PER BRINCH HANSEN

Information Science
California Institute of Technology

October 1975

CONCURRENT PASCAL MACHINE

CONCURRENT PASCAL MACHINE: STORE ALLOCATION

Per Brinch Hansen

Information Science
California Institute of Technology
Pasadena, California 91125

October 1975

Summary

This describes the allocation of core store among the classes, monitors, and processes of a Concurrent Pascal program running on a PDP 11/45 computer.

Key Words and Phrases: Concurrent Pascal implementation, Virtual machine, Store allocation, PDP 11/45 computer.

CR Categories: 4.2, 6.2

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

CONCURRENT PASCAL MACHINE

Per Brinch Hansen

Information Science
California Institute of Technology
Pasadena, California 91125

October 1975

This is a description of the virtual machine that executes Concurrent Pascal programs on the PDP 11/45 computer. It contains four papers entitled:

1. Concurrent Pascal machine: Store allocation
2. Concurrent Pascal machine: Code interpretation
3. Concurrent Pascal machine: Kernel
4. The Concurrent Pascal compiler

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

CONTENTS

1. INTRODUCTION

Concurrent Pascal is a programming language for structured programming of computer operating systems [1, 2]. The Concurrent Pascal compiler generates code for a virtual machine that can be simulated by microprogram or machine code on different computers [3].

This paper describes the allocation of core store among the processes of a Concurrent Pascal program running on a PDP 11/45 computer. The interpreter of the virtual code and the kernel that controls processor multiplexing are discussed in [4, 5].

2. CORE STORE

A Concurrent Pascal program defines a fixed number of processes. Figure 1 shows the core store during the execution of a Concurrent Pascal program. It contains code and data segments. The lengths of these are fixed at compile-time.

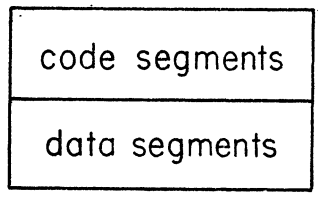

Fig.1. Core store

The code segments consist of virtual code generated by the Concurrent Pascal compiler, an interpreter that executes the virtual code, and a kernel that schedules the execution of concurrent processes (Fig. 2).

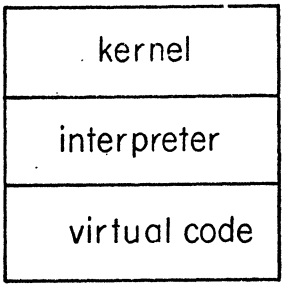

Fig. 2. Code segments

The interpreter and kernel are assembly language programs that implement the virtual machine. These two programs of 1 and 3 K words are loaded from disk into core by means of the operator's control panel. They in turn load the virtual code of a Concurrent Pascal program into core and start executing it as a single process, called the initial process. The latter can now create a fixed number of child processes. The kernel multiplexes the processor among these processes.

Each process has a data segment in core (Fig. 3). Data segments have fixed lengths determined during compilation. They exist forever during execution. This makes store allocation trivial: segments are allocated contiguously in their order of creation.

The segment length of the initial process and the start address of its code are defined at the beginning of the virtual code. The store requirements and code address of child processes are defined by initprocess instructions.

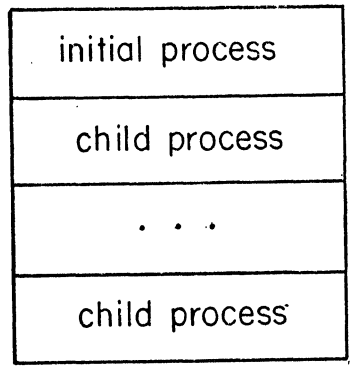

Fig. 3. Data segments

3. VIRTUAL STORE

On the PDP 11/45 computer, the storage space of a process consists of up to 8 segments of at most 4 K words each. These segments can be placed anywhere in core. An adressing mechanism makes them appear contiguous to the process.

This mechanism is not used by Concurrent Pascal to enforce access rights of processes. That is done at compile-time. It is just a (rather inconvenient) way of extending the addressing capability of a computer with a short word length by letting each process see part of a larger core store. It would be unnecessary on a machine that can address the whole store directly.

The virtual store of a process gives it access to a common segment shared by all processes and to its own data segment (called a private segment) (fig. 4.)

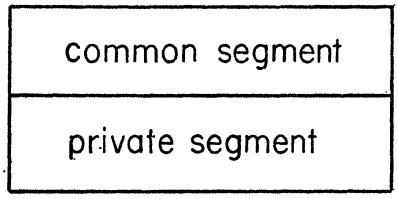

Fig. 4. Virtual store

The common segment consists of the interpreter, the virtual code, and the data segment of the initial process. The latter contains the monitors through which processes communicate (Fig. 5).

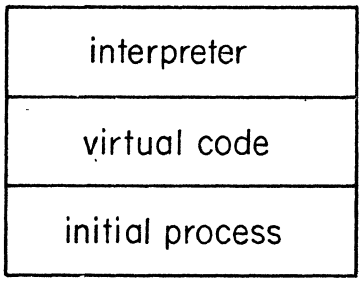

Fig. 5. Common segment

The initial process has no private data segment. Its data segment is included in the common segment.

