THE SOLO OPERATING SYSTEM: PROCESSES, MONITORS AND CLASSES

PER BRINCH HANSEN

(1976)

This paper describes the implementation of the Solo operating system written in Concurrent Pascal. It explains the overall structure and details of the system in which concurrent processes communicate by means of a hierarchy of monitors and classes. The concurrent program is a sequence of nearly independent components of less than one page of text each. The system has been operating since May 1975.

INTRODUCTION

This is a description of the program structure of the Solo operating system. Solo is a single-user operating system for the PDP 11/45 computer written in the programming language Concurrent Pascal (Brinch Hansen 1976a, 1976b).

The main idea in Concurrent Pascal is to divide the global data structures of an operating system into small parts and define the meaningful operations on each of them. In Solo, for example, there is a data structure, called a resource, that is used to give concurrent processes exclusive access to a disk. This data structure can only be accessed by means of two procedures that request and release access to the disk. The programmer specifies that these are the only operations one can perform on a resource, and the compiler checks that this rule is obeyed in the rest of the system. This approach

P. Brinch Hansen, The Solo operating system: processes, monitors and classes. *Software—Practice and Experience 6*, 2 (April–June 1976), 165–200. Copyright © 1975, Per Brinch Hansen.

to program reliability has been called *resource protection at compile-time* (Brinch Hansen 1973). It makes programs more reliable by detecting incorrect interactions of program components before they are put into operation. It makes them more efficient by reducing the need for hardware protection mechanisms.

The combination of a data structure and the operations used to access it is called an *abstract data type*. It is abstract because the rest of the system need only know what operations one can perform on it but can ignore the details of how they are carried out. A Concurrent Pascal program is constructed from three kinds of abstract data types: processes, monitors and classes. *Processes* perform concurrent operations on data structures. They use *monitors* to synchronize themselves and exchange data. They access private data structures by means of *classes*. Brinch Hansen (1975a) is an overview of these concepts and their use in concurrent programming.

Solo is the first major example of a hierarchical concurrent program implemented in terms of abstract data types. It has been in use since May 1975. This is a complete, annotated program listing of the system. It also explains how the system was tested systematically.

PROGRAM STRUCTURE

Solo consists of a hierarchy of *program layers*, each of which controls a particular kind of computer resource, and a set of concurrent processes that use these resources (Fig. 1):

- *Resource management* controls the scheduling of the operator's console and the disk among concurrent processes.
- *Console management* lets processes communicate with the operator after they have gained access to the console.
- *Disk management* gives processes access to the disk files and a catalog describing them.
- *Program management* fetches program files from disk into core on demand from processes that wish to execute them.
- *Buffer management* transmits data among processes.

These facilities are used by seven concurrent processes:

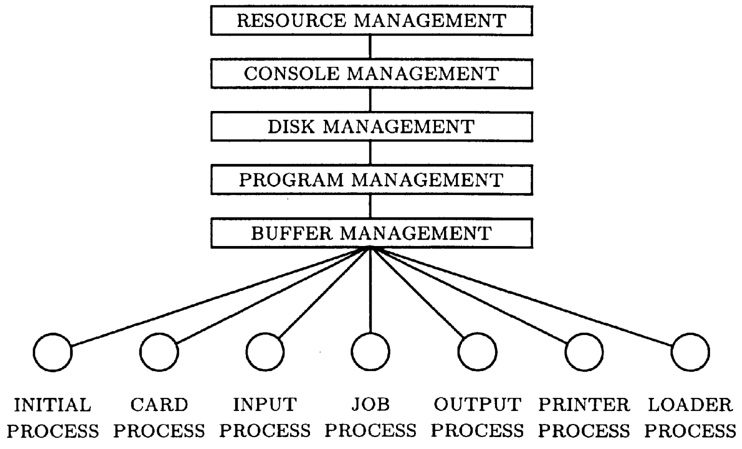

Figure 1 Program layers and processes.

- A *job process* executes Pascal programs upon request from the operator.
- Two *input/output processes* produce and consume the data of the job process.
- A *card process* feeds punched cards to the input process which then removes trailing blanks from them and packs the text into blocks.
- A *printer process* prints lines that are unpacked from blocks and sent to it by the output process.
- A *loader process* preempts and reinitializes the operating system when the operator pushes the bell key on the console.
- An *initial process* starts up the rest of the system after system loading.

The term *program layer* is only used as a convenient way of explaining the gross division of labor within the system. It cannot be represented by any language notation in Concurrent Pascal.

ABSTRACT DATA TYPES

Each program layer consists of one or more abstract data types (monitors and classes).

Resource management

A *fifo* class implements a first-in, first-out queue that is used to maintain multiprocess queues and message buffers.

A *resource* monitor gives processes exclusive access to a computer resource. It is used to control disk access.

A *typewriter resource* monitor gives processes exclusive access to a console and tells them whether they need to identify themselves to the operator.

Console management

A *typewriter* class transmits a single line between a process and a console (but does not give a process exclusive access to it).

A *terminal* class gives a process the illusion that it has its own private console by giving it exclusive access to the operator for input or output of a single line.

A *terminal stream* makes a terminal look character oriented.

Disk management

A *disk* class can access a page anywhere on disk (but does not give a process exclusive access to it). It uses a terminal to report disk failure.

A *disk file* can access any page belonging to a particular file. The file pages, which may be scattered on disk, are addressed indirectly through a page map. The disk address of the page map identifies the file. It uses a disk to access the map and its pages.

A *disk table* class makes a disk catalog of files look like an array of entries, some of which describe files, and some of which are empty. The entries are identified by numeric indices. It uses a disk file to access the catalog page by page.

A *disk catalog* monitor can look up files in a disk catalog by means of their names. It uses a resource to get exclusive acess to the disk and a disk table to scan the catalog.

A *data file* class gives a process access to a named disk file. It uses a resource, a disk catalog, and a disk file to access the disk.

Program management

A *program file* class can load a named disk file into core when a process wishes to execute it. It uses a resource, a disk catalog, and a disk file to do this.

A *program stack* monitor keeps track of nested program calls within a process.

Buffer management

The *buffer* monitors transmit various kinds of messages between processes: arguments (scalars or identifiers), lines, and pages.

The following defines the purpose, specification, and implementation of each of these abstract data types.

