Ausdrücke

Infix-Notation	Prefix-Notation	Postfix-Notation
$a * (b + c)$	$* a + b c$	$a b c + *$

Operator-Assoziativität und -Preceedence:

$a + b * c$	entwpricht	$a + (b * c)$	(Pascal, C, ...)
$a = b < c$	entwpricht	$(a = b) < c$	(Pascal)
$a == b < c$	entwpricht	$a == (b < c)$	(C)

Einige Sprachen unterstützen benutzerdefinierte Operatoren.

Bedingte Ausdrücke:

$(a > b)\ ?\ a : b$	(C)
if $a > b$ then a else b	(ML)
case x of $1 => f1(y) \mid 2 => f2(y) \mid \_ => g(y)$	(ML)

Bedingte Anweisungen

if x > 0 **then if** x < 10 **then** x := 0 **else** x := 1000
if x > 0 **then begin if** x < 10 **then** x := 0 **end else** x := 1000
if x > 0 **then if** x < 10 **then** x := 0 **end else** x := 1000 **end**
if a **then** S1 **else if** b **then** S2 **else if** c **then** S3 **else** S4 **end**

```
switch (operator) {
   case '+': result = operand1 + operand2; break;
   case '-': result = operand1 - operand2; break;
   default:  break; }
```

case OPERATOR **is**
    when '+' => result := operand1 + operand2;
    when '−' => result := operand1 − operand2;
    when others => **null**;
end case

Schleifen

for var := lower **to** upper **do** statement (Pascal)
for (int i = 0; i < 10; i++) {...} (C++)
for var **in** discrete_range **loop** body **end loop** (Ada)

while condition **do** statement (Pascal)
while (expression) statement; (C)
while condition **loop** loop_body **end loop** (Ada)

repeat statement **until** condition (Pascal)
do statement **while** (expression); (C)
loop statement; **exit when** condition **end loop** (Ada)

A: **loop** ... **loop** ... **exit** A; ... **end loop** ... **end loop** A;

Routinen

procedure p(**var** x: T; y: Q; **function** f(z: R): integer);

```
void proc(int* x, int y) {
   *x = *x + y; }

void proc(int& x, int y) {
   x = x + y; }
```

Alias-Probleme: Lesbarkeit der Programme wird erschwert
Optimierungen werden verhindert

Beispiel: u := x + z + f(x,y) + f(x,y) + x + z

Euclid vermeidet Alias-Probleme gänzlich

→ semantische Einschränkungen, dynamische Prüfungen

Ausnahmebehandlung in Ada

Vordefinierte Ausnahmen: Constraint_Error
Program_Error
Storage_Error
Tasking_Error

Benutzerdefinierte Ausnahmen (Bsp): Help: **exception**;

Expliziter Aufruf einer Ausnahme (Bsp): **raise** Help;

```
begin -- Block mit Ausnahmebehandlung
   ... statements ...
   exception when Help => ... statements ...
             when Constraint_Error => ... statements ...
             when others => ... statements ...
end;
```