4. DATA SEGMENTS

A data segment contains the stack and heap of a process (Figs. 3 and 6).

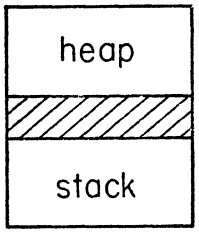

Fig.6. Data segment

The stack contains the permanent variables (and parameters) of a process as well as its temporary variables used within procedures (Fig. 7).

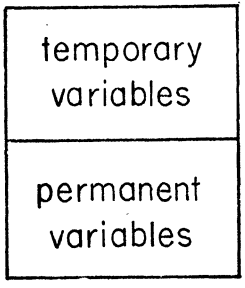

Fig. 7. Stack

The initial process is created by the kernel. It has no parameters. When a child process is created, its parameters are copied from the parent's stack (in the common data segment) into the child's stack (in a private segement).

The heap is only accessible to sequential Pascal programs executed by a Concurrent Pascal program.

5. PERMANENT VARIABLES

Figure 8 shows the representation of the permanent variables and parameters of a class, monitor, or process.

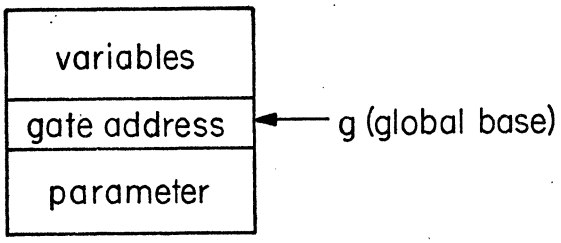

Fig. 8. Permanent variables

A monitor contains an address of a data structure called a gate. The gate is stored in the kernel. It is used to give a process exclusive access to the monitor [5]. (The gate address has no significance for classes and processes.)

At any time, a process can only operate on a single set of permanent variables. They are addressed relative to a global base address g. When a process is created its global base register points to its own permanent variables. When it calls a monitor (or class) procedure the current base address is pushed onto its stack, and the global base register is used to point to the permanent variables of that monitor (or class). Upon return from the procedure the previous base address is popped from the stack.

6. TEMPORARY VARIABLES

Figure 9 shows the representation of the parameters, variables, and temporaries of a procedure call. A dynamic link connects the procedure to the context in which it was called.

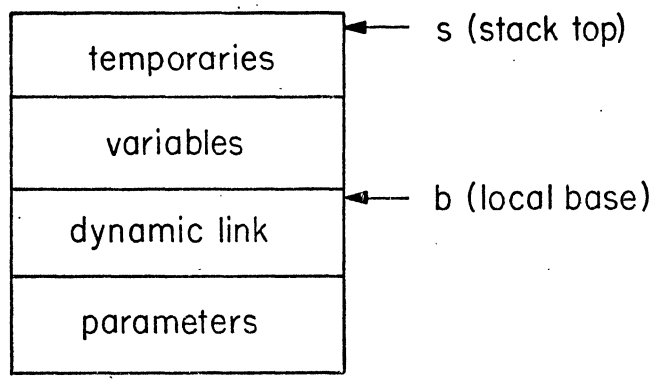

Fig. 9. Temporary variables

At any time, a process can only operate on a single set of temporary variables (and parameters). They are addressed relative to a local base address b. Temporaries are addressed relative to a stack top s.

The dynamic link defines the stack addresses g, b, and s used by a process before a procedure call and a return address q in the virtual code. The link also contains the current line number within the procedure to facilitate location of run-time errors.

When a process is created its global and local base registers both point to the permanent variables of that process. It is initialized with no temporaries and an empty heap.

When a process calls one of its own procedures, the local base register will point to the temporary variables of that procedure. Its global base address remains unchanged.

When a process calls a monitor (or class) procedure, the global base register will point to the permanent variables of that monitor (or class), and the local base register will point to the temporary variables of the monitor (or class) procedure.

Upon return from a procedure its temporary variables are popped from the stack and the previous values of the base registers are reestablished by means of the dynamic link.

7. COMPROMISES

In implementing Concurrent Pascal we followed one simple guideline: A computer should only do obvious things and should do them well. Where compromise was needed we firmly put simplicity first, efficiency second, and generality third. Like any other design rule it needs no justification other than the success that it leads to in practice.

It takes good nerves to follow this advice on a machine that invites the software designer to optimize register usage and use sliding addressing windows. We decided to simplify code generation by ignoring the instruction set and different registers of the PDP 11/45 and simulate a simple stack machine.

The virtual addressing mechanism is more difficult to ignore since it determines the amount of core store that can be used by a Concurrent Pascal program. The virtual store of the PDP 11/45 consists of two address spaces: one for machine code and another for data. Since the only machine code executed by a process is an interpreter of 1 K words, it is not worth keeping it in a separate address space. So we let the two address spaces be identical.

One of the main achievements of Concurrent Pascal is to make it possible to check the access rights of processes at compile-time. This makes it possible to make monitor calls almost as fast as simple procedure calls. To gain this efficiency, the virtual code and data of monitors were included in the address space of every process (otherwise, it would have been necessary to change address spaces and copy parameters during monitor calls).

