Z'; V[1] := 'z'; V[2] := 'Y'; V[3] := 'y';
  V[4] := 'X'; V[5] := 'x'; V[6] := 'W'; V[7] := 'w';
  V[8] := 'V'; V[9] := 'v'; V[10] := 'U'; V[11] := 'u';
  V[12] := 'T'; V[13] := 't'; V[14] := 'S'; V[15] := 's';
  V[16] := 'R'; V[17] := 'r'; V[18] := 'Q'; V[19] := 'q';
  V[20] := 'P'; V[21] := 'p'; V[22] := 'O'; V[23] := 'o';
  V[24] := 'N'; V[25] := 'n'; V[26] := 'M'; V[27] := 'm';
  V[28] := 'L'; V[29] := 'l'; V[30] := 'K'; V[31] := 'k';
  V[32] := 'J';

  Won := 0;
  Screen := 1;
  ClearScreen;
  SetCursorAt(10, 1);
  WRITELN('SORT RACE');
  SetCursorAt(8, 3);
  WRITELN('BUBBLE SORT');
  SetCursorAt(8, 8);
  WRITELN('LINEAR INSERTION');
  SetCursorAt(8, 13);
  WRITELN('QUICKSORT');

  FOR i:= 0 TO Limit DO
    BEGIN
      SetCursorAt(i+1, 5);
      Write(V[i]);
      SetCursorAt(i+1, 10);
      Write(V[i]);
      SetCursorAt(i+1, 15);
      Write(V[i]);
    END;
  SetCursorAt(40, 5);
  WRITELN;

  SetCursorAt(4, 20);
  WRITELN('PRESS RETURN   T W I C E   TO BEGIN THE RACE');
  READLN(A);
  SetCursorAt(4, 20);
  WRITELN('SORT RACE IN PROGRESS   ');

  cobegin
    Bubble(V, 5);
    LinearInsertionSort(V, 10);
    QuickSort(V, 15);
  coend;
  WriteAt(1, 20, ' ');
END {SortRace}.

```

Sorting Algorithm Race in Modula-2

```

MODULE Race;

(* This module implements a sort race between 5 different sorting *)
(* algorithms. The 5 algorithms are executed (pseudo) concurrently and *)
(* their progress is displayed on the screen. This program requires *)
(* that the ANSI.SYS display driver be resident on an IBM PC-type computer. *)
(* Tested using FST Modula-2 for IBM-PC, and Karlsruhe Modula-2 for Sun *)

FROM InOut IMPORT Write, WriteString;
FROM vt100 IMPORT ClearScreen, SetCursorAt; (* this module is shown *)
                                           (* after main program below*)

FROM Process IMPORT DefineProcess, (* Adds a procedure to the list of *)
                                (* processes to executed concurrently*)
                                Croak, (* Allows a process to kill itself. *)
                                GoToSleep, (* Will cause temporary self-suspend. *)
                                StartSystem, (* Starts concurrent execution. *)
                                SIGNAL, (* Semaphore TYPE. *)
                                Init, (* Initializes a user semaphore. *)
                                SEND, (* Signal operation on semaphore. *)
                                WAIT; (* Wait operation on semaphore. *)

CONST Limit = 51;
TYPE ItemType = CHAR;
   Vector = ARRAY[0..Limit] OF ItemType;

VAR A1,A2,A3,A4,A5: Vector;
    Screen: SIGNAL;

PROCEDURE WriteAt(row, col: CARDINAL; c: CHAR);
BEGIN
  WAIT(Screen);
  SetCursorAt(col,row); Write(c);
  SEND(Screen);
END WriteAt;

(* Insertion sort -----*)

PROCEDURE Insertion;
  VAR i,j: CARDINAL;
      row: CARDINAL;
      item: ItemType;
      exit: BOOLEAN;
BEGIN
  row := 5;
  WAIT(Screen);
  SetCursorAt(1,row); WriteString('Insertion:');
  SetCursorAt(14,row); FOR i:= 0 TO HIGH(A1) DO Write(A1[i]); END;
  SEND(Screen);