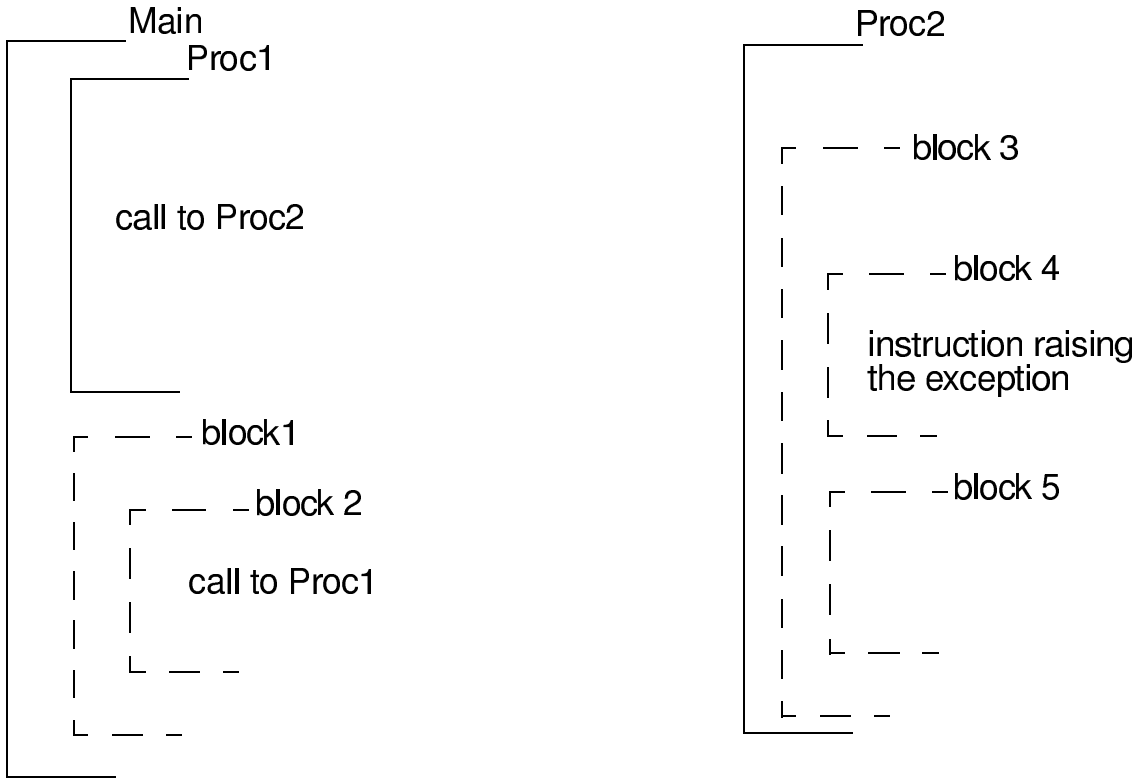

FIGURE 4.1 An example of an Ada program that raises an exception

Implementierung in SIMPLESEM

Ausnahmen erhalten vom Compiler eindeutige Namen.

Wenn eine Ausnahme aufgetreten ist, wird der entsprechende Handler dynamisch (entlang des “dynamic link”) gesucht.

Jeder “activation record” enthält einen Zeiger auf eine statische Tabelle; jeder Tabelleneintrag assoziiert einen Ausnahmenamen mit dem Rumpf des entsprechenden Handlers.

Es ist möglich, daß eine Ausnahme über dessen Scope hinaus propagiert wird.

Ausnahmebehandlung in C++

Beliebige Daten werden propagiert.

Aufruf einer Ausnahme:

```
throw Help(MSG1);
```

Deklaration propagierter Funktionen:

```
void foo() throw(Help, Zerodivide);
```

Spezielle Funktionen: unexpected()
terminate()

```
class Help {…};
    // objects of this class have a public attribute "kind" of type enumeration,
    // describing the kind of help requested, and other public fields which carry
    // specific information about the point in the program where help is requested
class Zerodivide {};
    // assume that objects of this class are generated by the run-time system
…
try {
    // fault tolerant block which may raise Help or Zerodivide exceptions
    …
}
catch(Help msg) {
    // handles a Help request transmitted by object msg
    switch(msg.kind) {
        case MSG1:
            …;
        case MSG2:
            …;
        …
    }
    …
}
catch(Zerodivide) {
    // handles a division by 0
…
}
```

FIGURE 4.3 An example of exception handling in C++

Ausnahmebehandlung in Java

Alle propagierten Ausnahmen müssen deklariert werden:

```
void foo() throws Help;
```

Form eines "try"-Blocks:

```
try
    block;
catch(exception_type_1)
    handler_1;
...
catch(exception_type_n)
    handler_n;
finally
    final_block;
```

Ausnahmebehandlung in ML

```
exception Neg

fun fact(n) =
   if n < 0 then raise Neg
   else if n = 0 then 1
      else n * fact(n - 1)

fun fact_0(n) =
   fact(n) handle Neg => 0;
```

```
try_several_methods is
local
    i: INTEGER;
    --it is automatically initialized to 0
do
    try_method(i);
rescue
    i := i + 1;
    if i < max_trials then
        --max_trials is a constant
        retry
    end
end
```

FIGURE 4.4 A “retrial” strategy in Eiffel

File F1

```
class A {};
void f() {
    …
    throw A();
    …
}
```

File F2

```
extern void f();
class A {};
void foo() {
    …
    try {
        …
        f();
        …
    }
    catch(A a) {
        …
    }
}
```