However, by putting simplicity and efficiency first, we have undoubtedly lost generality: A process must divide its address space of 32 K words between its private data and the code and common data of all processes. To avoid fragmentation of the virtual address space, processes have only a single segment in common. This is achieved by the following language restriction: only the initial process can create other processes and give them access to common data [2].

Segmentation of address space can be helpful when it supports the scope rules of a high-level language by associating data segments with procedures and classes. But when it arbitrarily cuts physical store into eight parts, segmentation becomes an obstacle to straightforward language implementation.

Acknowledgement

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

References

1. Brinch Hansen, P. The programming language Concurrent Pascal. IEEE Transactions on Software Engineering 1, 2 (June 1975), 199-207.

2. Brinch Hansen, P. Concurrent Pascal report. Information Science, California Institute of Technology, June 1975.

3. Hartmann, A.C. A Concurrent Pascal compiler for minicomputers. (Ph.D. Thesis). Information Science, California Institute of Technology, Sept. 1975.

4. Brinch Hansen, P. Concurrent Pascal machine: Code interpretation. Information Science, California Institute of Technology, Oct. 1975.

5. Brinch Hansen, P. Concurrent Pascal machine: Kernel. Information Science, California Institute of Technology, Oct. 1975.

CONCURRENT PASCAL MACHINE: CODE INTERPRETATION

Per Brinch Hansen

Information Science
California Institute of Technology
Pasadena, California 91125

October 1975

Summary

This describes the interpretation on the PDP 11/45 computer of virtual code generated by the Concurrent Pascal compiler.

Key Words and Phrases: Concurrent Pascal implementation, Virtual machine, Code interpretation, PDP 11/45 computer.

CR Categories: 4.2, 6.2

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

CONTENTS

1. INTRODUCTION

The Concurrent Pascal compiler generates code for a virtual machine [1, 2, 3]. At Caltech, the virtual machine is simulated by code on a PDP 11/45 computer. The processor is multiplexed among a fixed number of processes. Each of these has a stack of fixed (maximum) length [4, 5]. This describes the virtual code which is similar to the one used by Wirth's group for sequential Pascal [6]. The programming technique used to interpret the virtual code is called threaded code [7].

The use of virtual code designed directly to support a high-level language makes code generation straightforward and the compiler portable. (Our sequential Pascal compiler was moved to another minicomputer in one man-month.)

2. VIRTUAL CODE

We will use a programming example to illustrate the virtual code: a monitor that defines a send operation on a message buffer:

```
type page = array (.1..length.) of integer;
     buffer = monitor
              var contents: page; empty: boolean;
                  sender, receiver: queue;

              procedure entry send(message: page);
              begin
                if not empty then delay(sender);
                contents:= message;
                empty:= false;
                continue(receiver);
              end;

              .....
              end;
```

(The rest of the monitor can be ignored here.)

The virtual code generated for the send procedure is:

```
     entermonitor(stacklength, paramlength, linenumber, varlength)
     pushglobal(empty)
     not
     falsejump(L)
     globaladdr(sender)
     delay
L:   globaladdr(contents)
     pushlocal(message)
     copystructure(length)
     globaladdr(empty)
     pushconst(false)
     copyword
     globaladdr(receiver)
     continue
     exitmonitor
```

The enter monitor instruction defines the total amount of stack space needed by the procedure, the length of its parameters and local variables, and the number of the program line on which it begins.

The next instruction pushes the global variable empty onto the stack. The program then performs a not operation on it, and jumps to label L if the result is false; otherwise, it pushes the address of the global variable sender on the stack and performs a delay operation on it.

After this, the address of the buffer contents and the message are pushed onto the stack. (The message parameter is represented by a local variable that contains a reference to the actual argument.) A copy structure instruction moves the message into the buffer.

This is followed by an assignment of the constant false to the global variable empty. The procedure ends with a continue operation on the global variable receiver followed by an exit monitor instruction.

An instance of a buffer monitor can be declared and used as follows:

```
var channel: buffer; data: page;
..... channel.send(data); .....
```

This monitor call generates the following virtual code:

```
globaladdr(channel)
field(varlength)
globaladdr(data)
call(send)
```

The base address of the global variable channel is pushed onto the stack and incremented by a field instruction (to make it point to the gate address that separates the permanent variables of the monitors from its parameters - See [4], Fig. 8). Then the address of the global variable data is pushed onto the stack, and the monitor procedure send is called.

Variables are identified by their displacements relative to a local or global base address [4]. Program labels are represented by their displacements relative to a virtual program counter (making the virtual code relocatable).

There are about 50 different virtual instructions. To make the software interpreter fast, addressing modes (local or global) and data types (bytes, words, reals, or sets) are encoded into the operation codes. This expands the set of operation codes to 110. A quarter of these are used by Concurrent Pascal only. The rest are common to sequential and Concurrent Pascal.

This description only tries to explain the overall structure of the virtual code and its interpreter. The interpreter listing contains a complete definition of all virtual instructions. It is on the distribution tape of the Solo operating system [8]. The language constructs of Concurrent Pascal and the corresponding virtual code are defined by syntax graphs in [3].

