Resource Control

Goals

- To understand how reliable resource control can be achieved between concurrent processes

- To understand:
  - Bloom's criteria
  - requeue in Ada and its need

- I assume that you have done deadlock and its detection and recovery in an Operating Systems — see book

Resource Management

- Concerns of modularity and information hiding dictate that resources must be encapsulated and be accessed only through a high-level interface; e.g., in Ada, with a package:

```
package Resource_Control is
   type Resource is limited private;
   function Allocate return Resource;
   procedure Free(This_Resource : Resource);
private
   type Resource is  ...
end Resource_Control;
```

- If the resource manager is a server then the package body will contain a task (or an access object to a task type)
- A protected resource will use a protected object within the package body

Resource Managemenr

- With monitor-based synchronization, e.g. POSIX with condition variables and mutexes, or Java protected resources are naturally encapsulated within a monitor

```
public class ResourceManager
{
  public synchronized Resource allocate();
  public synchronized void free(Resource r);
}
```

- Other forms of synchronization, such as busy waiting and semaphores, do not give the appropriate level of encapsulation, and will therefore not be considered
- Conditional critical regions (CCRs) are also not explicitly evaluated, as protected objects are, essentially, a modern form of CCR

Expressive Power and Ease of Use

- Toby Bloom suggested criteria for evaluating synchronization primitives:

- Expressive power: the ability to express required constraints on synchronization
- Ease of use of a synchronization primitive encompasses:
  - the ease with which it expresses each of these synchronization constraints
  - the ease with which it allows the constraints to be combined to achieve more complex synchronization schemes

Bloom's Criteria

- The information needed to express synchronization constraints can be categorised:
  - the type of service request
  - the order in which requests arrive
  - the state of the server and any objects it manages
  - the parameters of a request
  - the history of the object (i.e., the sequence of all previous service requests)
  - the priority of the client

Conditional Waits and Avoidance

- There are, in general, two linguistic approaches to constraining access to a service.
  - **conditional wait**: all requests are accepted, but any process whose request cannot be met is suspended on an internal queue; the conventional monitor typifies this approach (e.g. `wait BufferNotFull`)
  - **avoidance**: requests are not accepted unless they can be met; the conditions under which a request can safely be accepted are expressed as a guard on the action of acceptance (e.g. **`when`** `Buffer_Not_Full`)

Request Type

- Can be used to give preference to one request over another (e.g, read requests over write requests to shared data)
- With monitors, the read and write operations could be programmed as distinct procedures, but outstanding calls on monitor procedures are handled in an arbitrary or FIFO way; it is not possible to deal with read requests first; nor is it feasible to know how many outstanding calls there are to monitor procedures
- In Ada, different request types are represented by different entries in the server task or protected object; before gaining access to the entity (in order to queue on an entry), there is again no way of giving preference over other call; preference to particular requests can be given once they are queued through guards which use the count attribute

Example

```
protected Resource_Manager is
   entry Update(...);
   entry Modify(...);
   procedure Lock;
   procedure Unlock;
private
   Manager_Locked : Boolean := False;
   ...
end resource_manager;
```

`Update` requests to be given preference over `Modify` request

```
protected body Resource_Manager is

   entry Update(...) when not Manager_Locked is
   begin ... end Update;

   entry Modify(...) when not Manager_Locked and
                Update'Count = 0 is
   begin ... end Modify;

   procedure Lock is
   begin  Manager_Locked := True;  end Lock;

   procedure Unlock is
   begin Manager_Locked := False;  end Unlock;


end Resource_Manager;
```

Request Order

- Needed to ensure fairness or to avoid starvation of a client
- Monitors usually deal with requests in FIFO order
- In Ada, outstanding requests of the same type (calls to the same entry) can also be serviced in a FIFO manner
- Outstanding requests of different types (for example, calls to different entries within a select statement) are serviced in an arbitrary order with the FIFO policy
- There is no way of servicing requests of different types according to order of arrival unless a FIFO policy is used and all clients first call a common `register' entry:

```
Server.Register;
Server.Action(...);
```

- But this double call is not without difficulty

Server State

- Some operations may be permissible only when the server and the objects it administers are in a particular state.
- For example, a resource can be allocated only if it is free, and an item can be placed in a buffer only if there is an empty slot
- With avoidance synchronization, constraints based on state are expressed as guards and, with servers, on the positioning of accept statements (or message receive operators)
- Monitors are similarly quite adequate — with condition variables being used to implement constraints

Request Parameters

- The order of operations of a server may be constrained by information contained in the parameters of requests
- Such information typically relates to the identity or to the size of the request
- Easy to do with monitor-type primitives
- E.g.,
  - a request for a set of resources contains a parameter that indicates the size of the set required
  - if not enough resources are available then the caller is suspended
  - when any resources are released, all suspended clients are woken up (in turn) to see if their request can now be met.