FIGURE 4.5 Two files containing parts of a C++ program

Pattern Matching

```
datatype day = Mon | Tue | Wed | Thu | Fri | Sat | Sun
fun day_off(Sun) = true
   | day_off(Sat) = true
   | day_off(_) = false

fun reverse(nil) = nil
   | reverse(head::tail) = reverse(tail) @ [head]

fun rev(nil) = nil
   | rev(0::tail) = [0] @ [tail]
   | rev(head::tail) = rev(tail) @ [head]
```

Nichtdeterminismus und Backtracking

A if B or
  C or
  D;

C if E and
  F and
  G;

D if I or
  H;

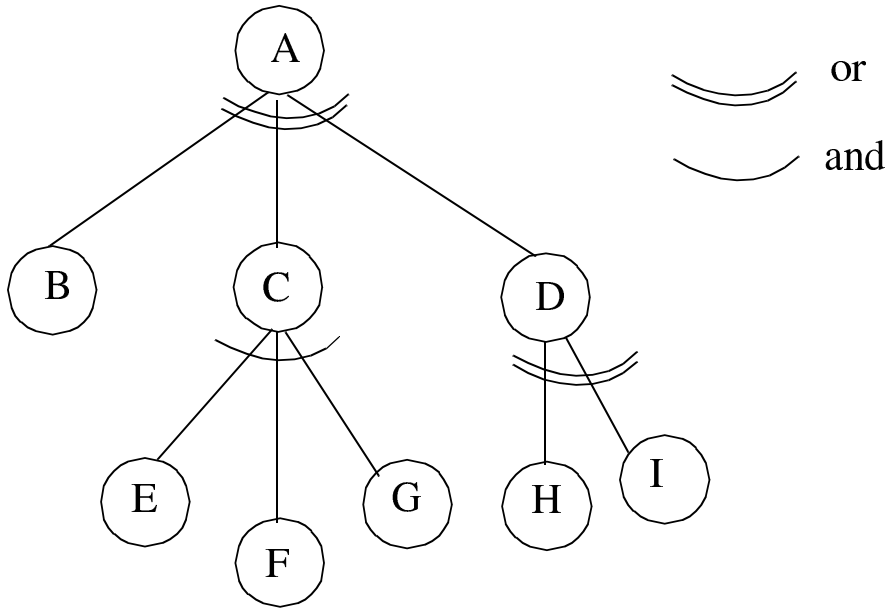

FIGURE 4.6 An and/or tree

Ereignisgesteuerte Berechnungen

Einige Sprachen unterstützen “event-driven computations” direkt (Visual Basic, Visual C++, Tcl/Tk)

In aktiven Datenbanken werden Trigger verwendet:

```
on event
when condition
   do action

on insert in EMPLOYEE
when TRUE
   do emp_number++
```