INPUT/OUTPUT

The following data types are used in elementary input/output operations:

```
type iodevice =
  (typedevice, diskdevice, tapedevice, printdevice, carddevice);

type iooperation = (input, output, move, control);

type ioarg = (writeeof, rewind, upspace, backspace);

type ioresult =
  (complete, intervention, transmission, failure,
  endfile, endmedium, startmedium);

type ioparam =
  record
    operation: iooperation;
    status: ioresult;
    arg: ioarg
  end;

const nl = '(:10:)'; ff = '(:12:)'; cr = '(:13:)'; em = '(:25:)';

const linelength = 132;
```

```
type line = array [1..linelength] of char;

const pagelength = 512;
type page = array [1..pagelength] of char;
```

They define the identifiers of peripheral devices, input/output operations and their results as well as the data types to be transferred (printer lines or disk pages). The details of input/output operations are explained in Brinch Hansen (1975b).

FIFO QUEUE

type fifo = class(limit: integer)

A fifo keeps track of the length and the head and tail indices of an array used as a first-in, first-out queue (but does not contain the queue elements themselves). A fifo is initialized with a constant that defines its range of queue indices 1..limit. A user of a fifo must ensure that the length of the queue remains within its physical limit:

$$0 \leq \text{arrivals} - \text{departures} \leq \text{limit}$$

The routines of a fifo are:

function arrival: integer

Returns the index of the next queue element in which an arrival can take place.

function departure: integer

Returns the index of the next queue element from which a departure can take place.

function empty: boolean

Defines whether the queue is empty (arrivals = departures).

function full: boolean

Defines whether the queue is full (arrivals = departures + limit).

Implementation:

A fifo queue is represented by its head, tail and length. The Concurrent Pascal compiler will ensure that these variables are only accessed by the routines of the class. In general, a class variable can only be accessed by calling one of the routines associated with it (Brinch Hansen 1975a). The final statement of the class is executed when an instance of a fifo queue is declared and initialized.

```
type fifo =
class(limit: integer);

var head, tail, length: integer;

function entry arrival: integer;
begin
  arrival := tail;
  tail := tail mod limit + 1;
  length := length + 1;
end;

function entry departure: integer;
begin
  departure := head;
  head := head mod limit + 1;
  length := length − 1;
end;

function entry empty: boolean;
begin empty := (length = 0) end;

function entry full: boolean;
begin full := (length = limit) end;

begin head := 1; tail := 1; length := 0 end;
```

RESOURCE

type resource = monitor

A resource gives exclusive access to a computer resource (but does not perform any operations on the resource itself). A user of a resource must request it before using it and release it afterwards. If the resource is released within a finite time it will also become available to any process requesting it within a finite time. In short, the resource scheduling is fair.

procedure request

Gives the calling process exclusive access to the resource.

procedure release

Makes the resource available for other processes.

Implementation:

A resource is represented by its state (free or used) and a queue of processes waiting for it. The multiprocess queue is represented by two data structures: an array of single-process queues and a fifo to keep track of the queue indices.

The initial statement at the end of the monitor sets the resource state to free and initializes the fifo variable with a constant defining the total number of processes that can wait in the queue.

The compiler will ensure that the monitor variables only can be accessed by calling the routine entries associated with it. The generated code will ensure that at most one process at a time is executing a monitor routine (Brinch Hansen 1975a). The monitor can delay and (later) continue the execution of a calling process.

A routine associated with a class or monitor is called by mentioning the class or monitor variable followed by the name of the routine. As an example

```
next.arrival
```

will perform an arrival operation on the fifo variable next.

```
const processcount = 7;
type processqueue = array [1..processcount] of queue;

type resource =
monitor

var free: boolean; q: processqueue; next: fifo;

procedure entry request;
begin
  if free then free := false
  else delay(q[next.arrival]);
end;

procedure entry release;
begin
  if next.empty then free := true
  else continue(q[next.departure]);
end;

begin free := true; init next(processcount) end;
```

TYPEWRITER RESOURCE

type typeresource = monitor

A typewriter resource gives processes exclusive access to a typewriter console. A calling process supplies an identification of itself and is told whether it needs to display it to the operator. The resource scheduling is fair as explained in the definition of the *resource* monitor.

procedure request(text: line; var changed: boolean)

Gives the calling process exclusive access to the resource. The process identifies itself by a text line. A boolean changed defines whether this is the same identification that was used in the last call of request (in which case there is no need to display it to the operator again).

procedure release

Makes the resource available again for other processes.

Implementation:

```
type typeresource =
monitor

var free: boolean; q: processqueue; next: fifo; header: line;

procedure entry request(text: line; var changed: boolean);
begin
  if free then free := false
  else delay(q[next.arrival]);
  changed := (header <> text);
  header := text;
end;

procedure entry release;
begin
  if next.empty then free := true
  else continue(q[next.departure]);
end;

begin
  free := true; header[1] := nl;
  init next(processcount);
end;
```

TYPEWRITER

type typewriter = class(device: iodevice)

A typewriter can transfer a text line to or from a typewriter console. It does not identify the calling process on the console or give it exclusive access to it. A typewriter is initialized with the identifier of the device it controls.

A newline character (nl) terminates the input or output of a line. A line that exceeds 73 characters is forcefully terminated by a newline character.

procedure write(text: line)

Writes a line on the typewriter.

procedure read(var text: line)

Rings the bell on the typewriter and reads a line from it. Single characters or the whole line can be erased and retyped by typing *control c* or *control l*. The typewriter responds to erasure by writing a question mark.

Implementation:

The procedure writechar is not a routine entry; it can only be called within the typewriter class. The standard procedure io delays the calling process until the transfer of a single character is completed.

```
type typewriter =
class(device: iodevice);

const linelimit = 73;
  cancelchar = '(:3:)'; "control c"
  cancelline = '(:12:)'; "control l"

procedure writechar(x: char);
var param: ioparam; c: char;
begin
  param.operation := output;
  c := x;
  io(c, param, device);
end;

procedure entry write(text: line);
var param: ioparam; i: integer; c: char;
begin
  param.operation := output;
  i := 0;
  repeat
    i := i + 1; c := text[i];
    io(c, param, device);
  until (c = nl) or (i = linelimit);
  if c <> nl then writechar(nl);
end;

procedure entry read(var text: line);
```

```
const bel = '(:7:)';
var param: ioparam; i: integer; c: char;
begin
  writechar(bel);
  param.operation := input;
  i := 0;
  repeat
    io(c, param, device);
    if c = cancelline then
      begin
        writechar(nl);
        writechar('?');
        i := 0;
      end
    else if c = cancelchar then
      begin
        if i > 0 then
          begin
            writechar('?');
            i := i − 1;
          end
      end
    else
      begin i := i + 1; text[i] := c end
  until (c = nl) or (i = linelimit);
  if c <> nl then
    begin
      writechar(nl);
      text[linelimit + 1] := nl;
    end;
end;

begin end;
```

TERMINAL

type terminal = class(access: typeresource)

A terminal gives a single process exclusive access to a typewriter, identifies the process to the operator and transfers a line to or from the device. The terminal uses a typewriter resource to get exclusive access to the device.

procedure read(header: line; var text: line)

Writes a header (if necessary) on the typewriter and reads a text line from it.

procedure write(header, text: line)

Writes a header (if necessary) followed by a text line on the typewriter.

