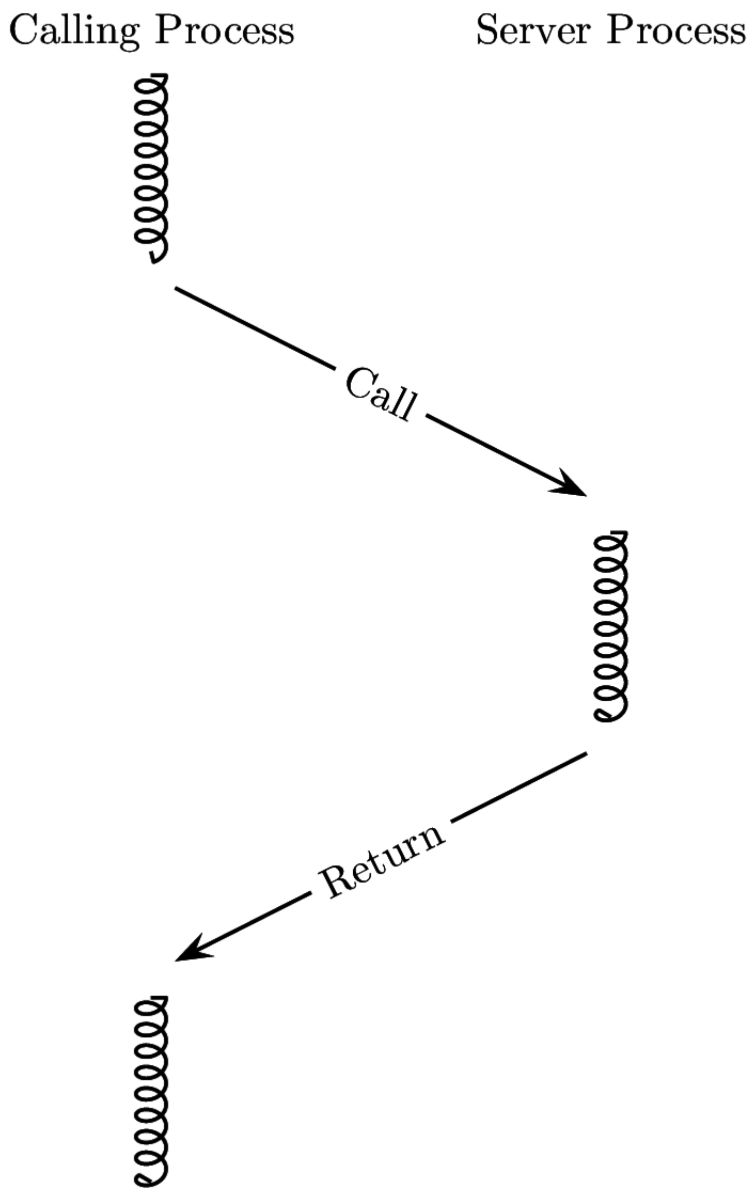
RPC and Rendezvous

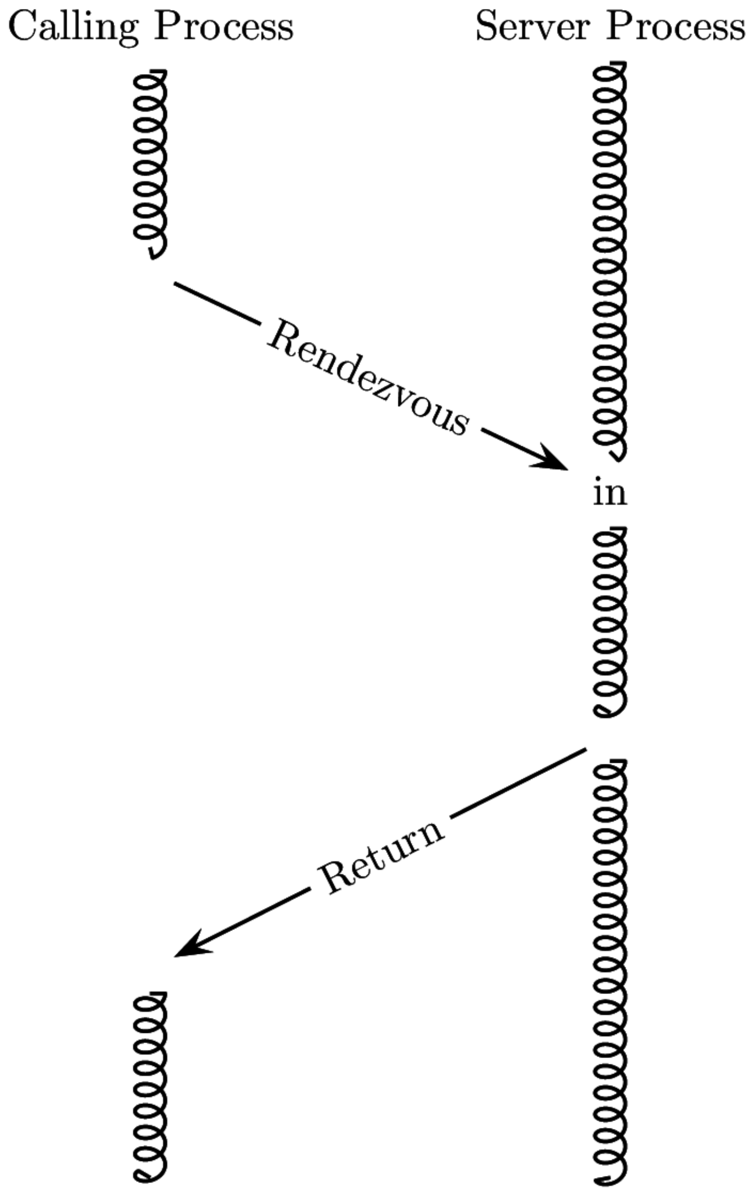
Andrews, Chapter 08

- Message passing:
  - Best for filters and interacting peers.
  - Clients and servers require two messages.
  - Each client needs a different reply channel.
  - Can lead to large number of channels.
- RPC and rendezvous best for client/servers:
  - Combine aspects of monitors and synchronous message passing.
  - Call and return like message passing.
  - Caller delays until return like monitor calls.
- RPC:
  - Create a new process for each call.
- Rendezvous:
  - Call to an existing process.
- Neither supports asynchronous messages directly, but we can program a buffer process.

RPC



Rendezvous



RPC *vs.* Monitors

- Monitors:
  - Monitors have two kinds of components: processes and monitors.