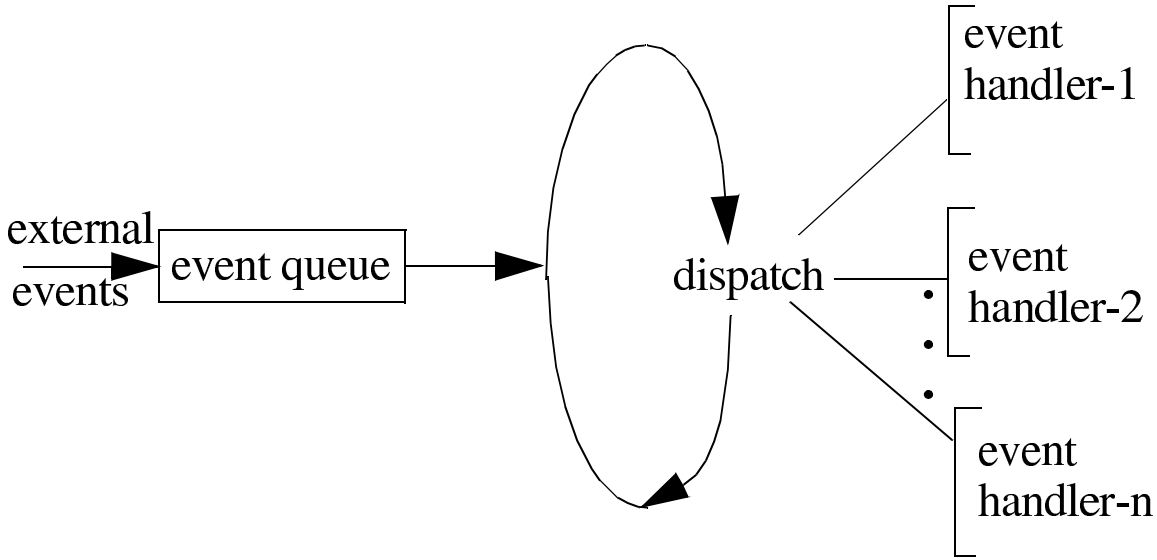

FIGURE 4.7 Structure of an event-driven computation

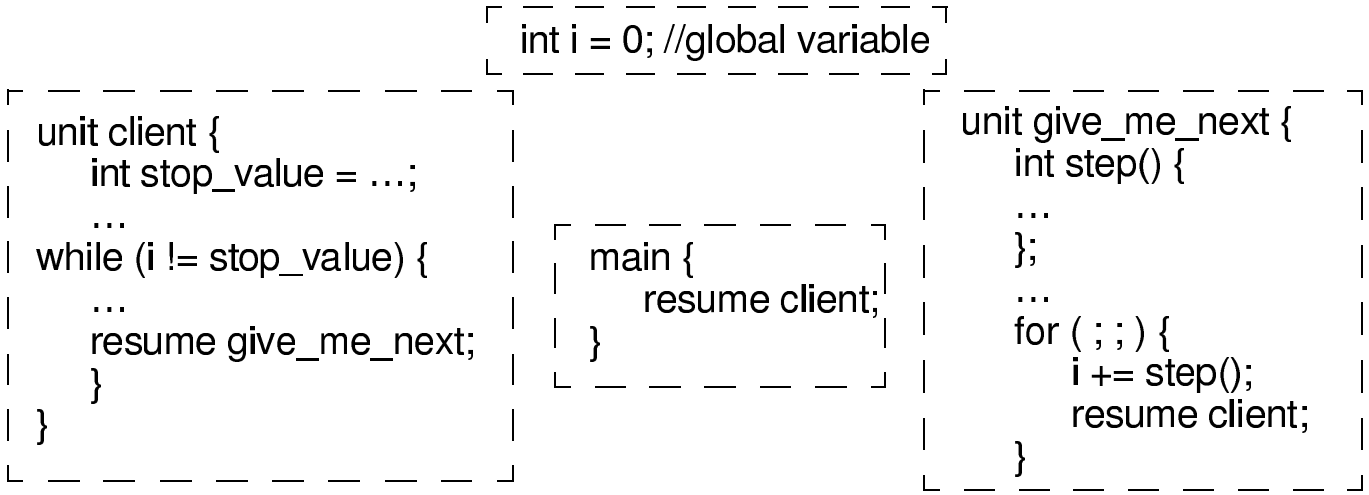

FIGURE 4.8 An example of coroutines

```
Process Producer

repeat forever {
    produce an element;
    append it to the buffer;
}
```

```
Process Consumer

repeat forever {
    remove an element from the buffer;
    operate on it;
}
```

FIGURE 4.9 Sample processes: producer and consumer

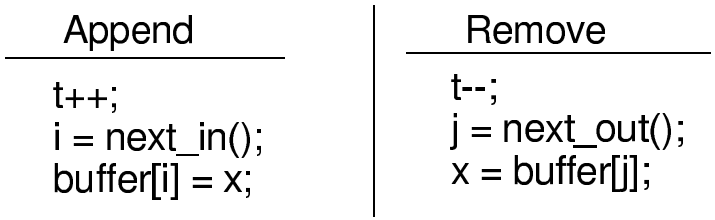

FIGURE 4.10 Operations to append and remove from a buffer

Prozesse

```
task type SERVER is
   entry NEXT_REQUEST(NR: in REQUEST);
   entry SHUT_DOWN;
end SERVER;

type SERVER_PTR is access SERVER;

MY_SERVER: SERVER;

task CHECKER is
   entry CHECK(T: in TEXT);
   entry CLOSE;
end CHECKER;

HIS_SERVER_PTR: SERVER_PTR := new SERVER;
```

Semaphore

Semaphore sind low-level Synchronisationsmechanismen.

