```
var
  clients:  array[1..max] of clienttype;

process screen;
var
  i: integer;
begin
  repeat
    select
      for i := 1 to max replicate
        coms[i] ? any;
        writeln('Message from process ',i);
    or
      terminate
    end
  forever
end;

var
  i: integer;

begin
  cobegin
    screen;
    for i := 1 to max do
      clients[i](i)
  coend
end.
```

9.3.2. The select statement with accept alternatives

The following is a solution to the bounded buffer problem using the Ada style of inter-process communication.

```
program pcon5;

(* buffered producer-consumer with ada rendezvous *)

process buffer;
  entry take(var ch: char);
  entry put(ch: char);

const
     buffmax = 4;
```

```
var
     store: array[0..buffmax] of char;
     nextin, nextout, count: integer;

begin
     nextin := 0;
     nextout := 0;
     count := 0;
     repeat
          select
          when count <> 0 =>
               accept take(var ch: char) do
                    ch := store[nextout];
               count := count - 1;
               nextout := (nextout + 1) mod (buffmax + 1);
          or
          when count <= buffmax =>
               accept put(ch: char)  do
                    store[nextin] := ch;
               count := count + 1;
               nextin := (nextin + 1) mod (buffmax + 1 );
          or
               terminate
          end  (* select *)
     forever
end;  (* buffer *)
```

```
process producer;
var
     local: char;
begin
     for local := 'a' to 'z' do
          buffer.put(local)
end;  (* producer *)
```

```
process consumer;
var
     local: char;
begin
     repeat
          buffer.take(local);
          write(local)
     until local = 'z';
     writeln
end;  (* consumer *)
```

```
begin
     cobegin
          producer;
          consumer;
          buffer
     coend
end.
```

9.4. Process States and Transitions

This section summarises the effects on process states of the features described in this chapter.

1. A process that attempts to execute a **`select`** on which there are no open alternatives with pending calls becomes blocked unless there is an **`else`** part. (In the special case that there are no open guards and no **`else`** part, a run-time error must be signalled).
2. A process that becomes blocked on a **`select`** with a **`terminate`** alternative enters the "termstate" state. It may return to the "executable" state if a call occurs on an open alternative or (in the case of a ,channel or entry mapped to a source of interrupts) when an appropriate interrupt occurs. It will prroceed directly to the "terminated" state if the run-time system detects that all processes are in "termstate" or are already "terminated".
3. A process that becomes blocked on a **`select`** with a **`timeout`** alternative is considered "delayed". It may become executable when the specified time has elapsed, or when a call occurs on an open alternative, or (in the case of a channel or entry mapped to a source of interrupts) when an appropriate interrupt occurs., whichever of these events occurs first.
4. A process that becomes blocked on a **`select`** with neither **`terminate`** nor **`timeout`** alternatives becomes "suspended" if none of the open-guarded alternatives is mapped to a source of interrupts, or "awaiting interrupt" if one or more such alternatives is so mapped.