3. THE INTERPRETER

The interpreter is an assembly language program of 1 K words. It consists of code pieces that execute virtual instructions and an operation table defining the location of these pieces (Fig. 1).

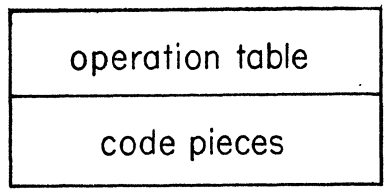

Fig. 1. Interpreter

A virtual instruction consists of an operation possibly followed by some arguments. Operations and arguments occupy one machine word each. The interpreter uses a virtual instruction counter q to point to the next operation or one of its arguments.

As an example, the virtual instructions:

```
pushconst(false)
copyword
```

are represented by three machine words:

```
pushconst
false
copyword
```

Upon entry to the push constant code in the interpreter, the virtual instruction counter q points to the argument of that instruction (the boolean value false). The interpreter executes push constant as follows:

```
s :- 2; store(s) := store(q); q :+ 2;
```

First, the stack top s is decremented by one word (2 bytes).*) Then the argument is moved from its virtual code location store(q) to the new stack location store(s). Finally, the virtual instruction counter q is incremented by one word (2 bytes). All this is done by a single machine instruction on the PDP 11/45.

The virtual instruction counter now points to the next virtual instruction copy word. The interpreter uses the operation code store(q) as an index in the operation table (beginning at address zero) to jump to the corresponding code piece:

```
goto store(store(q)); q :+ 2;
```

This is also done by a single machine instruction.

Every code piece of the interpreter ends with such a jump to its successor. These three store cycles are the only overhead of interpretation compared to directly executable code. This efficient form of interpretation is called threaded code [7]. Execution times for the virtual code on the PDP 11/45 computer are included in the Concurrent Pascal Report [2].

4. REGISTERS

The interpreter uses nine registers to execute the virtual code of a process. Three of these are scratch registers used during the execution of a single virtual instruction only. The rest have fixed functions throughout the execution of a process (Fig. 2).

*) The PDP 11 stack grows from high towards low addresses.

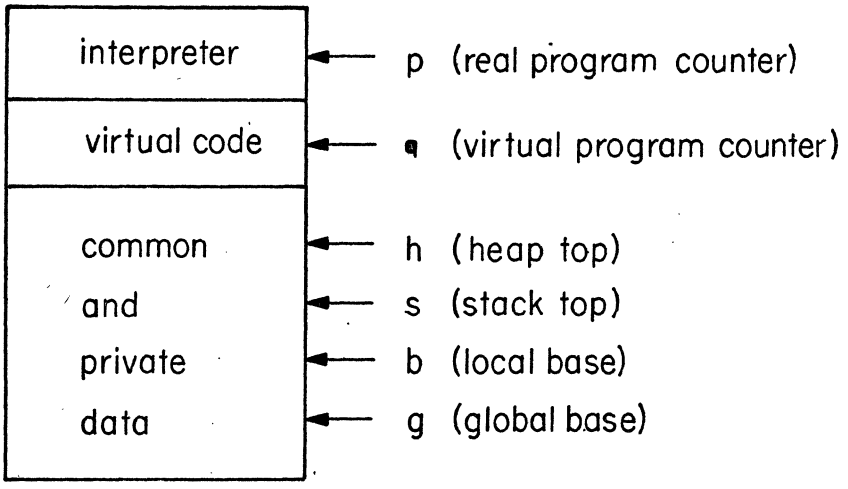

Fig. 2. Virtual store and register

The real program counter p remains within the interpreter. It uses a virtual program counter q to point to virtual instructions. The heap top h defines the current extent of the heap. (It is stored in a store location within the interpreter instead of a register.) The stack is addressed relative to three registers: a global base register g, a local base register b, and a stack top s as explained in [4].

Acknowledgement

The interpreter was programmed by Tom Zepko. The development of Concurrent Pascal has been supported by the National Science Foundation under grant number DCR74-17331.

References

1. Brinch Hansen, P. The programming language Concurrent Pascal. IEEE Transactions on Software Engineering 1, 2 (June 1975), 199-207.

2. Brinch Hansen, P. Concurrent Pascal report. Information Science, California Institute of Technology, June 1975.

3. Hartmann, A.C. A Concurrent Pascal compiler for minicomputers. (Ph.D. Thesis) Information Science, California Institute of Technology, Sept. 1975.

4. Brinch Hansen, P. Concurrent Pascal machine: Store allocation. Information Science, California Institute of Technology, Oct. 1975.

5. Brinch Hansen, P. Concurrent Pascal machine: Kernel. Information Science, California Institute of Technology, Oct. 1975.

6. Nori, K.V., et al. The Pascal P compiler: Implementation notes. Technical University, Zurich, Switzerland, Dec. 1974.

7. Bell, J.R. Threaded code. Comm. ACM 16, 6 (June 1973), 370-72.

8. Brinch Hansen, P. The Solo operating system. Information Science, California Institute of Technology, July 1975.

CONCURRENT PASCAL MACHINE KERNEL

Per Brinch Hansen

Information Science
California Institute of Technology
Pasadena, California 91125

October 1975

Summary

This describes the kernel of Concurrent Pascal on the PDP 11/45 computer. It controls processor multiplexing and scheduling of monitor calls.