Auf jedem Semaphor s sind zwei atomare Aktionen definiert:

P(s): if $s > 0$ then $s = s - 1$
else suspend current process

V(s): if there is a process suspended on the semaphore
then wake up process
else $s = s + 1$

Vor dem Eintritt in einen kritischen Abschnitt, der durch s gesichert ist, wird P(s) aufgerufen, nach Beendigung V(s).

```
buffer buf;                    // object of class buffer, with member functions
                               // append, remove, and size
semaphore mutex = 1;           // semaphore used to guarantee mutual exclusion
        in = 0;                // semaphore to control the reading from the buffer
        spaces =  buf.size(); // semaphore to control the writing into the buffer
process producer {
    int i;
    for ( ; ; ) {
        produce(i);
        P(spaces);        // wait for free spaces
        P(mutex);         // wait for buffer availability
                          // the buffer must be used in mutual exclusion
        buf.append(i);
        V(mutex);         // buffer is now available
        V(in)             // one more item in buffer
    };
};
process consumer {
    int j;
    for ( ; ; ) {
        P(in);            // wait for item in buffer
        P(mutex);         // wait for buffer availability
                          // the buffer must be used in mutual exclusion
         j = buf.remove();
        V(mutex);         // buffer is now available
        V(spaces)         // one more free space in buffer
    };
}
```

FIGURE 4.11 Producer-consumer example with semaphores

```
type fifostorage =
monitor
    var contents: array[1..n] of integer;  {buffer contents}
        tot: 0..n; {number of items in buffer}
        in,  {position of item to be added next}
        out: 1..n; {position of item to be removed next}
        sender, receiver: queue;
    procedure entry append (item: integer);
    begin if tot = n then delay(sender);
        contents[in] := item;
        in := (in mod n) + 1;
        tot := tot + 1;
        continue(receiver)
    end;
    procedure entry remove (var item: integer);
    begin if tot = 0 then delay(receiver);
        item := contents[out];
        out := (out mod n) + 1;
        tot := tot - 1;
        continue(sender)
    end;
begin {initialization part}
    tot := 0; in := 1; out := 1
end
```

FIGURE 4.12 Producer-consumer example with monitor

```
const n = 20;
type fifostorage =  …as above…
type producer =
    process (storage: fifostorage);
    var element: integer;
    begin cycle
                …
                storage.append(element);
                …
            end
    end;
type consumer =
    process(storage: fifostorage);
    var datum: integer;
    begin cycle
                …
                storage.remove(datum);
                …
            end
    end;
var meproducer: producer;
    youconsumer: consumer;
    buffer: fifostorage;
begin
    init buffer, meproducer(buffer), youconsumer(buffer)
end
```

FIGURE 4.13 A Concurrent Pascal program with two processes and one monitor

```
task Buffer_Handler is   --task declaration
    entry Append(Item: in Integer);
    entry Remove(Item: out Integer);
end;
task body Buffer_Handler is --task implementation
    N: constant Integer := 20;
    Contents: array (1..N) of Integer;
    In_Index, Out_Index: Integer range 1..N := 1;
    Tot: Integer range 0..N := 0;
begin loop
    select
        when Tot < N =>
            accept Append(Item: in Integer) do
                Contents(In_Index) := Item;
            end;
        In_index := (In_Index mod N)+1;
        Tot := Tot+ 1
    or
        when Tot > 0 =>
            accept Remove(Item: out Integer) do
                Item := Contents(Out_Index);
            end;
        Out_Index := (Out_Index mod N) + 1;
        Tot := Tot - 1;
    end select;
end loop;
end Buffer_Handler;
```

FIGURE 4.14 An Ada task that manages a buffer

PRODUCER

```
loop
    produce a new value V;
    Buffer_Handler.Append(V);
    exit when V is the end-of-stream
                value;
end loop;
```

CONSUMER

```
loop
    Buffer_Handler.Remove(V);
    exit when V is the end-of-stream
                value;
    consume V;
end loop;
```