The header identifies the calling process. It is only output if it is different from the last header output on the typewriter.

Implementation:

A class or monitor can only call other classes or monitors if they are declared as variables within it or passed as parameters during initialization (Brinch Hansen 1975a). So a terminal can only call the monitor *access* and the class *unit*. These access rights are checked during compilation.

```
type terminal =
class(access: typeresource);

var unit: typewriter;

procedure entry read(header: line; var text: line);
var changed: boolean;
begin
  access.request(header, changed);
  if changed then unit.write(header);
  unit.read(text);
  access.release;
end;

procedure entry write(header, text: line);
```

```
var changed: boolean;
begin
  access.request(header, changed);
  if changed then unit.write(header);
  unit.write(text);
  access.release;
end;

begin init unit(typedevice) end;
```

TERMINAL STREAM

type terminalstream = class(operator: terminal)

A terminal stream enables a process to identify itself once and for all and then proceed to read and write single characters on a terminal. A terminal stream uses a terminal to input or output a line at a time.

procedure read(var c: char)

Reads a character from the terminal.

procedure write(c: char)

Writes a character on the terminal.

procedure reset(text: line)

Identifies the calling process.

Implementation:

The terminal stream contains two line buffers for input and output.

```
type terminalstream =
class(operator: terminal);

const linelimit = 80;

var header: line; endinput: boolean;
  inp, out: record count: integer; text: line end;
```

```
procedure initialize(text: line);
begin
  header := text;
  endinput := true;
  out.count := 0;
end;

procedure entry read(var c: char);
begin
  with inp do
    begin
      if endinput then
        begin
          operator.read(header, text);
          count := 0;
        end;
      count := count + 1;
      c := text[count];
      endinput := (c = nl);
    end;
end;

procedure entry write(c: char);
begin
  with out do
    begin
      count := count + 1;
      text[count] := c;
      if (c = nl) or (count = linelimit) then
        begin
          operator.write(header, text);
          count := 0;
        end;
    end;
end;

procedure entry reset(text: line);
```

```
begin initialize(text) end;

begin initialize('unidentified:(:10:)') end;
```

DISK

type disk = class(typeuse: typeresource)

A disk can transfer any page to or from a disk device. A disk uses a typewriter resource to get exclusive access to a terminal to report disk failure. After a disk failure, the disk writes a message to the operator and repeats the operation when he types a newline character.

procedure read(pageaddr: integer; var block: univ page)

Reads a page identified by its absolute disk address.

procedure write(pageaddr: integer; var block: univ page)

Writes a page identified by its absolute disk address.

A page is declared as a universal type to make it possible to use the disk to transfer pages of different types (and not just text).

Implementation:

The standard procedure io delays the calling process until the disk transfer is completed (Brinch Hansen 1975b).

```
type disk =
class(typeuse: typeresource);

var operator: terminal;

procedure transfer(command: iooperation;
  pageaddr: univ ioarg; var block: page);
var param: ioparam; response: line;
begin
  with param, operator do
    begin
      operation := command;
```

```
          arg := pageaddr;
          io(block, param, diskdevice);
          while status <> complete do
            begin
              write('disk:(:10:)', 'error(:10:)');
              read('push return(:10:)', response);
              io(block, param, diskdevice);
            end;
        end;
    end;

    procedure entry read(pageaddr: integer; var block: univ page);
    begin transfer(input, pageaddr, block) end;

    procedure entry write(pageaddr: integer; var block; univ page);
    begin transfer(output, pageaddr, block) end;

    begin init operator(typeuse) end;
```

DISK FILE

type diskfile = class(typeuse: typeresource)

A disk file enables a process to access a disk file consisting of a fixed number of pages ($\leq$ 255). A disk file uses a typewriter resource to get exclusive access to the operator after a disk failure.

The disk file is identified by the absolute address of a page map that defines the length of the file and the disk addresses of its pages. To a calling process the pages of a file are numbered 1, 2, . . ., length.

Initially, the file is closed (inaccessible). A user of a file must open it before using it and close it afterwards. Read and write have no effect if the file is closed or if the page number is outside the range 1..length.

procedure open(mapaddr: integer)

Makes a disk file with a given page map accessible.

procedure close

Makes the disk file inaccessible.

function length: integer

Returns the length of the disk file (in pages). The length of a closed file is zero.

procedure read(pageno: integer; **var** *block:* **univ** *page)*

Reads a page with a given number from the disk file.

procedure write(pageno: integer; **var** *block:* **univ** *page)*

Writes a page with a given number on the disk file.