  FOR i:= 1 TO HIGH(A1) DO
    item := A1[i]; j:= i; exit:= FALSE;
    REPEAT
      DEC(j);
      IF (A1[j] > item) THEN
        A1[j+1]:= A1[j];
      ELSE
        A1[j+1]:= item; exit:= TRUE
      END IF;
    UNTIL exit;
  END FOR;
END Insertion;

```

```

        END;
        WriteAt(row,14+j+1,A1[j+1]);
    UNTIL (j = 0) OR (exit = TRUE);
    IF NOT exit THEN
        A1[0] := item; WriteAt(row,14,A1[0])
    END;
    END; (* FOR i:= 1 to HIGH() *)
    Croak;
END Insertion;

(* Heap Sort procedure -----*)

PROCEDURE HeapSort;
    VAR i : CARDINAL;
        row : CARDINAL;
        swap: ItemType;

    PROCEDURE MakeHeap(low, high: CARDINAL);
        VAR j, k: CARDINAL;
            exit: BOOLEAN;
            item: ItemType;
    BEGIN
        j:= 2*low; item:= A2[low];
        exit:= FALSE;
        WHILE ((j <= high) AND (NOT exit)) DO
            IF (j < high) AND (A2[j+1] > A2[j])
                THEN j:= j+1;
            END;
            IF (item >= A2[j]) THEN
                exit:= TRUE;
            ELSE
                k:= j DIV 2;
                A2[k] := A2[j];
                WriteAt(row,k+14,A2[k]); WriteAt(row,j+14,item);
                j:= 2*j;
            END;
        END;
        A2[j DIV 2] := item;
    END MakeHeap;

    BEGIN
        row := 7;
        WAIT(Screen);
        SetCursorAt(1,row); WriteString('Heap Sort:');
        SetCursorAt(14,row); FOR i:= 0 TO HIGH(A2) DO Write(A2[i]); END;
        SEND(Screen);

        FOR i:= (HIGH(A2) DIV 2) TO 0 BY -1 DO
            MakeHeap(i,HIGH(A2));
        END;
        FOR i:= HIGH(A2) TO 1 BY -1 DO
            swap:= A2[0]; A2[0] := A2[i]; A2[i] := swap;
            WriteAt(row,14,A2[0]); WriteAt(row,14+i,A2[i]);
            MakeHeap(0,i-1);
        END;
        Croak;
    END HeapSort;

(* Shell sort procedure -----*)

PROCEDURE ShellSort;
    CONST NPASS = 4;
    VAR steps: ARRAY[1..NPASS] OF CARDINAL;
        step : CARDINAL;

```

```

        i,j : CARDINAL;
        pass : CARDINAL;
        row : CARDINAL;
        item : ItemType;
        exit : BOOLEAN;
BEGIN
    row := 9;
    WAIT(Screen);
    SetCursorAt(1,row); WriteString('Shell:  ');
    SetCursorAt(14,row); FOR i:= 0 TO HIGH(A3) DO Write(A3[i]); END;
    SEND(Screen);
        (* 'steps' contains decreasing increments for each *)
        (* pass. The last pass has increment 1. *)
    steps[NPASS] := 1;
    FOR pass := NPASS-1 TO 1 BY -1 DO steps[pass] := 2*steps[pass+1]; END;

    FOR pass := 1 TO NPASS DO
        step := steps[pass];
        (* Do a straight insertion sort with 'step' as *)
        (* an increment instead of 1. *)
        i:= step;
        WHILE i <= HIGH(A3) DO (* Use WHILE instead of FOR because *)
            (* loop increment is not a constant. *)
            item := A3[i]; j:= i; exit:= FALSE;
            LOOP
                IF (j < step) OR exit
                THEN EXIT;
                ELSE DEC(j,step); (* exit if decrement would set j < 0 *)
                END;
                IF (A3[j] > item)
                THEN A3[j+step] := A3[j]
                ELSE A3[j+step] := item;
                exit:= TRUE
                END;
                WriteAt(row,14+j+step,A3[j+step]);
            END; (* LOOP *)
            IF (NOT exit) THEN
                A3[0] := item; WriteAt(row,14,A3[0])
            END;
            INC(i,step);
        END; (* WHILE i *)
    END; (* FOR pass *)
    Croak;
END ShellSort;

```

(* Bubble sort procedure -----*)