FIGURE 4.15 Sketch of the producer and consumer tasks in Ada

```
protected type Fifo_Storage is
    entry Append (Item: in Integer);
    entry Remove (Item: out Integer);
private
    N: constant Integer := 20;
    Contents: array (1..N) of Integer;
    In_Index, Out_Index: Integer range 1..N := 1;
    Tot: Integer range 0..N := 0;
end Fifo_Storage;

protected body Fifo_Storage is
    entry Append (Item: in Integer) when Tot < N is
    begin
        Contents(In_Index) := Item;
        In_Index := (In_Index mod N) + 1;
        Tot := Tot + 1
    end Append;

    entry Remove (Item: out Integer) when Tot > 0 is
    begin
        Item := Contents(Out_Index);
        Out_Index := (Out_Index mod N) + 1;
        Tot := Tot - 1;
    end Remove;
end Fifo_Storage;
```

FIGURE 4.16 A protected Ada type implementing a buffer

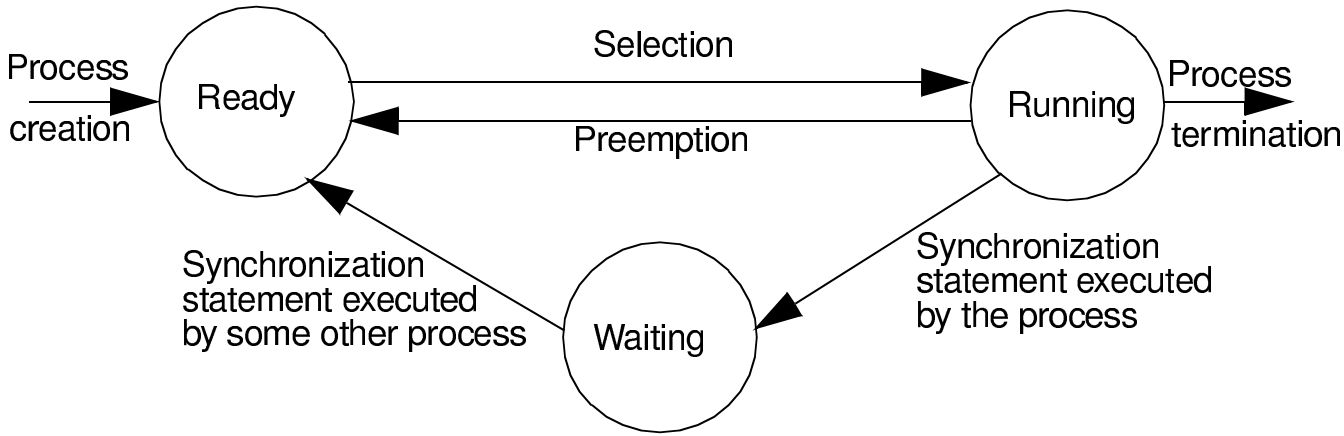

FIGURE 4.17 State diagram for a process

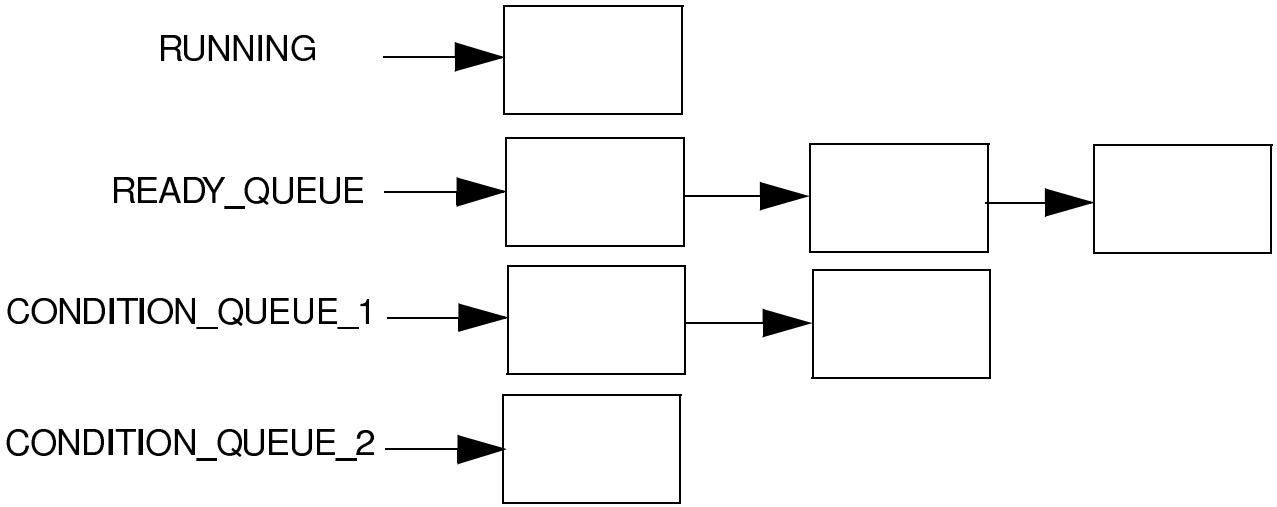

FIGURE 4.18 Data structures of the kernel

Operationen des Betriebssystemkerns

Betriebssystemkern ist ADT mit diesen Operationen:

enqueue: $\text{Queue} \times \text{Descriptor} \rightarrow \text{Queue}$
dequeue: $\text{Queue} \rightarrow \text{Queue} \times \text{Descriptor}$
empty: $\text{Queue} \rightarrow \text{Boolean}$

Jeder Uhren-Interrupt führt diese Operation aus:

```
Suspend_and_Select()
   RUNNING = process_status;
   READY_QUEUE.enqueue(RUNNING);
   RUNNING = READY_QUEUE.dequeue();
   process_status = RUNNING;
```

Implementierung der Semaphore

```
Suspend_on_Condition(c)
   RUNNING = process_status;
   CONDITION_QUEUE(c).enqueue(RUNNING);
   RUNNING = READY_QUEUE.dequeue();
   process_status = RUNNING;

Awaken(c)
   RUNNING = process_status;
   READY_QUEUE.enqueue(RUNNING);
   READY_QUEUE.enqueue(
                        CONDITION_QUEUE(c).dequeue());
   RUNNING = READY_QUEUE.dequeue();
   process_status = RUNNING;
```

Implementierung der Monitore