Implementation:

The variable *length* is prefixed with the word *entry.* This means that its value can be used directly outside the class. It can, however, only be changed within the class. So a *variable entry* is similar to a function entry. Variable entries can only be used within classes.

```
const maplength = 255;
type filemap =
  record
    filelength: integer;
    pageset: array [1..maplength] of integer
  end;

type diskfile =
class(typeuse: typeresource);

var unit: disk; map: filemap; opened: boolean;

entry length: integer;

function includes(pageno: integer): boolean;
begin
  includes := opened &
    ( 1 <= pageno) & (pageno <= length);
end;

procedure entry open(mapaddr: integer);
```

```
begin
  unit.read(mapaddr, map);
  length := map.filelength;
  opened := true;
end;

procedure entry close;
begin
  length := 0;
  opened := false;
end;

procedure entry read(pageno: integer; var block: univ page);
begin
  if includes(pageno) then
    unit.read(map.pageset[pageno], block);
end;

procedure entry write(pageno: integer; var block: univ page);
begin
  if includes(pageno) then
    unit.write(map.pageset[pageno], block);
end;

begin
  init unit(typeuse);
  length := 0;
  opened := false;
end;
```

CATALOG STRUCTURE

The disk contains a catalog of all files. The following data types define the structure of the catalog:

```
const idlength = 12;
type identifier = array [1..idlength] of char;

type filekind = (empty, scratch, ascii, seqcode, concode);
type fileattr =
  record
    kind: filekind;
    addr: integer;
    protected: boolean;
    notused: array [1..5] of integer
  end;

type catentry =
  record
    id: identifier;
    attr: fileattr;
    key, searchlength: integer
  end;

const catpagelength = 16;
type catpage = array [1..catpagelength] of catentry;

const cataddr = 154;
```

The catalog is itself a file defined by a page map stored at the *catalog address.* Every *catalog page* contains a fixed number of catalog entries. A *catalog entry* describes a file by its identifier, attributes and hash key. The search length defines the number of files that have a hash key equal to the index of this entry. It is used to limit the search for a non-existing file name.

The *file attributes* are its kind (empty, scratch, ascii, sequential or concurrent code), the address of its page map, and a boolean defining whether it is protected against accidental deletion or overwriting. The latter is checked by all system programs operating on the disk, but not by the operating system. Solo provides a mechanism for protection, but does not enforce it.

DISK TABLE

type disktable = class(typeuse: typeresource; cataddr: integer)

A disk table makes a disk catalog look like an array of catalog entries identified by numeric indices 1, 2, ..., length. A disk table uses a typewriter resource to get exclusive access to the operator after a disk failure and a catalog address to locate a catalog on disk.

function length: integer

Defines the number of entries in the catalog.

procedure read(i: integer; var elem: catentry)

Reads entry number *i* in the catalog. If the entry number is outside the range 1..length the contents of the entry is undefined.

Implementation:

A disk table stores the most recently used catalog page to make a sequential search of the catalog fast.

```
type disktable =
class(typeuse: typeresource; cataddr: integer);

var file: diskfile; pageno: integer; block: catpage;

entry length: integer;

procedure entry read(i: integer; var elem: catentry);
var index: integer;
begin
  index := (i − 1) div catpagelength + 1;
  if pageno <> index then
    begin
      pageno := index;
      file.read(pageno, block);
    end;
  elem := block[(i − 1) mod catpagelength + 1];
end;

begin
  init file(typeuse);
  file.open(cataddr);
  length := file.length ∗ catpagelength;
  pageno := 0;
end;
```

DISK CATALOG

type diskcatalog =
monitor(typeuse: typeresource; diskuse: resource; cataddr: integer)

The disk catalog describes all disk files by means of a set of named entries that can be looked up by processes. A disk catalog uses a resource to get exclusive access to the disk during a catalog lookup and a typewriter resource to get exclusive access to the operator after a disk failure. It uses a catalog address to locate the catalog on disk.

procedure lookup(id: identifier; var attr: fileattr; var found: boolean)

Searches for a catalog entry describing a file with a given identifier and indicates whether it found it. If so, it also returns the file attributes.

Implementation:

A disk catalog uses a disk table to make a cyclical search for an identifier. The initial catalog entry is selected by hashing. The search stops when the identifier is found or when there are no more entries with the same hash key. The disk catalog has exclusive access to the disk during the lookup to prevent competing processes from causing disk arm movement.

```
type diskcatalog =
monitor(typeuse: typeresource; diskuse: resource; cataddr: integer);

var table: disktable;

function hash(id: identifier): integer;
var key, i: integer; c: char;
begin
  key := 1; i := 0;
  repeat
    i := i + 1; c := id[i];
    if c <> ' ' then
      key := key * ord(c) mod table.length + 1;
  until (c = ' ') or (i = idlength);
  hash := key;
end;
```

```
procedure entry lookup(id: identifier;
  var attr: fileattr; var found: boolean);
var key, more, index: integer; elem: catentry;
begin
  diskuse.request;
  key := hash(id);
  table.read(key, elem);
  more := elem.searchlength;
  index := key; found := false;
  while not found & (more > 0) do
    begin
      table.read(index, elem);
      if elem.id = id then
        begin attr := elem.attr; found := true end
      else
        begin
          if elem.key = key then more := more − 1;
          index := index mod table.length + 1;
        end;
    end;
  diskuse.release;
end;

begin init table(typeuse, cataddr) end;
```

DATA FILE

type datafile =
class(typeuse: typeresource; diskuse: resource; catalog: diskcatalog)

A data file enables a process to access a disk file by means of its name in a diskcatalog. The pages of a data file are numbered 1, 2, . . ., length. A data file uses a resource to get exclusive access to the disk during a page transfer and a typewriter resource to get exclusive access to the operator after disk failure. It uses a catalog to look up the the file.

Initially a data file is inaccessible (closed). A user of a data file must open it before using it and close it afterwards. If a process needs exclusive access to a data file while using it, this must be ensured at higher levels of programming.

procedure open(id: identifier; var found: boolean)

Makes a file with a given identifier accessible if it is found in the catalog.

procedure close

Makes the file inaccessible.

procedure read(pageno: integer; var block: univ page)

Reads a page with a given number from the file. It has no effect if the file is closed or if the page number is outside the range 1..length.

procedure write(pageno: integer; var block: univ page)

Writes a page with a given number on the file. It has no effect if the file is closed or if the page number is outside the range 1..length.

function length: integer

Defines the number of pages in the file. The length of a closed file is zero.