```

PROCEDURE Bubble;
    VAR i,j: CARDINAL;
        row: CARDINAL;
        temp: ItemType;
BEGIN
    row := 11;
    WAIT(Screen);
    SetCursorAt(1,row); WriteString('Bubble:  ');
    SetCursorAt(14,row); FOR i:= 0 TO HIGH(A4) DO Write(A4[i]); END;
    SEND(Screen);

    i:= HIGH(A4);
    WHILE (i > 0) DO
        j:= 0;
        WHILE (j < i) DO
            IF A4[j] > A4[j+1] THEN

```

```

        temp:= A4[j+1];
        A4[j+1]:= A4[j];
        A4[j]:= temp;
        WriteAt(row,14+j,A4[j]); WriteAt(row,14+j+1,A4[j+1]);
    END;
    j:= j+1;
END;
i:= i-1;
END;
Croak;
END Bubble;

```

(* Merge sort procedure -----*)

```

PROCEDURE MergeSort;
VAR
    i: CARDINAL;
    Q: ItemType;
    TempArray: Vector;
    Left, TopLeft, Right, TopRight, M, CurrentLength: CARDINAL;
    Count, Max: CARDINAL;
    row : CARDINAL;
BEGIN
    row := 13;
    WAIT(Screen);
    SetCursorAt(1,row); WriteString('MergeSort:');
    SetCursorAt(14,row); FOR i:= 0 TO HIGH(A5) DO Write(A5[i]); END;
    SEND(Screen);

    Max := HIGH(A5);
    CurrentLength := 1;
    WHILE CurrentLength < Max DO
        TempArray := A5;
        Left := 0;
        M := 0;
        WHILE Left <= Max DO
            Right := Left + CurrentLength;
            TopLeft := Right;
            IF TopLeft > Max THEN
                TopLeft := Max + 1;
            END;
            TopRight := Right + CurrentLength;
            IF TopRight > Max THEN
                TopRight := Max + 1;
            END;

            WHILE (Left < TopLeft) AND (Right < TopRight) DO
                IF TempArray[Left] <= TempArray[Right] THEN
                    A5[M] := TempArray[Left];
                    WriteAt(row,14+M,A5[M]);
                    Left := Left + 1;
                ELSE
                    A5[M] := TempArray[Right];
                    WriteAt(row,14+M,A5[M]);
                    Right := Right + 1;
                END;
                M := M + 1;
            END;

            WHILE Left < TopLeft DO
                A5[M] := TempArray[Left];
                WriteAt(row,14+M,A5[M]);
                Left := Left + 1;
                M := M + 1;
            END;
        END;
        CurrentLength := CurrentLength * 2;
    END;
END;

```



```

        END;

        WHILE Right < TopRight DO
            A5[M] := TempArray[Right];
            WriteAt(row, 14+M, A5[M]);
            Right := Right + 1;
            M := M + 1;
        END;

        Left := TopRight;
    END;

    CurrentLength := 2 * CurrentLength;
END;
Croak;
END MergeSort;

BEGIN
    A1:= "ZzYyXxWwVvUuTtSsRrQqPpOoNnMmLlKkJjIiHhGgFfEeDdCcBbAa";
    A2:= A1; A3:= A1; A4:= A1; A5:= A1;

    ClearScreen;
    Init(Screen);
    SEND(Screen);

    SetCursorAt(1,20); WriteString('Starting sort processes -----');

    DefineProcess(Insertion, 1000);
    DefineProcess(HeapSort , 1000);
    DefineProcess(ShellSort, 1000);
    DefineProcess(Bubble   , 1000);
    DefineProcess(MergeSort, 1000);

    SetCursorAt(1,20); WriteString('Main procedure idle -----');

    StartSystem;

    SetCursorAt(1,20); WriteString('Main procedure ending -----');

END Race.

DEFINITION MODULE vt100;
    (* EXPORT QUALIFIED ClearScreen, SetCursorAt; *)
    PROCEDURE ClearScreen;
    PROCEDURE SetCursorAt(Column, Row: CARDINAL);
END vt100.