Example in Java

```
public class ResourceManager
{
  private final int maxResources = ...;
  private int resourcesFree;

  public ResourceManager() { resourcesFree = maxResources; }

  public synchronized void allocate(int size) throws
         IntegerConstraintError, InterruptedException
  {
    if(size > maxResources) throw new
        IntegerConstraintError(1, maxResources, size);
    while(size > resourcesFree) wait();
    resourcesFree = resourcesFree - size;
  }

  public synchronized void free(int size)
  {
    resourcesFree = resourcesFree + size;
    notifyAll();
  }
}
```

Parameters and Avoidance Synchronisation

- With simple avoidance synchronization, the guards only have access to variables local to the server (or protected object)
- The data being carried with the call cannot be accessed until the call has been accepted
- It is, therefore, necessary to construct a request as a double interaction

Resource Allocation Example in Ada

- Associate an entry family with each type of request.
- Each permissible parameter value is mapped onto a unique index of the family so that requests with different parameters are directed to different entries.
- Obviously, this is only appropriate if the parameter is of discrete type.
- For small ranges, the technique described earlier for request type can be used, with the select statements enumerating the individual entries of the family
- However, for larger ranges a more complicated solution is needed

```
package Resource_Manager is
  Max_Resources : constant Integer := 100;
  type Resource_Range is new Integer range
                  1..Max_Resources;
  subtype Instances_Of_Resource is
                  Resource_Range range 1..50;

  procedure Allocate(Size : Instances_Of_Resource);
  procedure Free(Size : Instances_Of_Resource);
end Resource_Manager;
```

```
package body Resource_Manager is

  task Manager is
    entry Sign_In(Size : Instances_Of_Resource);
    entry Allocate(Instances_Of_Resource);
    entry Free(Size : Instances_Of_Resource);
  end Manager;

  procedure Allocate(Size : Instances_Of_Resource) is
  begin
    Manager.Sign_In(Size);  -- size is a parameter
    Manager.Allocate(Size); -- size is an index
  end Allocate;

  procedure Free(Size : Instances_Of_Resource) is
  begin
    Manager.Free(Size);
  end Free;
```

```
task body Manager is
  Pending : array(Instances_Of_Resource) of
     Natural := (others => 0);
  Resource_Free : Resource_Range := Max_Resources;
  Allocated : Boolean;
begin
  loop
    select  -- wait for first request
      accept Sign_In(Size : Instances_Of_Resource) do
        Pending(Size) := Pending(Size) + 1;
      end Sign_In;
    or
      accept Free(Size : Instances_Of_Resource) do
        resource_free := resource_free + size;
      end Free;
    end select;
```

```
loop  -- main loop
  loop
    -- accept any pending sign-in/frees, do not wait
    select
      accept Sign_In(Size : Instances_Of_Resource) do
        Pending(Size) := Pending(Size) + 1;
      end Sign_In;
    or
      accept Free(Size : Instances_Of_Resource) do
        Resource_Free := Resource_Free + Size;
      end Free;
    else
      exit;
    end select;
  end loop;
```

```
            -- now service largest request
            Allocated := False;
            for Request in reverse Instances_Of_Resource loop
              if Pending(Request) > 0 and
                          Resource_Free >= Request then
                accept Allocate(Request);
                  Pending(Request) := Pending(Request) - 1;
                  Resource_Free := Resource_Free - Request;
                  Allocated := True;
                  exit; --loop to accept new sign-ins
               end if;
            end loop;
            exit when not Allocated;
          end loop;
        end loop;
      end Manager;
    end Resource_Manager;
```