Implementation:

```
type datafile =
class(typeuse: typeresource; diskuse: resource; catalog: diskcatalog);

var file: diskfile; opened: boolean;

entry length: integer;

procedure entry open(id: identifier; var found: boolean);
var attr: fileattr;
begin
  catalog.lookup(id, attr, found);
  if found then
    begin
      diskuse.request;
      file.open(attr.addr);
      length := file.length;
      diskuse.release;
    end;
  opened := found;
end;

procedure entry close;
begin
  file.close;
  length := 0;
  opened := false;
end;

procedure entry read(pageno: integer; var block: univ page);
begin
  if opened then
    begin
      diskuse.request;
      file.read(pageno, block);
      diskuse.release;
    end;
end;

procedure entry write(pageno: integer; var block: univ page);
begin
  if opened then
    begin
      diskuse.request;
      file.write(pageno, block);
      diskuse.release;
    end;
end;

begin
  init file(typeuse);
  length := 0;
  opened := false;
end;
```

PROGRAM FILE

type progfile =
class(typeuse: typeresource; diskuse: resource; catalog: diskcatalog)

A program file can transfer a sequential program from a disk file into core. The program file is identified by its name in a disk catalog. A program file uses a resource to get exclusive access to the disk during program loading and a typewriter resource to get exclusive access to the operator after disk failure. It uses a disk catalog to look up the file.

procedure open(id: identifier; var state: progstate)

Loads a program with a given identifier from disk and returns its state. The program state is one of the following: ready for execution, not found, the disk file is not sequential code, or the file is too big to be loaded into core.

function store: progstore

Defines the variable in which the program file is stored. A program store is an array of disk pages.

Implementation:

A program file has exclusive access to the disk until it has loaded the entire program. This is to prevent competing processes from slowing down program loading by causing disk arm movement.

```
type progstate = (ready, notfound, notseq, toobig);

const storelength1 = 40;
type progstore1 = array [1..storelength1] of page;

type progfile1 =
class(typeuse: typeresource; diskuse: resource; catalog: diskcatalog);

var file: diskfile;

entry store: progstore1;

procedure entry open(id: identifier; var state: progstate);
```

```
var attr: fileattr; found: boolean; pageno: integer;
begin
  catalog.lookup(id, attr, found);
  with diskuse, file, attr do
    if not found then state := notfound
    else if kind <> seqcode then state := notseq
    else
      begin
        request;
        open(addr);
        if length <= storelength1 then
          begin
            for pageno := 1 to length do
              read(pageno, store[pageno]);
            state := ready;
          end
        else state := toobig;
        close;
        release;
      end;
end;

begin init file(typeuse) end;
```

Solo uses two kinds of program files (progfile1 and progfile2); one for large programs and another one for small ones. They differ only in the dimension of the program store used. The need to repeat the entire class definition to handle arrays of different lengths is an awkward inheritance from Pascal.

PROGRAM STACK

type progstack = monitor

A program stack maintains a last-in, first-out list of identifiers of programs that have called one another. It enables a process to keep track of nested calls of sequential programs.

For historical reasons a program stack was defined as a monitor. In the present version of the system it might as well have been a class.

function space: boolean

Tells whether there is more space in the program stack.

function any: boolean

Tells whether the stack contains any program identifiers.

procedure push(id: identifier)

Puts an identifier on top of the stack. It has no effect if the stack is full.

procedure pop(var line, result: univ integer)

Removes a program identifier from the top of the stack and defines the line number at which the program terminated as well as its result. The result either indicates normal termination or one of several run-time errors as explained in the Concurrent Pascal report (Brinch Hansen 1975b).

procedure get(var id: identifier)

Defines the identifier stored in the top of the stack (without removing it). It has no effect if the stack is empty.

Implementation:

A program stack measures the extent of the heap of the calling process before pushing an identifier on the stack. If a pop operation shows abnormal program termination, the heap is reset to its original point to prevent the calling process from crashing due to lack of data space.

The standard routines, *attribute* and *setheap*, are defined in the Concurrent Pascal report.

```
type resulttype =
  (terminated, overflow, pointererror, rangeerror, varianterror,
  heaplimit, stacklimit, codelimit, timelimit, callerror);

type attrindex =
  (caller, heaptop, progline, progresult, runtime);

type progstack =
monitor

const stacklength = 5;

var stack:
  array [1..stacklength] of
    record progid: identifier; heapaddr: integer end;
  top: 0..stacklength;

function entry space: boolean;
begin space := (top < stacklength) end;

function entry any: boolean;
begin any := (top > 0) end;

procedure entry push(id: identifier);
begin
  if top < stacklength then
    begin
      top := top + 1;
      with stack[top] do
        begin
          progid := id;
          heapaddr := attribute(heaptop);
        end;
    end;
end;

procedure entry pop(var line, result: univ integer);
const terminated = 0;
begin
  line := attribute(progline);
  result := attribute(progresult);
  if result <> terminated then
    setheap(stack[top].heapaddr);
  top := top − 1;
end;
```

```
procedure entry get(var id: identifier);
begin
  if top > 0 then id := stack[top].progid;
end;

begin top := 0 end;
```

PAGE BUFFER

type pagebuffer = monitor

A page buffer transmits a sequence of data pages from one process to another. Each sequence is terminated by an end of file mark.

procedure read(var text: page; var eof: boolean)

Receives a message consisting of a text page and an end of file indication.

procedure write(text: page; eof: boolean)

Sends a message consisting of a text page and an end of file indication.

If the end of file is true then the text page is empty.

Implementation:

A page buffer stores a single message at a time. It will delay the sending process as long as the buffer is full and the receiving process until it becomes full ($0 \leq \text{writes} - \text{reads} \leq 1$).

```
type pagebuffer =
monitor

var buffer: page; last, full: boolean;
  sender, receiver: queue;

procedure entry read(var text: page; var eof: boolean);
begin
  if not full then delay(receiver);
  text := buffer; eof := last; full := false;
  continue(sender);
```

```
end;

procedure entry write(text: page; eof: boolean);
begin;
  if full then delay(sender);
  buffer := text; last := eof; full := true;
  continue(receiver);
end;

begin full := false end;
```

Solo also implements buffers for transmission of arguments (enumerations and identifiers) and lines. They are similar to the page buffer (but use no end of file marks). The need to duplicate routines for each message type is an inconvenience caused by the fixed data types of Pascal.

CHARACTER STREAM

type charstream = class(buffer: pagebuffer)

A character stream enables a process to communicate with another process character by character. A character stream uses a page buffer to transmit one page of characters at a time from one process to another.

A sending process must open its stream for writing before using it. The last character transmitted in a sequence should be an end of medium (em).

A receiving process must open its stream for reading before using it.

procedure initread

Opens a character stream for reading.

procedure initwrite

Opens a character stream for writing.

procedure read(var c: char)

Reads the next character from the stream. The effect is undefined if the stream is not open for reading.

procedure write(c: char)

Writes the next character in the stream. The effect is undefined if the stream is not open for writing.