IMPLEMENTATION MODULE vt100;

FROM InOut    IMPORT Write;

    VAR ASCIIOffset: CARDINAL;

    PROCEDURE ClearScreen;
    BEGIN
        Write(CHR(27)); Write([' ');
        Write('2'); Write('J');
    END ClearScreen;

```

```

PROCEDURE SetCursorAt(column, row: CARDINAL);
  BEGIN
    Write(CHR(13));
    Write(CHR(27)); Write(['');
    Write(CHR((row DIV 10) + ASCIIOffset));
    Write(CHR((row MOD 10) + ASCIIOffset));
    Write(';');
    Write(CHR((column DIV 10) + ASCIIOffset));
    Write(CHR((column MOD 10) + ASCIIOffset));
    Write('H');
  END SetCursorAt;

BEGIN
  ASCIIOffset := ORD("0");
END vt100.

```

Sorting Algorithm Race in occam

```
--
-- Sort Race in occam
-- Panos Papaioannou, The George Washington University, 1989
-----
EXTERNAL PROC clear.screen :
EXTERNAL PROC goto.x.y (value x,y) :
EXTERNAL PROC num.from.keyboard (var n) :
EXTERNAL PROC num.to.screen.f (value n,d) :
EXTERNAL PROC str.to.screen (value rubbish[]) :
--
DEF high = 10 :
CHAN BubbleOut,LinearOut,finish1,finish2:
--
PROC Swap(VAR V[], VALUE i,j) =
  VAR Temp :
  SEQ
    Temp := V[i]
    V[i] := V[j]
    V[j] := Temp :
--
PROC delay =
  VAR count:
  SEQ
    count:=0
    SEQ i=[0 FOR 1000]
      count:=count+1 :
--
PROC LinearInsertionSort =
  VAR Top,Bottom,CurrentBottom,current,position,V1[high]:
  SEQ
    V1[0] := -3
    V1[1] := -1
    V1[2] := 1
    V1[3] := 2
    V1[4] := 3
    V1[5] := 6
    V1[6] := 0
    V1[7] := 9
    V1[8] := 8
    V1[9] := 10
    Top := 0
    Bottom := high
    SEQ CurrentBottom = [Top FOR Bottom]
      SEQ
        current:=CurrentBottom
        WHILE ((Top) < current )
          SEQ
            IF
              V1[current] < V1[current-1]
                SEQ
                  Swap(V1, current, current-1)
                  LinearOut ! V1[0]          -- I Want the Screen
                  SEQ i=[1 FOR high-1]
                    LinearOut ! V1[i]
                  current:=current-1
        finish1 ! TRUE:
--
--
```

```

PROC BubbleSort =
  VAR CurrentBottom, AnotherPassNeeded, Top, Current, V2[high]:
  SEQ
    V2[0] := -3
    V2[1] := -1
    V2[2] := 1
    V2[3] := 2
    V2[4] := 3
    V2[5] := 6
    V2[6] := 0
    V2[7] := 9
    V2[8] := 8
    V2[9] := 10
    Top := 0
    CurrentBottom := high
    AnotherPassNeeded := TRUE
    WHILE AnotherPassNeeded AND (CurrentBottom > 0)
      SEQ
        AnotherPassNeeded := FALSE
        SEQ Current = [Top FOR CurrentBottom-1]
        IF
          V2[Current+1] < V2[Current]
            SEQ
              Swap(V2, Current+1, Current)
              BubbleOut ! V2[0]          -- I Want the Screen
              SEQ i=[1 FOR high-1]
              BubbleOut ! V2[i]
              AnotherPassNeeded := TRUE
            CurrentBottom := CurrentBottom - 1
        finish2 ! TRUE :
  --

```