Key Words and Phrases: Concurrent Pascal implementation, Virtual machine, Process and monitor implementation.

CR Categories: 4.22, 4.32

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

CONTENTS

1. INTRODUCTION

This describes the kernel of Concurrent Pascal - an assembly language program that multiplexes a PDP 11/45 processor among concurrent processes and gives them exclusive access to monitors [1-4].

The kernel was first written in a programming language that resembles Concurrent Pascal. It consists of a collection of data structures representing processes, monitors, and peripherals. Each data structure consists of two parts: one defines how the data is represented in store, the other what operations one can perform on the data. This combination of a data representation and the possible operations on it is usually called a class or an abstract data type. (It is abstract because other parts of the program can ignore the details of data representation and think of it solely in terms of the meaningful operations defined on it.)

The abstract version of the kernel was translated by hand into assembly language (retaining the abstract version as comments). This programming method has several advantages:

(1) A complex program can be programmed as a sequence of small, self-contained components (classes).

(2) These components can be tested one at a time from the bottom up.

(3) If the program only accesses a component through procedures (or macros) associated with it, new (untested) components cannot make old (tested) components fail.

(4) In the rare cases, where it is necessary to use assembly language, one can still use an abstract programming language as a thinking tool, and make the production of assembly code a simple clerical procedure (manual translation).

After an initial test period of one month the Concurrent Pascal kernel has been used without problems for nine months. So it seems that this form of programming could be called reliable machine programming.

The details of the kernel are simplified in the following (but most of the simplifications are pointed out). The distribution tape of the Solo operating system contains a complete kernel listing [5].

2. PROCESSOR MULTIPLEXING

The computer executes one process at a time. While one process is running, other processes must await their turn in a ready queue. Every 17 msec the computer switches from one process to another to give the illusion that they are executed simultaneously.

A process is represented by a record within the kernel. When a process is preempted all registers used to interpret its code are stored in its process record [4]. The register values are restored when the execution of the process is resumed:

```
type registers = record ... end;
     process = @ registers;
```

A process queue is represented by a sequence of references to process records:

```
type processqueue = sequence of process;
```

The only operations on a process queue are:

put	enters a process in the queue
get	removes a process from the queue
any	tells whether the queue contains anything
empty	tells whether the queue is empty

The running process is represented by a class:

Again, the picture is simplified: the clock will only preempt a process when it has used a reasonable amount of processor time; and it will never interrupt a process inside a monitor procedure.

The class running also contains procedures for process creation. After system loading, the kernel calls a procedure initparent that starts execution of the initial process:

```
procedure initparent(length: integer);
begin
  new(user);
  virtual.defcommon(length);
  initialize registers;
end;
```

A procedure new allocates space for a process record in a heap inside the kernel. A procedure defcommon within another class virtual is then called to define the location of the common segment used by the initial process and its descendants [3]. Finally, the registers are initialized to define the limits of the stack and the heap within the segment as well as the start address of the process code [4].

The initial process can, in turn, call a kernel procedure initchild to create other processes:

```
procedure initchild(length: integer);
begin
  ready.enter(preempted);
  new(user);
  virtual.defprivate(length);
  initialize registers;
end;
```

This is similar to the previous procedure, except that the parent is preempted in favor of its child. Again, details have been ignored, such as the accounting of processor time spent by processes.

When a process terminates its execution, it is preempted forever (but its data segment continues to exist):

```
procedure endprocess;
begin user:= nil end;
```

This leaves the processor idle upon exit from the kernel. To make it busy again, the following statement is always executed upon kernel exit:

```
if running.user = nil then ready.select;
```

3. MONITOR IMPLEMENTATION

Within the kernel, a monitor variable is represented by a data structure, called a gate, that only gives one process at a time access to the monitor:

```
type gate = class
            var open: boolean; waiting: processqueue;

            procedure enter;
            begin
              if open then open:= false
                      else waiting.put(running.preempted);
            end;

            procedure leave;
            begin
              if waiting.empty then open:= true
                               else ready.enter(waiting.get);
            end;

            procedure delay(var q: process);
            begin q:= running.preempted; leave end;

            procedure continue(var q: process);
            begin
              if q = nil then leave else
              begin ready.enter(q); q:= nil end;
            end;

            begin open:= false; waiting.initialize end;
```

A gate is represented by a boolean defining whether it is open, and a queue of processes waiting to enter it. Initially, the gate is open and nobody is waiting outside it.

At the beginning and end of a monitor procedure, a process executes an enter and a leave operation:

Enter: If the gate is open, the process enters and closes it; otherwise, it is preempted to wait outside the gate.

Leave: If nobody is waiting outside the gate, it is left open; otherwise, a single waiting process is resumed (by transferring it to the ready queue).

These are the short-term operations that force processes to enter a monitor one at a time. A monitor can also delay processes for longer periods of time and resume them again by means of delay and continue operations on single-process queues:

Delay: Preempts the running process and enters it in a given single-process queue. The monitor can now be entered by another process.