  - Processes communicate and synchronize by calling monitor procedures.
  - Processes and monitors are all in the same address space.
- RPC:
  - One program component: the module.
  - Modules have both processes and procedures.
  - Modules reside in different address spaces (*e.g.* nodes in a network).

Modules

```
module mname
    headers of exported operations;
body
    variable declarations;
    initialization code;
    procedures for exported operations;
    local procedures and processes;
```

- Local processes are called *background processes*
- The header of an operation `opname` is specified with op:

```
op opname( formals) [returns  result]
```

- The operation itself is implemented by `proc`:

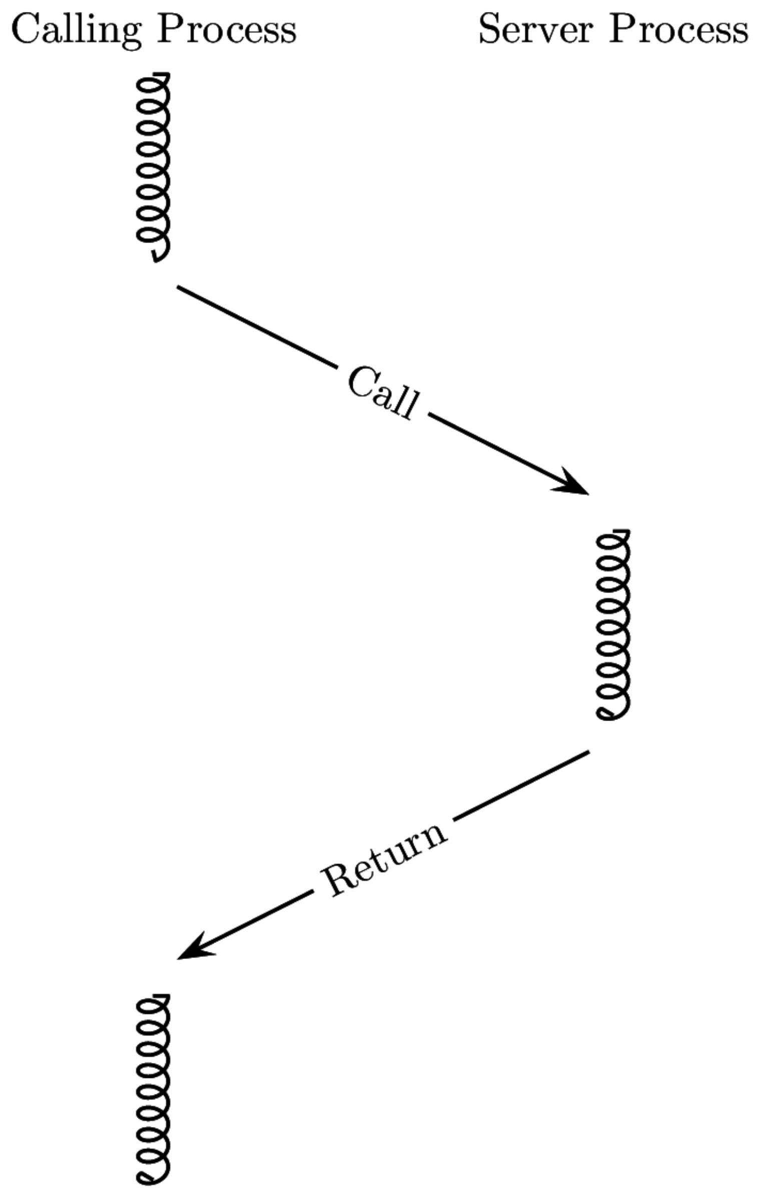
```
proc opname( formal identifiers) returns  result identifier
    declarations of local variables;
    statements;
end
```