```

PROC ScreenController =
  VAR active1, active2, temp2[high], temp1[high] :
  SEQ
    active1:=TRUE
    active2:=TRUE
    WHILE (active1) OR (active2)
      ALT
        BubbleOut ? temp2[0]
        SEQ
          SEQ i=[1 FOR high-1]
          BubbleOut ? temp2[i]
          goto.x.y (5,5)
          SEQ i=[0 FOR high]
          SEQ
            delay
            num.to.screen.f(temp2[i],3)
        LinearOut ? temp1[0]
        SEQ
          SEQ i=[1 FOR high-1]
          LinearOut ? temp1[i]
          goto.x.y (5,10)
          SEQ i=[0 FOR high]
          SEQ
            delay
            num.to.screen.f(temp1[i],3)
        finish1 ? ANY
      SEQ
        active1:= FALSE
        goto.x.y(5,11)
        str.to.screen(" LINEAR SORT FINISHED")
      finish2 ? ANY

```

```

        SEQ
            active2:= FALSE
            goto.x.y(5,6)
            str.to.screen(" BUBBLE SORT  FINISHED") :

--  MAIN
--
SEQ
    goto.x.y (5 ,4)
    str.to.screen(" BUBBLESORT ")
    goto.x.y (5 ,9)
    str.to.screen(" LINEARSORT ")
    PAR
        ScreenController
        LinearInsertionSort
        BubbleSort

```

Modula-2 Library Modules for Concurrent Programming

DEFINITION MODULE Process;

(* This module provides a simple set of concurrent process services *)
(* including synchronization using binary semaphores. *)

(*EXPORT QUALIFIED DefineProcess,
KillProcess,
GoToSleep,
StartSystem,
SIGNAL,
Init,
SEND,
WAIT,
Awaited;*)

TYPE

SIGNAL; (* Defines a binary semaphore. *)

PROCEDURE DefineProcess(p: PROC; wssize: CARDINAL);

(* Add a procedure to the list of procedures to be executed
concurrently with the call to StartSystem. The procedure p
must be a parameterless procedure. *)

PROCEDURE Croak;

(* Allows a process to terminate its own execution permanently. *)

PROCEDURE GoToSleep;

(* Allows a process to temporarily suspend its own execution. It
is suspended and then immediately added to the run queue. *)

PROCEDURE StartSystem;

(* The procedures specified by previous DefineProcess calls are
executed pseudo-concurrently. *)

PROCEDURE Init(VAR s: SIGNAL);

(* Initializes a user declared SIGNAL (semaphore). *)

PROCEDURE WAIT(VAR s: SIGNAL);

(* Issues a wait operation on the specified SIGNAL. *)

PROCEDURE SEND(VAR s: SIGNAL);

(* Issues a signal operation on the specified SIGNAL. *)

PROCEDURE Awaited(s: SIGNAL): BOOLEAN;

(* Returns TRUE if there are processes WAITing on the specified SIGNAL. *)

END Process.

(* ----- *)

IMPLEMENTATION MODULE Process;

(* This module provides a simple set of concurrent process services *)
(* including synchronization using binary semaphores. *)

FROM SYSTEM IMPORT ADDRESS, (* ADDRESS type *)
NEWPROCESS, (* Creates a process *)
TRANSFER; (* Coroutine transfer of control *)

```

(* FROM System IMPORT Terminate; *) (* Terminate program, exit to DOS *)

FROM Storage IMPORT ALLOCATE;

FROM Queue  IMPORT Queue, (* type *)
               Qmakeempty, Qempty, Qinsert, Qremove, Qdefine;

FROM InOut  IMPORT WriteString, WriteLn;

TYPE
  SIGNAL      = POINTER TO semaphore;

  semaphore = RECORD
    sent : BOOLEAN;
    procs: Queue
  END;

  processptr = POINTER TO ADDRESS;

VAR
  MAIN      : processptr;
  currentprocess: processptr;
  readyqueue : Queue;

PROCEDURE deadlockhandler;
BEGIN
  WriteString('Deadlock has occurred');
  WriteLn;
  TRANSFER( currentprocess^, MAIN^ );
END deadlockhandler;

PROCEDURE Init( VAR s: SIGNAL );
BEGIN
  NEW(s);
  s^.sent := FALSE;
  Qdefine(s^.procs);
  Qmakeempty(s^.procs);
END Init;