Continue: Forces the running process to leave the monitor and resumes any process that may be waiting in a given single-process queue.

Details ignored: When a process is resumed within a monitor it will preempt the running process (unless the latter is engaged in nested monitor calls).

When a monitor variable is initialized, the kernel executes a procedure that allocates its gate in the heap and initializes it:

```
procedure initgate(var g: @gate);
begin new(g); g@.initialize end;
```

The gate reference is stored in the stack of the calling process and passed as a parameter to the kernel each time one of the gate operations are executed [3].

4. PERIPHERALS

A peripheral device is represented by a class:

```
var peripheral: class(device: integer);
                var user: process;

                procedure start(operation: T);
                begin
                  startdevice(device, operation);
                  user:= running.preempted;
                end;

                procedure interrupt;
                begin
                  ready.enter(user);
                  user:= nil;
                end;

                begin user:= nil end;
```

The class defines the device number of the peripheral and its current user process. An io statement in Concurrent Pascal is translated into a call of a procedure that starts a data transfer and preempts the calling process. An interrupt resumes the user process.

Details: The interrupt procedure also returns a status word to the calling process and (usually) gives it priority over the running process.

Only one process at a time can use a peripheral. This must be guaranteed by the operating system written in Concurrent Pascal (and not by the kernel). The main function of the kernel is to make peripherals look uniform with respect to simple input/output operations and exception indications. It does not perform error recovery.

5. KERNEL CLASSES

The kernel consists of a hierarchy of classes (some of which have already been described):

Newcore	Allocates process records and gates in a heap.
Processqueue	Implements process queues.
Signal	Implements a queue in which processes can wait until a timing signal is sent.
Time	Keeps track of real time.
Timer	Measures time intervals.
Clock	Delays calling processes for one second.
Core	Allocates core store.
Virtual	Allocates virtual store.
Running	Creates, executes, and preempts processes.
Ready	Schedules processes for execution.
Gate	Gives processes exclusive access to monitors.
Peripherals	Handles simple input/output.

6. PROGRAMMING AND TESTING

The kernel was translated manually line by line into assembly language using the abstract program as comments. A small example is sufficient to illustrate this programming technique:

```
  gate: .word 1              ;  type gate =
                             ;  class
  open = 0                   ;  var open: boolean;
  wait = open + .boolean     ;      waiting: processqueue;
                             ;
  enter:                     ;  procedure enter;
    mov  gate, r0            ;  begin
    dec  (r0)                ;    if open
    beq  1$                  ;      then
    clr (r0)+                ;        open:= false
    mov  r0, procq           ;
    jsr  pc, preempt         ;
    mov  preval, elem        ;      else
    jsr  pc, put             ;        waiting.put(running.preempted);
1$: rts  pc                  ;  end;

  etc.
```

The kernel was tested, class by class, by means of test programs written in Concurrent Pascal:

Test 1:	Initialization and process creation
Test 2:	Clock interrupt and processor switching
Test 3-4:	Monitor gates
Test 5:	Teletype
Test 6:	Timer and clock
Test 7:	Teletype interrupt key

In test 1, clock interrupts were turned off. In tests 2-6 they were simulated manually by means of the Bell key on the teletype. Test 7 used normal clock interrupts. The only test output used was a message on the teletype every time a process arrives in a queue or departs from one. This technique for testing a multiprogramming system is explained in [6].

It took 10 test runs to make test 1 work (!) The rest of them required 18 runs altogether. Finally, the peripherals were tested by means of Concurrent Pascal programs in normal operation. After this initial testing the kernel seems to be correct.

7. SIZE AND PERFORMANCE

The classes of the kernel are of the following size:

	words:
Newcore	560
Processqueue	30
Signal	40
Time	20
Timer	10
Clock	60
Core	40
Virtual	160
Running	570
Ready	130
Gate	110
6 Peripherals	1020
Initialization	160
Kernel	2910

Newcore has room for 10 process records and 25 monitor gates. 60 per cent of running is process creation and termination. Each peripheral is controlled by about 150 words of code.

The most critical execution times are:

empty kernel call	20 usec
monitor call	200 usec
delay, continue	600 usec
clock interrupt	1000 usec

A monitor call causes the interpreter to call the kernel twice: at the beginning and end of the procedure. The 200 usec assumes that the process can enter the monitor immediately and continue its execution when it returns from it. This should be compared with the execution time of a simple procedure call (60 usec).

The figures for delay and continue (600 usec) illustrate the cost of switching the processor from one process to another.

Acknowledgement

The kernel was programmed in assembly language by Robert Deverill. The development of Concurrent Pascal has been supported by the National Science Foundation under grant number DCR74-17331.

References

1. Brinch Hansen, P. The programming language Concurrent Pascal. IEEE Transactions on Software Engineering 1, 2 (June 1975), 199-207.

2. Brinch Hansen, P. Concurrent Pascal report. Information Science, California Institute of Technology, June 1975.

3. Brinch Hansen, P. Concurrent Pascal machine: Store allocation. Information Science, California Institute of Technology, Oct. 1975.

4. Brinch Hansen, P. Concurrent Pascal machine: Code interpreter. Information Science, California Institute of Technology, Oct. 1975.