- A process in one module calls a procedure in another module with:

```
call mname.opname( arguments)
```

RPC

- A *new process* services the call.
- Arguments are passed as messages.
- The calling process delays during the call.
- When the call returns it sends results as a message and terminates.
- After receiving results, the calling process continues.



Synchronization in Modules

- Synchronization between caller and server is implicit.
- From client appears as normal procedure call.
- Need mutually exclusive access for more than one call to the server and background processes. Two approaches:
  - All server processes are mutually exclusive (similar to monitors). Can use semaphores or condition variables for other synchronization.
  - Server processes execute concurrently. Need to explicitly program both mutual exclusion and synchronization. Can use any of the methods discussed in the book.
- The first approach is simpler to implement and program with.
  - Shared variables automatically protected from concurrent access.
  - Context switching can occur only at entry, exit, and delay points.
- The second approach is more general.
  - Fits better with modern architectures, which are multi-core.
  - Time slicing can gain control of runaway processes.
- The text assumes the second approach.
  - All RPC modules will have explicit synchronization programmed using semaphores.

```
module TimeServer
  op get_time() returns int;  # retrieve time of day
  op delay(int interval);     # delay interval ticks
body
  int tod = 0;         # the time of day
  sem m = 1;           # mutual exclusion semaphore
  sem d[n] = ([n] 0);  # private delay semaphores
  queue of (int waketime, int process_id) napQ;
  ## when m == 1, tod < waketime for delayed processes

  proc get_time() returns time {
    time = tod;
  }

  proc delay(interval) {    # assume interval > 0
    int waketime = tod + interval;
    P(m);
    insert (waketime, myid) at appropriate place on napQ;
    V(m);
    P(d[myid]);   # wait to be awakened
  }

  process Clock {
    start hardware timer;
    while (true) {
      wait for interrupt, then restart hardware timer;
      tod = tod+1;
      P(m);
      while (tod >= smallest waketime on napQ) {
        remove (waketime, id) from napQ;
        V(d[id]);  # awaken process id
      }
      V(m);
    }
  }
end TimeServer
```

Figure 8.1 A time server module.

```
module FileCache   # located on each diskless workstation
  op read(int count; result char buffer[*]);
  op write(int count; char buffer[]);
body
  cache of file blocks;
  variables to record file descriptor information;
  semaphores for synchronization of cache access (if needed);

  proc read(count,buffer) {
    if (needed data is not in cache) {
      select cache block to use;
      if (need to write out the cache block)
        FileServer.writeblk(...);
      FileServer.readblk(...);
    }
    buffer = appropriate count bytes from cache block;
  }

  proc write(count,buffer) {
    if (appropriate block not in cache) {
      select cache block to use;
      if (need to write out the cache block)
        FileServer.writeblk(...);
    }
    cache block = count bytes from buffer;
  }
end FileCache
```

- Running on client. Synchronization not necessary if one cache per process.