Access to “in” Parameters in Guards

```
protected resource_control is  -- NOT VALID ADA
  entry allocate(size : instances_of_resource);
  procedure free(size : instances_of_resource);
private
  resource_free : resource_range := MAX_RESOURCES;
end resource_control;

protected body resource_control is
  entry allocate(size : instances_of_resource)
      when resources_free >= size is    -- NOT VALID ADA
  begin
    resource_free := resource_free - size;
  end allocate;

  procedure free(size : instances_of_resource) is
  begin
    resource_free := resource_free + size;
  end free;
end resource_control;
```

Double Interactions and Atomic Actions

- A double interaction results from the lack of expressive power in simple avoidance synchronization.
- To program reliable resource control procedures, this structure must be implemented as an atomic action.
- With Ada, between the two calls; i.e., after `Sign_In` but before `Allocate`, an intermediate state of the client is observable from outside the “atomic action”:

```
begin
   Manager.Sign_In(Size);
   Manager.Allocate(Size);
end;
```

Double Interactions and Atomic Actions

- This state is observable in the sense that another task can abort the client between the two calls and leave the server in some difficulty
  - If the server assumes the client will make the second call, the abort will leave the server waiting for the call (that is, deadlocked)
  - If the server protects itself against the abort of a client (by not waiting indefinitely for the second call), it may assume the client has been aborted when in fact it is merely slow in making the call; hence the client is blocked erroneously

Handling Aborts

- In the context of real-time software, three approaches have been advocated for dealing with the abort problem
  - Define the abort primitive to apply to an atomic action rather than a process; forward or backward error recovery can then be used when communicating with the server
  - Assume that abort is only used in extreme situations where the breaking of the atomic action is of no consequence
  - Try and protect the server from the effect of client abort
- The third approach, in Ada, involves removing the need for the double call by requeuing the first call (rather than have the client make the second call)

Requester Priority

- If processes are runnable, the dispatcher can order their executions according to priority; the dispatcher cannot, however, have any control over processes suspended waiting for resources
- It is, therefore, necessary for the order of operations of the resource manager to be also constrained by the relative priorities of the client processes
- In Ada, Real-Time Java and POSIX it is possible to define a queuing policy that is priority ordered; but in general concurrent programming languages, processes are released from primitives in either an arbitrary or FIFO manner

Requester Priority

- It is possible to program clients so that they access the resource via different interfaces
- For a small priority range, this is now equivalent to the Request Type constraint
- For large priority ranges, it becomes equal to using Request Parameters

Requester Priority and Monitors

- Although monitors are often described as having a FIFO queue discipline, this is not really a fundamental property; priority ordered monitors are clearly possible.
- POSIX and RTJ implementation of monitors not only allows priority queues but also (conceptually) merges the external queue (of processes waiting to enter the monitor) and the internal one (of processes released by the signalling of a condition variable) to give a single priority ordered queue
- Hence, a higher priority process waiting to gain access to the monitor will be given preference over a process that is released internally

The Requeue Facility

- Enhances the usability of avoidance synchronization
- Requeue, in Ada, moves the task (which has been through one guard or barrier) to beyond another guard
- Analogy: consider a person (task) waiting to enter a room (protected object) which has one or more doors (guarded entries); once inside, the person can be ejected (requeued) from the room and once again be placed behind a (potentially closed) door
- Ada allows requeues between task and protected object entries; a requeue can be to the same entry, to another entry in the same unit, or to another unit altogether
- Requeues from task to protected object entries are allowed; however, the main use is to send the calling task to a different entry of the same unit

Resource Control

```
type Request_Range is range 1 .. Max;
type Resource ...;
type Resources is array(Request_Range range <>) of Resource;
protected Resource_Controller is
  entry Request(R : out Resources; Amount: Request_Range);
  procedure Free(R : Resources; Amount: Request_Range);
private
   entry Assign(R : out Resources; Amount: Request_Range);
   Freed : Request_Range := Request_Range'Last;
   New_Resources_Released : Boolean := False;
   To_Try : Natural := 0;

end Resource_Controller;
```