5. Brinch Hansen, P. The Solo operating system. Information Science, California Institute of Technology, July 1975.

6. Brinch Hansen, P. Testing a multiprogramming system. Software 3, 2 (April-June 1973), 145-50.

THE CONCURRENT PASCAL COMPILER

Per Brinch Hansen

Information Science
California Institute of Technology
Pasadena, California 91125

October 1975

Summary

This describes a portable seven-pass compiler for the programming language Concurrent Pascal. The compiler is written in sequential Pascal and requires 16 K words of core store. The paper summarizes the passes and their interface to an operating system. It also explains how the compiler was tested and how well it performs on the PDP 11/45 computer.

Key Words and Phrases: Concurrent Pascal compiler, Multipass compilation, Abstract data types, Scope analysis, Testing, Performance.

CR Categories: 4.12, 4.22

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

CONTENTS

1. INTRODUCTION

This describes a portable Concurrent Pascal compiler for minicomputers. Concurrent Pascal is a programming language that includes abstract data types for sequential and concurrent programming (classes, monitors, and processes) [1, 2].

The Concurrent Pascal compiler generates code for a virtual machine that can be simulated on different minicomputers by microprogram or machine code. At Caltech, it runs on a PDP 11/45 computer [3].

The compiler was designed and programmed by Alfred Hartmann and myself [4]. It is written in the programming language sequential Pascal [5]. Its structure is inspired by the Gier Algol and Siemens Cobol compilers [6, 7]. The compiler is divided into seven passes. The following describes the overall division of labor among the passes as well as their size and performance. The compiler is described in detail in [4].

2. MULTIPASS COMPILATION

Our goal was to make a compiler that can compile operating systems on a minicomputer with at least 16 K words of core store and a slow disk (50 msec/page). To fit into a small core store, the compiler was divided into 7 passes:

Pass 1: Symbol analysis
Pass 2: Syntax analysis
Pass 3: Scope analysis
Pass 4: Declaration analysis
Pass 5: Statement analysis
Pass 6: Code selection
Pass 7: Code assembly

The main efficiency problem is to avoid random references to the slow disk and access it strictly sequentially during compilation. The compiler is loaded one pass at a time. Each pass makes a single sequential scan of the program text and outputs intermediate code on the disk. This becomes the input to the next pass.

So the compiler can be viewed as a pipeline consisting of passes connected by disk buffers. Since the available machine is sequential only one pass is executed at a time.

A multipass compiler not only makes store allocation and disk access efficient. It also simplifies the programming task considerably. In a one-pass compiler, each procedure performs a variety of compilation tasks at once [8]. This tends to make procedures and symbol tables large and complicated. In a multipass compiler, syntax analysis, semantic analysis, and code generation can be dealt with separately in smaller passes that use simpler data structures tailored to their tasks.

Each pass is essentially a minicompiler that only needs to know the syntax of its input and output languages. The data structures and procedures used by one pass are irrelevant to another. We found it extremely helpful to define the function of each pass by syntax graphs of its expected input and output. These graphs are included in [4].

3. INTERMEDIATE FILES

The compiler uses four files: source text and listing, pass input and output. The latter are stored on disk, the former on any available medium. These files are accessed by five procedures implemented within the operating system [9]:

read	inputs a character from the source text
write	outputs a character on the source listing
get	inputs a disk page from the previous pass
put	outputs a disk page to the next pass
length	defines the number of disk pages output by the previous pass

After each pass, the disk files exchange roles: the output file of the previous pass becomes the input file of the next pass, and the former input file becomes the next output file.

Disk access times are reduced as follows: The pages of the intermediate files are interleaved on the disk. This makes the disk head sweep slowly across both files during a pass instead of moving wildly back and forth between them. The pages that contain the compiler code are arranged on the disk in a manner that minimizes rotational delay during compiler loading [9].

A pass can build tables in core store and leave them there for the next pass. This is done by passing a single heap pointer as a parameter from each pass to its successor.

The loading and execution of the passes is controlled by a small Pascal program that also opens and closes all intermediate files [9].

4. PASS SUMMARY

Symbol analysis scans the program text character by character and converts symbols, identifiers, and numeric constants into unique integers. Identifiers are looked up by hashing. This pass does not distinguish between different uses of the same identifier in different contexts.

Syntax analysis checks the program syntax by means of a set of recursive procedures - one for each language construct [8]. Syntax errors are handled by erasing part of the program text to make it look syntactically correct to the rest of the compiler.

Scope analysis checks the access rights of processes, monitors, classes, procedures, and with statements. It uses a stack to handle nested contexts. The top of the stack defines the identifiers that are declared within the current context. They are popped at the end of the context. Every identifier referred to by the program is looked up in the nested name table to see if it is accessible. Different uses of the same identifier in several contexts are replaced by unique integers. This pass also replaces constant identifiers by their values or addresses. Apart from this, scope analysis is only concerned about whether an identifier can be used within a given context, but does not worry about what kind of object it refers to.

Declaration analysis checks that declarations of constants, types, variables, and procedures are consistent and computes the length of types and the addresses of variables. It builds a table of identifier attributes and distributes them whereever they are referenced in statements. After this pass, declarations have disappeared from the intermediate code.