Figure 8.2 (a) Distributed file system: File cache.

```
module FileServer   # located on a file server
  op readblk(int fileid, offset; result char blk[1024]);
  op writeblk(int fileid, offset; char blk[1024]);
body
  cache of disk blocks;
  queue of pending disk access requests;
  semaphores to synchronize access to the cache and queue;
  # N.B. synchronization code not shown below

  proc readblk(fileid, offset, blk) {
    if (needed block not in the cache) {
      store read request in disk queue;
      wait for read operation to be processed;
    }
    blk = appropriate disk block;
  }

  proc writeblk(fileid, offset, blk) {
    select block from cache;
    if (need to write out the selected block) {
      store write request in disk queue;
      wait for block to be written to disk;
    }
    cache block = blk;
  }

  process DiskDriver {
    while (true) {
      wait for a disk access request;
      start a disk operation; wait for interrupt;
      awaken process waiting for this request to complete;
    }
  }
end FileServer
```

- Running on server. Synchronization necessary.

Figure 8.2 (b) Distributed file system: File server.

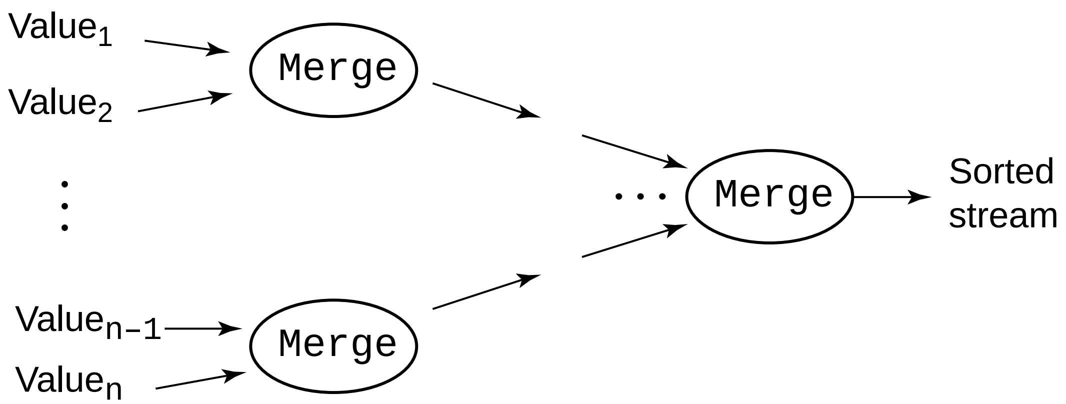


Figure 7.3 A sorting network of **`Merge`** processes.

Sorting network of merge filters using RPC

- Sorting networks easy with message passing.
  - Using RPC we need to program process-to-process communication.
- Also have to link instances of merge filters together.
  - Use dynamic naming.
  - *Capabilities* are pointers to operations.
  - We pass output channels as capabilities to the operations.

```
call Merge[i].initialize(Merge[j].in2)
```

```
optype stream = (int);  # type of data stream operations

module Merge[i = 1 to n]
  op in1 stream, in2 stream;  # input streams
  op initialize(cap stream);  # link to output stream
body
  int v1, v2;    # input values from streams 1 and 2
  cap stream out;  # capability for output stream
  sem empty1 = 1, full1 = 0, empty2 = 1, full2 = 0;

  proc initialize(output) {  # provide output stream
    out = output;
  }

  proc in1(value1) {  # produce next value for stream 1
    P(empty1); v1 = value1; V(full1);
  }

  proc in2(value2) {  # produce next value for stream 2
    P(empty2); v2 = value2; V(full2);
  }

  process M {
    P(full1); P(full2);  # wait for two input values
    while (v1 != EOS and v2 != EOS)
      if (v1 <= v2)
          { call out(v1); V(empty1); P(full1); }
      else  # v2 < v1
          { call out(v2); V(empty2); P(full2); }
    # consume the rest of the non-empty input stream
    if (v1 == EOS)
      while (v2 != EOS)
          { call out(v2); V(empty2); P(full2); }
    else  # v2 == EOS
      while (v1 != EOS)
          { call out(v1); V(empty1); P(full1); }
    call out(EOS);  # append sentinel
  }
end Merge
```

Figure 8.3 Merge-sort filters using RPC.

```
chan in1(int), in2(int), out(int);
process Merge {
  int v1, v2;
  receive in1(v1);  # get first two input values
  receive in2(v2);
  # send smaller value to output channel and repeat
  while (v1 != EOS and v2 != EOS) {
    if (v1 <= v2)
      { send out(v1); receive in1(v1); }
    else   # (v2 < v1)
      { send out(v2); receive in2(v2); }
  }
  # consume the rest of the non-empty input channel
  if (v1 == EOS)
    while (v2 != EOS)
      { send out(v2); receive in2(v2); }
  else   # (v2 == EOS)
    while (v1 != EOS)
      { send out(v1); receive in1(v1); }
  # append a sentinel to the output channel
  send out(EOS);
}
```

Figure 7.2 A filter process that merges two input streams.