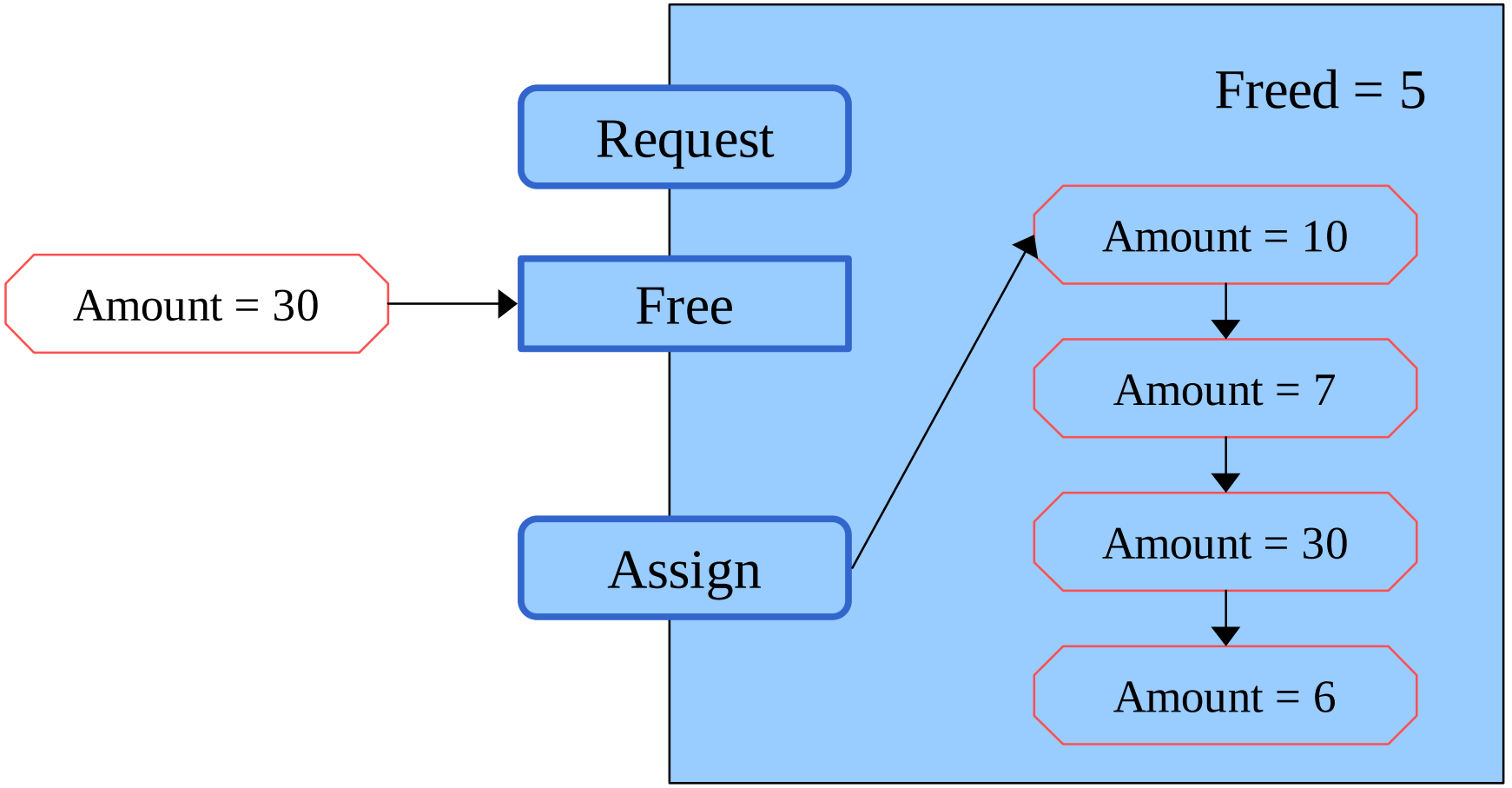
Resource Control II

```
protected body Resource_Controller is
  entry Request(R : out Resources; Amount: Request_Range)
        when Freed > 0 is
  begin
    if Amount <= Freed then Freed := Freed - Amount;
    else requeue Assign; end if;
  end Request;
  procedure Free(R : Resources; Amount: Request_Range) is
  begin
    Freed := Freed + Amount;
    -- free resources
    if Assign'Count > 0 then
      To_Try := Assign'Count;
      New_Resources_Released := True;
    end if;
  end Free;
```