Implementation:

```
type charstream =
class(buffer: pagebuffer);

var text: page; count: integer; eof: boolean;

procedure entry read(var c: char);
begin
  if count = pagelength then
    begin
      buffer.read(text, eof);
      count := 0;
    end;
  count := count + 1;
  c := text[count];
  if c = em then
    begin
      while not eof do buffer.read(text, eof);
      count := pagelength;
    end;
end;

procedure entry initread;
begin count := pagelength end;

procedure entry write(c: char);
begin
  count := count + 1;
  text[count] := c;
  if (count = pagelength) or (c = em) then
    begin
      buffer.write(text, false); count := 0;
      if c = em then buffer.write(text, true);
    end;
end;

procedure entry initwrite;
begin count := 0 end;

begin end;
```

TASKS AND ARGUMENTS

The following data types are used by several processes:

```
type taskkind = (inputtask, jobtask, outputtask);

type argtag = (niltype, booltype, inttype, idtype, ptrtype);
  argtype = record tag: argtag; arg: identifier end;

const maxarg = 10;
type arglist = array [1..maxarg] of argtype;

type argseq = (inp, out);
```

The *task kind* defines whether a process is performing an input task, a job task, or an output task. It is used by sequential programs to determine whether they have been loaded by the right kind of process. As an example, a program that controls card reader input can only be called by an input process.

A process that executes a sequential program can pass a list of arguments to it. A program *argument* consists of a tag field defining its type (boolean, integer, identifier, or pointer) and another field defining its value. (Since Concurrent Pascal does not include the variant records of Sequential Pascal one can only represent a program argument by the largest one of its variants—an identifier.)

A job process is connected to two input and output processes by *argument buffers* called its input and output sequences.

JOB PROCESS

```
type jobprocess =
process
  (typeuse: typeresource; diskuse: resource;
  catalog: diskcatalog; inbuffer, outbuffer: pagebuffer;
  inrequest, inresponse, outrequest, outresponse: argbuffer;
  stack: progstack)
"program data space" +16000
```

A job process executes Sequential Pascal programs that can call one another recursively. Initially, it executes a program called *do* with console input. A job process also implements the interface between sequential programs and the Solo operating system as defined in Brinch Hansen (1976b).

A job process needs access to the operator's console, the disk, and its catalog. It is connected to an input and an output process by two page buffers and four argument buffers as explained in Brinch Hansen (1976a). It uses a program stack to handle nested calls of sequential programs.

It reserves a data space of 16,000 bytes for user programs and a code space of 20,000 bytes. This enables the Pascal compiler to compile itself.

Implementation:

The private variables of a job process give it access to a terminal stream, two character streams for input and output, and two data files. It uses a large program file to store the currently executed program. These variables are inaccessible to other processes.

The job process contains a declaration of a sequential program that defines the types of its arguments and the variable in which its code is stored (the latter is inaccessible to the program). It also defines a list of interface routines that can be called by a program. These routines are implemented within the job process. They are defined in Brinch Hansen (1976b).

Before a job process can call a sequential program it must load it from disk into a program store and push its identifier onto a program stack. After termination of the program, the job process pops its identifier, line number, and result from the program stack, reloads the previous program from disk and returns to it.

A process can only interact with other processes by calling routines within monitors that are passed as parameters to it during initialization (such as the catalog declared at the beginning of a job process). These access rights are checked at compile-time (Brinch Hansen 1975a).

```
type jobprocess =
process
  (typeuse: typeresource; diskuse: resource;
  catalog: diskcatalog; inbuffer, outbuffer: pagebuffer;
  inrequest, inresponse, outrequest, outresponse: argbuffer;
  stack: progstack);
"program data space" +16000

const maxfile = 2;
type file = 1..maxfile;

var operator: terminal; opstream: terminalstream;
  instream, outstream: charstream;
  files: array [file] of datafile;
  code: progfile1;

program job(var param: arglist; store: progstore1);
entry read, write, open, close, get, put, length,
  mark, release, identify, accept, display, readpage,
  writepage, readline, writeline, readarg, writearg,
  lookup, iotransfer, iomove, task, run;

procedure call(id: identifier; var param: arglist;
  var line: integer; var result: resulttype);
var state: progstate; lastid: identifier;
begin
  with code, stack do
    begin
      line := 0;
      open(id, state);
      if (state = ready) & space then
        begin
          push(id);
          job(param, store);
          pop(line, result);
        end
```

```
         else if state = toobig then result := codelimit
         else result := callerror;
         if any then
            begin get(lastid); open(lastid, state) end;
      end;
end;

procedure entry read(var c: char);
begin instream.read(c) end;

procedure entry write(c: char);
begin outstream.write(c) end;

procedure entry open(f: file; id: identifier; var found: boolean);
begin files[f].open(id, found) end;

procedure entry close(f: file);
begin files[f].close end;

procedure entry get(f: file; p: integer; var block: page);
begin files[f].read(p, block) end;

procedure entry put(f: file; p: integer; var block: page);
begin files[f].write(p, block) end;

function entry length(f: file): integer;
begin length := files[f].length end;

procedure entry mark(var top: integer);
begin top := attribute(heaptop) end;

procedure entry release(top: integer);
begin setheap(top) end;

procedure entry identify(header: line);
begin opstream.reset(header) end;

procedure entry accept(var c: char);
```

```
begin opstream.read(c) end;

procedure entry display(c: char);
begin opstream.write(c) end;

procedure entry readpage(var block: page; var eof: boolean);
begin inbuffer.read(block, eof) end;

procedure entry writepage(block: page; eof: boolean);
begin outbuffer.write(block, eof) end;

procedure entry readline(var text: line);
begin end;

procedure entry writeline(text: line);
begin end;

procedure entry readarg(s: argseq; var arg: argtype);
begin
  if s = inp then inresponse.read(arg)
  else outresponse.read(arg);
end;

procedure entry writearg(s: argseq; arg: argtype);
begin
  if s = inp then inrequest.write(arg)
  else outrequest.write(arg);
end;

procedure entry lookup(id: identifier;
  var attr: fileattr; var found: boolean);
begin catalog.lookup(id, attr, found) end;

procedure entry iotransfer(device: iodevice;
  var param: ioparam; var block: page);
begin
  if device = diskdevice then
    begin
```

```
      diskuse.request;
      io(block, param, device);
      diskuse.release;
    end
  else io(block, param, device);
end;

procedure entry iomove(device: iodevice; var param: ioparam);
begin io(param, param, device) end;

function entry task: taskkind;
begin task := jobtask end;

procedure entry run(id: identifier; var param: arglist;
  var line: integer; var result: resulttype);
begin call(id, param, line, result) end;

procedure initialize;
var i: integer; param: arglist; line: integer; result: resulttype;
begin
  init operator(typeuse), opstream(operator),
    instream(inbuffer), outstream(outbuffer);
  instream.initread; outstream.initwrite;
  for i := 1 to maxfile do
    init files[i](typeuse, diskuse, catalog);
  init code(typeuse, diskuse, catalog);
  with param[2] do
    begin tag := idtype; arg := 'console     ' end;
  call( 'do           ', param, line, result);
  operator.write('jobprocess:(:10:)', 'terminated (:10)');
end;

begin initialize end;
```