- Much simpler code with message passing.

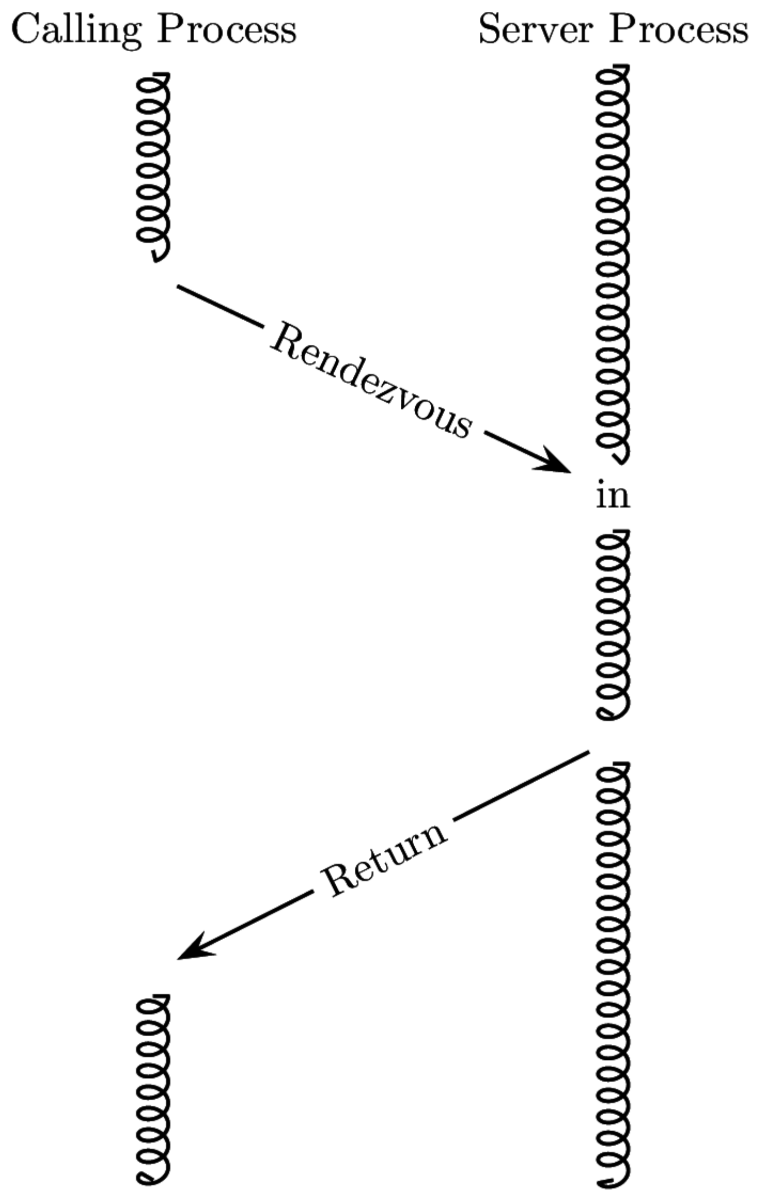
```
module Exchange[i = 1 to 2]
  op deposit(int);
body
  int othervalue;
  sem ready = 0;   # used for signaling

  proc deposit(other) {    # called by other module
    othervalue = other;    # save other’s value
    V(ready);              # let Worker pick it up
  }
  process Worker {
    int myvalue;
    call Exchange[3-i].deposit(myvalue);  # send to other
    P(ready);             # wait to receive other’s value
    ...
  }
end Exchange
```

Figure 8.4 Exchanging values using RPC.

- Note semaphore use to wait until other operation is done.
- Trivial code with asynchronous message passing.

Rendezvous

- Rendezvous combines communication and synchronization.
- Clients invoke operations with `call`
- A server process uses an *input statement* to wait for and handle the call.
- Operations are not handled concurrently.