Statement analysis checks that operands and operators are compatible. This is done by means of a stack that simulates program execution by operating on data types rather than data values [6]. In this pass and the previous one, semantic errors are handled by replacing undefined types or incorrect operands by universal ones that are compatible with anything. This prevents an avalanche of error messages from a single semantic error.

Code selection selects code pieces to be generated and computes the length of procedure code and temporary variables. It leaves a table of program labels, stack requirements, and large constants in core store.

Code assembly outputs virtual code in which program labels are replaced by relative addresses. The generation of virtual code is straightforward; no optimization is attempted. It is interpreted by machine code on the PDP 11/45 computer [3]. This pass also prints error messages from the other passes (but does not generate code, if there are any errors).

5. SCOPE ANALYSIS

It is the scope rules more than anything else that distinguishes Concurrent Pascal from other programming languages (such as sequential Pascal).

A Concurrent Pascal program consists of a hierarchy of abstract data types (classes, monitors, and processes). An abstract data type can only be accessed through procedures associated with it. A procedure can refer to its own temporary variables and to the permanent variables of the data type it operates on.

Data types and procedures cannot be recursive. This implies that procedures associated with a data type cannot call one another.

To enforce these rules, scope analysis associates an access attribute with every identifier [4].

Names with external access may only be referenced outside the scope in which they are declared. Example: monitor procedures.

Names with internal access may only be referenced inside the scope in which they are declared. Examples: monitor variables and procedure parameters.

Names with incomplete access may not be referenced until their declaration has been completed. Example: type declarations.

6. TESTING

The compiler was tested by means of a technique invented by Naur [6]. The passes were tested in their natural order starting with pass 1. For each pass we used a Concurrent Pascal text to force the pass to execute all statements at least once.

During testing the compiler lists the source text and the intermediate code produced by each pass. A comparison of the input and output of a pass immediately reveals if something is wrong. The corresponding input operator usually tells in which procedure the problem is. After correction of the error the test is repeated.

This test output mechanism of about 20 lines is a permanent part of the compiler and can always be turned on to document compiler errors revealed by a particular program text.

The generated code checks that subscripts are within range, that pointers are initialized, and that references to variant records are compatible with their tag values. These checks were invaluable during testing of the compiler. In a sample of 64 compiler failures during testing, 50 per cent were range errors, 20 per cent were pointer errors, and 28 per cent variant errors. All made the compiler terminate with a message of the form "pass 3, line 307, range error" (or something similar). Only one of the failures made the compiler go into an endless loop without any indication of what was wrong. Anyone who has tested compilers written in assembly language will recognize the value of an abstract programming language that makes checking at compile and run time possible.

It took four months to write the compiler and three months to test it. It has been in use since January 1975 without problems.

A sequential Pascal compiler was derived from the concurrent one in one additional man-month. It can compile itself in 16 K words of core store. This compiler was moved from the PDP 11/45 computer to another minicomputer in another man-month.

7. SIZE AND PERFORMANCE

The following shows the storage requirements of the compiler when it compiles the Solo operating system - a Concurrent Pascal program of 1300 lines [9]:

	virtual code (K words)	data (K words)
Common	1000	1300
Pass 1	4000	5600
Pass 2	5600	1200
Pass 3	7800	6200
Pass 4	5800	4800
Pass 5	4000	300
Pass 6	3000	650
Pass 7	3600	650
Compiler	34800	20700

The compiler runs in 16 K words of core store. This includes 2 K words of common input/output procedures and data buffers.

After a basic loading time of 7 sec the compilation speed is 240 char/sec (or about 10 lines/sec). The compiler is about 60 per cent disk limited.

Acknowledgement

My attitude to computer programming has been deeply influenced by the years I spent in Peter Naur's group at Regnecentralen in Denmark. The compilation techniques developed thirteen years by him are still among the best. The design of the Concurrent Pascal compiler is the result of daily conversations with my student Al Hartmann. The programming was done almost entirely by him. The development of Concurrent Pascal has been supported by the National Science Foundation under grant number DCR74-17331.

References

1. Brinch Hansen, P. The programming language Concurrent Pascal. IEEE Transactions on Software Engineering 1, 2 (June 1975), 199-207.

2. Brinch Hansen, P. Concurrent Pascal report. Information Science, California Institute of Technology, June 1975.

3. Brinch Hansen, P. Concurrent Pascal machine. Information Science, California Institute of Technology, Oct. 1975.

4. Hartmann, A.C. A Concurrent Pascal compiler for minicomputers. (Ph.D. Thesis) Information Science, California Institute of Technology, Sept. 1975.

5. Wirth, N. The programming language Pascal. Acta Informatica 1, (1971), 35-63.

6. Naur, P. The design of the Gier Algol compiler. BIT 3, 2-3 (1963), 124-40 & 145-66.

7. Brinch Hansen, P., and House, R. The Cobol compiler for the Siemens 3003. BIT 6, 1 (1966), 1-23.

8. Wirth, N. The design of a Pascal compiler. Software 1, (1971), 309-33.

9. Brinch Hansen, P. The Solo operating system. Information Science, California Institute of Technology, July 1975.