IO PROCESS

type ioprocess =
process
(typeuse: typeresource; diskuse: resource;
catalog: diskcatalog; slowio: linebuffer;
buffer: pagebuffer; request, response: argbuffer;
stack: progstack; iotask: taskkind)
"program data space" +2000

An io process executes Sequential Pascal programs that produce or consume data for a job process. It also implements the interface between these programs and the Solo operating system.

An io process needs access to the operator, the disk, and the catalog. It is connected to a card reader (or a line printer) by a line buffer and to a job process by a page buffer and two argument buffers. It uses a program stack to handle nested calls of sequential programs.

It reserves a data space of 2,000 bytes for input/output programs and a code space of 4,000 bytes.

Initially, it executes a program called *io*

Implementation:

The implementation details are similar to a job process.

```
type ioprocess =
process
  (typeuse: typeresource; diskuse: resource;
  catalog: diskcatalog; slowio: linebuffer;
  buffer: pagebuffer; request, response: argbuffer;
  stack: progstack; iotask: taskkind);
"program data space" +2000

type file = 1..1;

var operator: terminal; opstream: terminalstream;
  iostream: charstream; iofile: datafile;
  code: progfile2;

program driver(var param: arglist; store: progstore2);
```

```
entry read, write, open, close, get, put, length,
  mark, release, identify, accept, display, readpage,
  writepage, readline, writeline, readarg, writearg,
  lookup, iotransfer, iomove, task, run;

procedure call(id: identifier; var param: arglist;
  var line: integer; var result: resulttype);
var state: progstate; lastid: identifier;
begin
  with code, stack do
    begin
      line := 0;
      open(id, state);
      if (state = ready) & space then
        begin
          push(id);
          driver(param, store);
          pop(line, result);
        end
      else if state = toobig then result := codelimit
      else result := callerror;
      if any then
        begin get(lastid); open(lastid, state) end;
    end;
end;

procedure entry read(var c: char);
begin iostream.read(c) end;

procedure entry write(c: char);
begin iostream.write(c) end;

procedure entry open(f: file; id: identifier; var found: boolean);
begin iofile.open(id, found) end;

procedure entry close(f: file);
begin iofile.close end;
```

```
procedure entry get(f: file; p: integer; var block: page);
begin iofile.read(p, block) end;

procedure entry put(f: file; p: integer; var block: page);
begin iofile.write(p, block) end;

function entry length(f: file): integer;
begin length := iofile.length end;

procedure entry mark(var top: integer);
begin top := attribute(heaptop) end;

procedure entry release(top: integer);
begin setheap(top) end;

procedure entry identify(header: line);
begin opstream.reset(header) end;

procedure entry accept(var c: char);
begin opstream.read(c) end;

procedure entry display(c: char);
begin opstream.write(c) end;

procedure entry readpage(var block: page; var eof: boolean);
begin buffer.read(block, eof) end;

procedure entry writepage(block: page; eof: boolean);
begin buffer.write(block, eof) end;

procedure entry readline(var text: line);
begin slowio.read(text) end;

procedure entry writeline(text: line);
begin slowio.write(text) end;

procedure entry readarg(s: argseq; var arg: argtype);
begin request.read(arg) end;
```

```
procedure entry writearg(s: argseq; arg: argtype);
begin response.write(arg) end;

procedure entry lookup(id: identifier;
  var attr: fileattr; var found: boolean);
begin catalog.lookup(id, attr, found) end;

procedure entry iotransfer(device: iodevice;
  var param: ioparam; var block: page);
begin
  if device = diskdevice then
    begin
      diskuse.request;
      io(block, param, device);
      diskuse.release;
    end
  else io(block, param, device);
end;

procedure entry iomove(device: iodevice; var param: ioparam);
begin io(param, param, device) end;

function entry task: taskkind;
begin task := iotask end;

procedure entry run(id: identifier; var param: arglist;
  var line: integer; var result: resulttype);
begin call(id, param, line, result) end;

procedure initialize;
var param: arglist; line: integer; result: resulttype;
begin
  init operator(typeuse), opstream(operator),
    iostream(buffer), iofile(typeuse, diskuse, catalog),
    code(typeuse, diskuse, catalog);
  if iotask = inputtask then iostream.initwrite
  else iostream.initread;
```

```
    call( 'io            ', param, line, result);
    operator.write('ioprocess:(:10:)', 'terminated (:10)');
  end;

  begin initialize end;
```

CARD PROCESS

type cardprocess =
process(typeuse: typeresource; buffer: linebuffer)

A card process transmits cards from a card reader through a line buffer to an input process. The card process can access the operator to report device failure and a line buffer to transmit data. It is assumed that the card reader is controlled by a single card process. As long as the card reader is turned off or is empty the card process waits. It begins to read cards as soon as they are available in the reader. After a transmission error the card process writes a message to the operator and continues the input of cards.

Implementation:

The standard procedure *wait* delays the card process one second (Brinch Hansen 1975b). This reduces the processor time spent waiting for operator intervention.

```
type cardprocess =
process(typeuse: typeresource; buffer: linebuffer);

var operator: terminal; param: ioparam;
  text: line; ok: boolean;
begin
  init operator(typeuse);
  param.operation := input;
  cycle
    repeat
      io(text, param, carddevice);
      case param.status of
        complete:
          ok := true;
        intervention:
              begin ok := false; wait end;
            transmission, failure:
              begin
                operator.write('cards:(:10:)', 'error(:10:)');
                ok := false;
              end
          end
        until ok;
        buffer.write(text);
      end;
    end;
```

PRINTER PROCESS

type printerprocess =
process(typeuse: typeresource; buffer: linebuffer)

A printer process transmits lines from an output process to a line printer. The printer process can access the operator to report device failure and a line buffer to receive data. It is assumed that the line printer is controlled only by a single printer process. After a printer failure the printer process writes a message to the operator and repeats the output of the current line until it is successful.