PROCEDURE SEND( VAR s : SIGNAL);
  VAR prevprocess: processptr;
BEGIN
  IF NOT Qempty( s^.procs ) (* a process is waiting on semaphore *)
  THEN Qinsert( readyqueue, currentprocess);
    prevprocess := currentprocess;
    Qremove(s^.procs, currentprocess);
    TRANSFER( prevprocess^, currentprocess^);
  ELSE s^.sent := TRUE;
    IF NOT Qempty( readyqueue )
    THEN Qinsert( readyqueue, currentprocess);
      prevprocess := currentprocess;
      Qremove(readyqueue, currentprocess);
      TRANSFER( prevprocess^, currentprocess^);
    END
  END
END SEND;

PROCEDURE WAIT( VAR s: SIGNAL);
  VAR prevprocess: processptr;
BEGIN
  IF s^.sent
  THEN s^.sent := FALSE
  ELSIF NOT Qempty( readyqueue )

```

```

        THEN Qinsert( s^.procs, currentprocess);
        prevprocess := currentprocess;
        Qremove(readyqueue, currentprocess);
        TRANSFER( prevprocess^, currentprocess^);
    ELSE deadlockhandler;
END
END WAIT;

PROCEDURE Awaited( s: SIGNAL): BOOLEAN;
BEGIN
    RETURN NOT Qempty(s^.procs);
END Awaited;

PROCEDURE DefineProcess( p: PROC; wssize: CARDINAL);
    VAR workspace : ADDRESS;
        newprocess : processptr;
BEGIN
    ALLOCATE( workspace, wssize);
    NEW( newprocess );
    NEWPROCESS(p, workspace, wssize, newprocess^);
    Qinsert( readyqueue, newprocess);
END DefineProcess;

PROCEDURE GoToSleep;
    VAR prevprocess : processptr;
BEGIN
    IF NOT Qempty( readyqueue )
    THEN Qinsert( readyqueue, currentprocess);
        prevprocess := currentprocess;
        Qremove(readyqueue, currentprocess);
        TRANSFER( prevprocess^, currentprocess^);
    ELSE deadlockhandler;
    END;
END GoToSleep;

PROCEDURE Croak;
    VAR killedprocess : processptr;
BEGIN
    NEW( killedprocess );
    IF NOT Qempty( readyqueue )
    THEN Qremove(readyqueue, currentprocess);
        TRANSFER( killedprocess^, currentprocess^);
    ELSE TRANSFER( killedprocess^, MAIN^);
    END;
END Croak;

PROCEDURE StartSystem;
BEGIN
    IF NOT Qempty( readyqueue )
    THEN
        NEW( currentprocess );
        NEW( MAIN );
        Qremove( readyqueue, currentprocess );
        TRANSFER( MAIN^, currentprocess^ );
    END;
END StartSystem;

BEGIN (* Process module initialization *)
    Qdefine( readyqueue);
    Qmakeempty( readyqueue);
END Process.

```


Queue Abstract Data Type in Modula-2

DEFINITION MODULE Queue;

```
(* This module exports a Queue abstract data type and the supporting *)
(* queue services: *)
(* Qdefine - Initializes a queue. *)
(* Qmakeempty - Force a queue to empty. *)
(* Qinsert - Enqueue an item. *)
(* Qremove - Remove the next item from the queue *)
(* Qempty - Is the queue empty? *)
```

FROM SYSTEM IMPORT ADDRESS;

TYPE Queue;

TYPE QueueItem = ADDRESS;

PROCEDURE Qdefine(VAR Q: Queue);

PROCEDURE Qempty(Q: Queue) : BOOLEAN;

PROCEDURE Qinsert(VAR Q: Queue; Item: QueueItem);

PROCEDURE Qmakeempty(VAR Q: Queue);

PROCEDURE Qremove(VAR Q: Queue; VAR Item: QueueItem);

VAR Qoverflow: BOOLEAN;

Qunderflow: BOOLEAN;

END Queue.

(* ----- *)

IMPLEMENTATION MODULE Queue;

FROM Storage IMPORT ALLOCATE, DEALLOCATE;

TYPE Queue = POINTER TO QueueHeader;

QueueBlockPtr = POINTER TO QueueBlock;

```
QueueBlock =
  RECORD
    item : QueueItem;
    next : QueueBlockPtr;
  END;
```