Input Statements

```
in op1(formals1) and B1 by e1 -> S1;
[] op2(formals2) and B2 by e2 -> S1;
...
[] opN(formalsN) and BN by eN -> SN;
ni
```

- Provided by **Ada** in slightly modified form.
- Part before the `->` is the **guard**
- `opi` is the name of the operation
- `Bi` is the **synchronization expression**
  - boolean determines if this branch is open
- `ei` is the **scheduling expression**
  - number determines priority
- `Si` is a statement list.

Input Statements

```
in op1(formals1) and B1 by e1 -> S1;
[] op2(formals2) and B2 by e2 -> S1;
...
[] opN(formalsN) and BN by eN -> SN;
ni
```

- A guard **succeeds** when
  1. the operation has been called
  2. the synchronization expression is true (or omitted)
- The boolean can depend on the parameters (not true in Ada).
- Execution of `in` delays until some guard succeeds.
- If more than one guard succeeds the `in` statement services the oldest.
- If scheduling expressions are present, oldest of minima.
- Synchronization and scheduling expressions also present in some message-passing systems, *e.g.* MPI.

Rendezvous

```
module BoundedBuffer
  op deposit(typeT), fetch(result typeT);
body
  process Buffer {
    typeT buf[n];
    int front = 0, rear = 0, count = 0;
    while (true)
      in deposit(item) and count < n ->
            buf[rear] = item;
            rear = (rear+1) mod n; count = count+1;
      [] fetch(item) and count > 0 ->
            item = buf[front];
            front = (front+1) mod n; count = count-1;
      ni
  }
end BoundedBuffer
```

Figure 8.5 Rendezvous implementation of a bounded buffer.

- Bounded buffers are, of course, asynchronous channels.

Monitor

```
monitor Bounded_Buffer {

  typeT buf[n];     # an array of some type T
  int front = 0,    # index of first full slot
      rear = 0;     # index of first empty slot
      count = 0;    # number of full slots
  ## rear == (front + count) % n
  cond not_full,    # signaled when count < n
       not_empty;   # signaled when count > 0

  procedure deposit(typeT data) {
    while (count == n) wait(not_full);
    buf[rear] = data; rear = (rear+1) % n; count++;
    signal(not_empty);
  }

  procedure fetch(typeT &result) {
    while (count == 0) wait(not_empty);
    result = buf[front]; front = (front+1) % n; count--;
    signal(not_full);
  }

}
```

Figure 5.4 Monitor implementation of a bounded buffer.

- Bounded buffer with monitor.

- Explicit `waits` instead of synchronization conditions.

```
module Table
  op getforks(int), relforks(int);
body
  process Waiter {
    bool eating[5] = ([5] false);
    while (true)
      in getforks(i) and not (eating[left(i)] and
           not eating[right(i)] -> eating[i] = true;
      [] relforks(i) ->
            eating[i] = false;
      ni
  }
end Table

process Philosopher[i = 0 to 4] {
  while (true) {
    call getforks(i);
    eat;
    call relforks(i);
    think;
  }
}
```

Figure 8.6 Centralized dining philosophers using rendezvous.

Rendezvous

```
module TimeServer
  op get_time() returns int;
  op delay(int);
  op tick();      # called by clock interrupt handler
body TimeServer
  process Timer {
    int tod = 0;  # time of day
    while (true)
      in get_time() returns time -> time = tod;
      [] delay(waketime) and waketime <= tod -> skip;
      [] tick() -> { tod = tod+1; restart timer; }
      ni
  }
end TimeServer
```

Figure 8.7 A time server using rendezvous.

- Server must monitor delayed requests.
- Not provided by Ada.

RPC

```
module TimeServer
  op get_time() returns int;  # retrieve time of day
  op delay(int interval);     # delay interval ticks
body
  int tod = 0;         # the time of day
  sem m = 1;           # mutual exclusion semaphore
  sem d[n] = ([n] 0);  # private delay semaphores
  queue of (int waketime, int process_id) napQ;
  ## when m == 1, tod < waketime for delayed processes

  proc get_time() returns time {
    time = tod;
  }

  proc delay(interval) {    # assume interval > 0
    int waketime = tod + interval;
    P(m);
    insert (waketime, myid) at appropriate place on napQ;
    V(m);
    P(d[myid]);   # wait to be awakened
  }

  process Clock {
    start hardware timer;
    while (true) {
      wait for interrupt, then restart hardware timer;
      tod = tod+1;
      P(m);
      while (tod >= smallest waketime on napQ) {
        remove (waketime, id) from napQ;
        V(d[id]);  # awaken process id
      }
      V(m);
    }
  }
end TimeServer
```

Figure 8.1 A time server module.

```
module SJN_Allocator
  op request(int time), release();
body
  process SJN {
    bool free = true;
    while (true)
      in request(time) and free by time -> free = false;
      [] release() -> free = true;
      ni
  }
end SJN_Allocator
```

Figure 8.8 Shortest-job-next allocator using rendezvous.