Implementation:

```
type printerprocess =
process(typeuse: typeresource; buffer: linebuffer);

var operator: terminal; param: ioparam; text: line;
begin
  init operator(typeuse);
  param.operation := output;
  cycle
    buffer.read(text);
    io(text, param, printdevice);
    if param.status <> complete then
      begin
        operator.write('printer:(:10:)', 'inspect(:10:)');
        repeat
          wait;
          io(text, param, printdevice);
        until param.status = complete;
      end;
  end;
end;
```

LOADER PROCESS

type loaderprocess =
process(diskuse: resource)

A loader process preempts the operating system and reinitializes it when the operator pushes the *bell* key (*control g*) on the console. A loader process needs access to the disk to be able to reload the system.

Implementation:

A control operation on the typewriter delays the loader process until the operator pushes the bell key (Brinch Hansen 1975b).

The operating system is stored on consecutive disk pages starting at the *Solo address*. It is loaded by means of a control operation on the disk as defined in Brinch Hansen (1975b). Consecutive disk pages are used to make the system kernel of Concurrent Pascal unaware of the structure of a particular filing system (such as the one used by Solo). The disk contains a sequential program *start* that can copy the Solo system from a concurrent code file into the consecutive disk segment defined above.

```
type loaderprocess =
process(diskuse: resource);

const soloaddr = 24;
var param: ioparam;

procedure initialize(pageno: univ ioarg);
begin
  with param do
    begin
      operation := control;
```

```
        arg := pageno;
      end;
  end;

  begin
    initialize(soloaddr);
    "await bel signal"
    io(param, param, typedevice);
    "reload solo system"
    diskuse.request;
    io(param, param, diskdevice);
    diskuse.release;
  end;
```

INITIAL PROCESS

The initial process initializes all other processes and monitors and defines their access rights to one another. After initialization the operating system consists of a fixed set of components: a card process, an input process, a job process, an output process, a printer process, and a loader process. They have access to an operator, a disk, and a catalog of files. Process communication takes place by means of two page buffers, two line buffers and four argument buffers (see also Fig. 1).

Implementation:

When a process, such as the initial process, terminates its execution, its variables continue to exist (because they may be used by other processes).

```
  var
    typeuse: typeresource;
    diskuse: resource; catalog: diskcatalog;
    inbuffer, outbuffer: pagebuffer;
    cardbuffer, printerbuffer: linebuffer;
    inrequest, inresponse, outrequest, outresponse: argbuffer;
    instack, outstack, jobstack: progstack;
    reader: cardprocess; writer: printerprocess;
    producer, consumer: ioprocess; master: jobprocess;
    watchdog: loaderprocess;
  begin
  init
    typeuse, diskuse,
    catalog(typeuse, diskuse, cataddr),
    inbuffer, outbuffer,
    cardbuffer, printerbuffer,
    inrequest, inresponse, outrequest, outresponse,
    instack, outstack, jobstack,
    reader(typeuse, cardbuffer),
    writer(typeuse, printerbuffer),
    producer(typeuse, diskuse, catalog, cardbuffer,
      inbuffer, inrequest, inresponse, instack, inputtask),
    consumer(typeuse, diskuse, catalog, printerbuffer),
      outbuffer, outrequest, outresponse, outstack, outputtask),
    master(typeuse, diskuse, catalog, inbuffer, outbuffer,
      inrequest, inresponse, outrequest, outresponse,
      jobstack),
    watchdog(diskuse);
end;
```

CONCLUSION

The Solo system consists of 22 line printer pages of Concurrent Pascal text divided into 23 component types (10 classes, 7 monitors, and 6 processes). A typical component is less than one page long and can be studied in isolation as an (almost) independent piece of program. All program components called by a given component are explicitly declared within that component (either as permanent variables or a parameters to it). To understand a component it is only necessary to know *what* other components called by it do, but *how* they do it is irrelevant.

The entire system can be studied component by component as one would read a book. In that sense, Concurrent Pascal supports *abstraction* and *hierarchical structuring* of concurrent programs very nicely.

It took 4 compilations to remove the formal programming errors from the Solo system. It was then tested systematically from the bottom up by adding one component type at a time and trying it by means of short test processes. The whole program was tested in 27 runs (or about 1 run per component type). This revealed 7 errors in the test processes and 2 trivial ones in the system itself. Later, about one third of it was rewritten to speed up program loading. This took about one week. It was then compiled and put into operation in one day and has worked ever since.

I can only suggest two plausible explanations for this unusual testing experience. It seems to be vital that the compiler prevents new components from destroying old ones (since old components cannot call new ones, and new ones can only call old ones through routines that have already been tested). This strict checking of hierarchical access rights makes it possible for a large system to evolve gradually through a sequence of intermediate, stable subsystems.

I am also convinced now that the use of abstract data types which hide implementation details within a fixed set of routines encourages a clarity of design that makes programs practically correct before they are even tested. The slight inconvenience of strict type checking is of minor importance compared to the advantages of instant program reliability.

Although Solo is a small concurrent program of only 1,300 lines it does implement a virtual machine that is very convenient to use for program development (Brinch Hansen 1976a). The availability of cheap microprocessors will put increasing pressure on software designers to develop special-purpose operating systems at very low cost. Concurrent Pascal is one example of a programming tool that may make this possible.

P. Brinch Hansen 1973. *Operating System Principles*, Chapter 7, Resource Protection. Prentice-Hall, Englewood Cliffs, NJ.

P. Brinch Hansen 1975a. The programming language Concurrent Pascal. *IEEE Trans. on Software Engineering*, **1**, 2.

P. Brinch Hansen 1975b. *Concurrent Pascal Report.* Information Science, California Institute of Technology, (June).

P. Brinch Hansen 1976a. The Solo operating system: a Concurrent Pascal program. *Software—Practice and Experience*, **6**, 2 (April–June).

P. Brinch Hansen 1976b. The Solo operating system: job interface. *Software—Practice and Experience*, **6**, 2 (April–June),.

Acknowledgements

The development of Concurrent Pascal and Solo has been supported by the National Science Foundation under grant number DCR74–17331.