"mutual exclusion" durch Abschalten von Interrupts während der Ausführung von Monitor-Prozeduren

```
Continue(c)
   RUNNING = process_status (interrupts enabled,
       program counter set to return point from monitor call);
   READY_QUEUE.enqueue(RUNNING);
   if not CONDITION_QUEUE(c).empty()
      READY_QUEUE.enqueue(
                          CONDITION_QUEUE(c).dequeue());
   RUNNING = READY_QUEUE.dequeue();
   process_status = RUNNING;
```

Implementierung von Rendezvous (1)

Jedes "entry" hat einen Descriptor mit diesen Feldern:

- O: Boolean; true if entry is open
- W: waiting queue (task descriptors of callers)
- T: descriptor of task owning entry
- I: pointer to first instruction of accept body

```
Call_Entry(e)
   RUNNING = process_status;
   DESCR(e).W.enqueue(RUNNING);
   if DESCR(e).O
      for all entries oe of DESCR(e).T do oe.O = false;
      RUNNING = DESCR(e).T;
      RUNNING.ip = DESCR(e).I;
   else RUNNING = READY_QUEUE.dequeue();
   process_status = RUNNING;
```

Implementierung von Rendezvous (2)

```
At_End_Of_Accept_Body(e)
   RUNNING = process_status;
   READY_QUEUE.enqueue(DESCR(e).W.dequeue());
   READY_QUEUE.enqueue(RUNNING);
   RUNNING = READY_QUEUE.dequeue();
   process_status = RUNNING;

Execute_Accept_Statement(e)
   if DESCR(e).W.empty()
      DESCR(e).O = true;
      DESCR(e).T = process_status;
      RUNNING = READY_QUEUE.dequeue();
      process_status = RUNNING;
   -- else simply continue executing the accept body
```

Implementierung von Rendezvous (3)

```
Execute_Select_Statement()
   LOE = list of open entries in selection;
   if LOE is empty raise Program_Error;
   else
      if DESCR(e).W.empty() (for all e in LOE)
         for all e in LOE do
            DESCR(e).O = true;
            DESCR(e).T = process_status;
         RUNNING = READY_QUEUE.dequeue();
         process_status = RUNNING;
      else
         choose an e with not DESCR(e).W.empty();
         proceed execution from instruction DESCR(e).I;
```