Rendezvous

```
optype stream = (int);  # type of data streams

module Merge[i = 1 to n]
  op in1 stream, in2 stream;  # input streams
  op initialize(cap stream);  # link to output stream
body
  process Filter {
    int v1, v2;       # values from input streams
    cap stream out;  # capability for output stream
    in initialize(c) -> out = c ni
    # get first values from input streams
    in in1(v) -> v1 = v; ni
    in in2(v) -> v2 = v; ni
    while (v1 != EOS and v2 != EOS)
      if (v1 <= v2)
          { call out(v1); in in1(v) -> v1 = v; ni }
      else   # v2 < v1
          { call out(v2); in in2(v) -> v2 = v; ni }
    # consume the rest of the non-empty input stream
    if (v1 == EOS)
      while (v2 != EOS)
          { call out(v2); in in2(v) -> v2 = v; ni }
    else  # v2 == EOS
      while (v1 != EOS)
          { call out(v1); in in1(v) -> v1 = v; ni }
    call out(EOS);
  }
end Merge
```

Figure 8.9 Merge sort filters using rendezvous.

RPC

```
optype stream = (int);  # type of data stream operations

module Merge[i = 1 to n]
  op in1 stream, in2 stream;  # input streams
  op initialize(cap stream);  # link to output stream
body
  int v1, v2;    # input values from streams 1 and 2
  cap stream out;  # capability for output stream
  sem empty1 = 1, full1 = 0, empty2 = 1, full2 = 0;

  proc initialize(output) {  # provide output stream
    out = output;
  }

  proc in1(value1) {  # produce next value for stream 1
    P(empty1); v1 = value1; V(full1);
  }

  proc in2(value2) {  # produce next value for stream 2
    P(empty2); v2 = value2; V(full2);
  }

  process M {
    P(full1); P(full2);  # wait for two input values
    while (v1 != EOS and v2 != EOS)
      if (v1 <= v2)
          { call out(v1); V(empty1); P(full1); }
      else  # v2 < v1
          { call out(v2); V(empty2); P(full2); }
    # consume the rest of the non-empty input stream
    if (v1 == EOS)
      while (v2 != EOS)
          { call out(v2); V(empty2); P(full2); }
    else  # v2 == EOS
      while (v1 != EOS)
          { call out(v1); V(empty1); P(full1); }
    call out(EOS);  # append sentinel
  }
end Merge
```

Figure 8.3 Merge-sort filters using RPC.

Rendezvous

```
module Exchange[i = 1 to 2]
  op deposit(int);
body
  process Worker {
    int myvalue, othervalue;
    if (i == 1) {   # one process calls
      call Exchange[2].deposit(myvalue);
      in deposit(othervalue) -> skip; ni
    } else {        # the other process receives
      in deposit(othervalue) -> skip; ni
      call Exchange[1].deposit(myvalue);
    }
    ...
  }
end Exchange
```

Figure 8.10 Exchanging values using rendezvous.

- Can't be symmetric.

RPC

```
module Exchange[i = 1 to 2]
  op deposit(int);
body
  int othervalue;
  sem ready = 0;   # used for signaling

  proc deposit(other) {    # called by other module
    othervalue = other;    # save other's value
    V(ready);              # let Worker pick it up
  }
  process Worker {
    int myvalue;
    call Exchange[3-i].deposit(myvalue);  # send to other
    P(ready);             # wait to receive other's value
    ...
  }
end Exchange
```

Figure 8.4 Exchanging values using RPC.

- Symmetric.