```
QueueHeader =
  RECORD
    head: QueueBlockPtr;
    tail: QueueBlockPtr;
  END;
```

PROCEDURE Qdefine(VAR Q: Queue);

BEGIN

ALLOCATE(Q, SIZE(QueueHeader));

Q^.head := NIL;

Q^.tail := NIL;

```

END Qdefine;

PROCEDURE Qmakeempty(VAR Q: Queue);
  VAR Qb: QueueBlockPtr;
BEGIN
  Qb := Q^.head;
  Q^.head := NIL;
  Q^.tail := NIL;
  WHILE (Qb <> NIL) DO
    DEALLOCATE(Qb, SIZE(QueueBlock));
  END
END Qmakeempty;

PROCEDURE Qempty(Q: Queue) : BOOLEAN;
BEGIN
  RETURN Q^.head=NIL;
END Qempty;

PROCEDURE Qinsert(VAR Q: Queue; Item: QueueItem);
  VAR Qb : QueueBlockPtr;
BEGIN
  ALLOCATE(Qb, SIZE(QueueBlock));
  Qb^.item := Item;
  Qb^.next := NIL;
  IF Qempty(Q)
    THEN Q^.head := Qb;
    ELSE Q^.tail^.next := Qb;
  END;
  Q^.tail := Qb;
END Qinsert;

PROCEDURE Qremove(VAR Q: Queue; VAR Item : QueueItem);
  VAR Qb: QueueBlockPtr;
BEGIN
  IF Qempty(Q)
    THEN Qunderflow := TRUE;
    ELSE Qb := Q^.head;
        Q^.head := Q^.head^.next;
        Item := Qb^.item;
  END;
END Qremove;

END Queue.

```

The Software Engineering Institute (SEI) is a federally funded research and development center, operated by Carnegie Mellon University under contract with the United States Department of Defense.

The SEI Software Engineering Curriculum Project is developing a wide range of materials to support software engineering education. A *curriculum module* (CM) identifies and outlines the content of a specific topic area, and is intended to be used by an instructor in designing a course. A *support materials package* (SM) contains materials related to a module that may be helpful in teaching a course. An *educational materials package* (EM) contains other materials not necessarily related to a curriculum module. Other publications include software engineering curriculum recommendations and course designs.

SEI educational materials are being made available to educators throughout the academic, industrial, and government communities. The use of these materials in a course does not in any way constitute an endorsement of the course by the SEI, by Carnegie Mellon University, or by the United States government.

SEI curriculum modules, support materials, and educational materials may be copied or incorporated into other materials, but not for profit, provided that appropriate credit is given to the SEI and to the original author of the materials.

Comments on SEI educational materials and requests for additional information should be addressed to the Software Engineering Curriculum Project, Software Engineering Institute, Carnegie Mellon University, Pittsburgh, Pennsylvania 15213. Electronic mail can be sent to education@sei.cmu.edu on the Internet.

Curriculum Modules (* Support Materials available)

CM-1 [superseded by CM-19]
CM-2 Introduction to Software Design
CM-3 The Software Technical Review Process*
CM-4 Software Configuration Management*
CM-5 Information Protection
CM-6 Software Safety
CM-7 Assurance of Software Quality
CM-8 Formal Specification of Software*
CM-9 Unit Testing and Analysis
CM-10 Models of Software Evolution: Life Cycle and Process
CM-11 Software Specifications: A Framework
CM-12 Software Metrics
CM-13 Introduction to Software Verification and Validation
CM-14 Intellectual Property Protection for Software
CM-15 Software Development and Licensing Contracts
CM-16 Software Development Using VDM
CM-17 User Interface Development*
CM-18 [superseded by CM-23]
CM-19 Software Requirements
CM-20 Formal Verification of Programs
CM-21 Software Project Management
CM-22 Software Design Methods for Real-Time Systems*
CM-23 Technical Writing for Software Engineers
CM-24 Concepts of Concurrent Programming
CM-25 Language and System Support for Concurrent Programming*

Educational Materials

EM-1 Software Maintenance Exercises for a Software Engineering Project Course
EM-2 APSE Interactive Monitor: An Artifact for Software Engineering Education