Skipping §8.3, §8.4, §8.5

Tasks, Rendezvous, and Protected Types in Ada

- `http://en.wikibooks.org/wiki/Ada_Programming/Tasking`

```
task Name is
  entry declarations;
end;
```

```
entry Identifier(formals);
```

```
task body Name is
  local declarations;
begin
  statements;
end Name;
```

```
call T.E(actuals);
```

Ada tasks, entries, and call statements.

```
accept E(formals) do
  statement list;
end;


select when B₁ => accept statement; additional statements;
or ...
or     when Bₙ => accept statement; additional statements;
end select;


select entry call; additional statements;
else   statements;
end select;


select entry call; additional statements;
or     delay statement; additional statements;
end select;
```

Ada accept and select statements.

```
protected type Name is
  function, procedure, or entry declarations;
private
  variable declarations;
end Name;


protected body Name is
  function, procedure, or entry bodies;
end Name;


requeue Opname;
```

Ada protected types and requeue statement.

- Similar to Monitors
- At most one task executes a protected procedure at a time.
- Caller waits until it can have access *and* the guard is true.
- `requeue` puts the task back on the calling queue.

```
protected type Barrier is
  procedure Arrive;
private
  entry Go;              -- used to delay early arrivals
  count : Integer := 0; -- number who have arrived
  time_to_leave : Boolean := False;
end Barrier;

protected body Barrier is
entry ~~procedure~~ Arrive is begin
    count := count+1;
    if count < N then
      requeue Go;   -- wait for others to arrive
    else
      count := count-1; time_to_leave := True;
    end if;
  end;

  entry Go when time_to_leave is begin
    count := count-1;
    if count = 0 then time_to_leave := False; end if;
  end;
end Barrier;
```

Figure 8.17 Barrier synchronization in Ada.

- `requeue` statement defers completion of the call.
- Is it reusable?

```
with Ada.Text_IO; use Ada.Text_IO;
procedure Dining_Philosophers is
  subtype ID is Integer range 1..5;

  task Waiter is            -- Waiter spec
    entry Pickup(I : in ID);
    entry Putdown(I : in ID);
  end
  task body Waiter is separate;

  task type Philosopher is   -- Philosopher spec
    entry init(who : ID);
  end;

  DP : array(ID) of Philosopher;  -- the philosophers
  rounds : Integer;               -- number of rounds

  task body Philosopher is   -- Philosopher body
    myid : ID;
  begin
    accept init(who); myid := who; end;
    for j in 1..rounds loop
      -- "think"
      Waiter.Pickup(myid);  -- pick forks up
      -- "eat"
      Waiter.Putdown(myid); -- put forks down
    end loop;
  end Philosopher;

begin  -- read in rounds, then start the philosophers
  Get(rounds);
  for j in ID loop
    DP(j).init(j);
  end loop;
end Dining_Philosophers;
```

Figure 8.18 Dining philosophers in Ada: Main program.

```
separate (Dining_Philosophers)
task body Waiter is
  entry Wait(ID);     -- used to requeue philosophers
  eating : array (ID) of Boolean; -- who is eating
  want : array (ID) of Boolean;   -- who wants to eat
  go : array(ID) of Boolean;      -- who can go now
begin
  for j in ID loop           -- initialize the arrays
    eating(j) := False; want(j) := False;
  end loop;
  loop             -- basic server loop
   select
     accept Pickup(i : in ID) do  -- DP(i) needs forks
       if not(eating(left(i)) or eating(right(i))) then
         eating(i) := True;
       else
         want(i) := True; requeue Wait(i);
       end if;
     end;
   or
     accept Putdown(i : in ID) do  -- DP(i) is done
       eating(i) := False;
     end;
     -- check neighbors to see if they can eat now
     if want(left(i)) and not eating(left(left(i))) then
       accept Wait(left(i));
       eating(left(i)) := True; want(left(i)) := False;
     end if;
     if want(right(i)) and not eating(right(right(i)))
       then  accept Wait(right(i));
       eating(right(i)) := True; want(right(i)) := False;
     end if;
   or
     terminate;  -- quit when philosophers have quit
   end select;
  end loop;
end Waiter;
```

Figure 8.19 Dining philosophers in Ada: Waiter task.

```
module Table
  op getforks(int), relforks(int);
body
  process Waiter {
    bool eating[5] = ([5] false);
    while (true)
      in getforks(i) and not (eating[left(i)] and
           not eating[right(i)] -> eating[i] = true;
      [] relforks(i) ->
            eating[i] = false;
      ni
  }
end Table

process Philosopher[i = 0 to 4] {
  while (true) {
    call getforks(i);
    eat;
    call relforks(i);
    think;
  }
}
```

Figure 8.6 Centralized dining philosophers using rendezvous.

Skipping §8.7

The SR Language