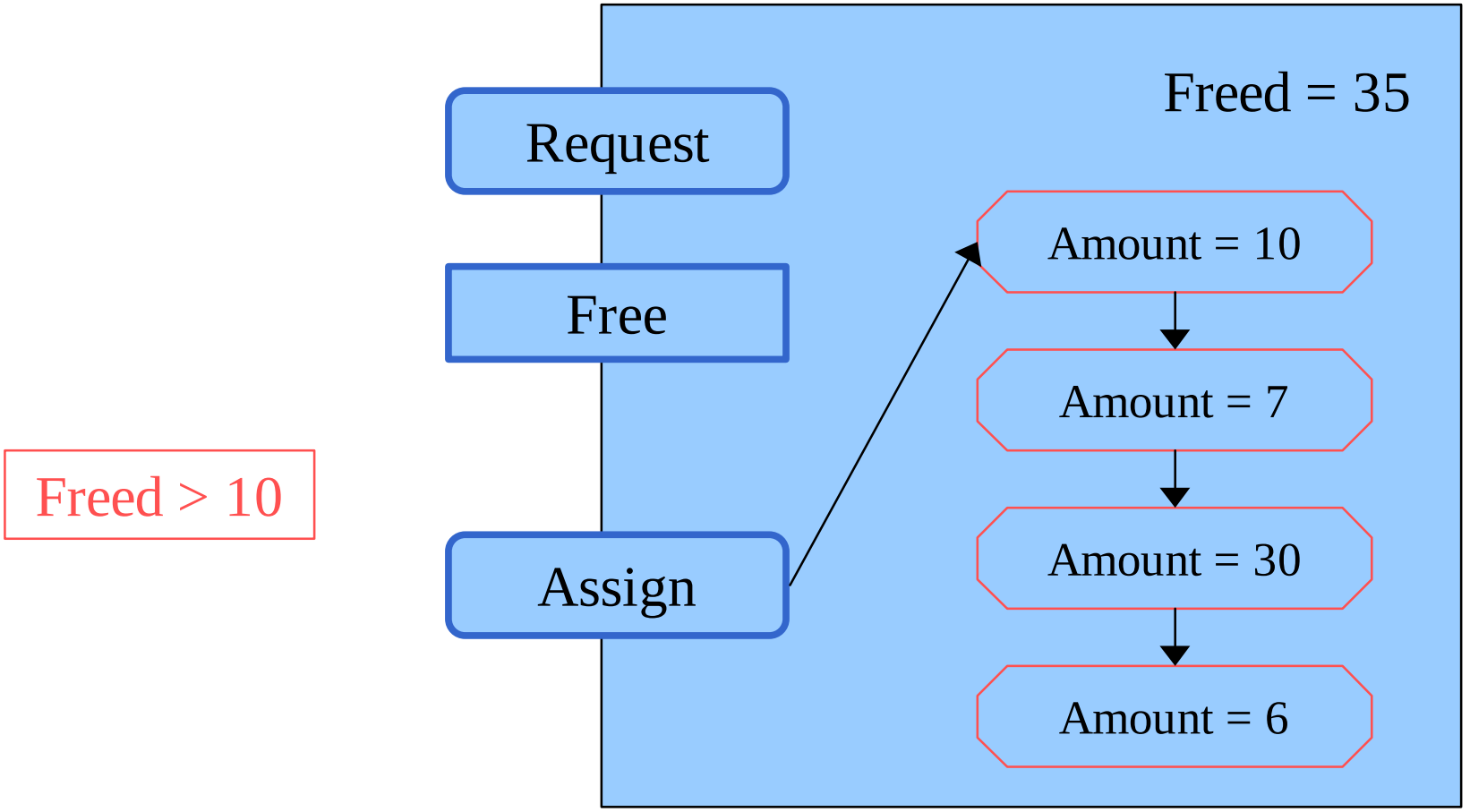
Resource Control III

```
  entry Assign(;R : out Resources; Amount: Request_Range)
          when New_Resources_Released is
   begin
     To_Try := To_Try - 1;
     if To_Try = 0 then
       New_Resources_Released := False;
     end if;
     if Amount <= Free then
       Freed := Freed - Amount;
       -- allocate
     else
       -- assumes FIFO queuing
       requeue Assign;
     end if;
   end Assign;
end Resource_Controller;
```

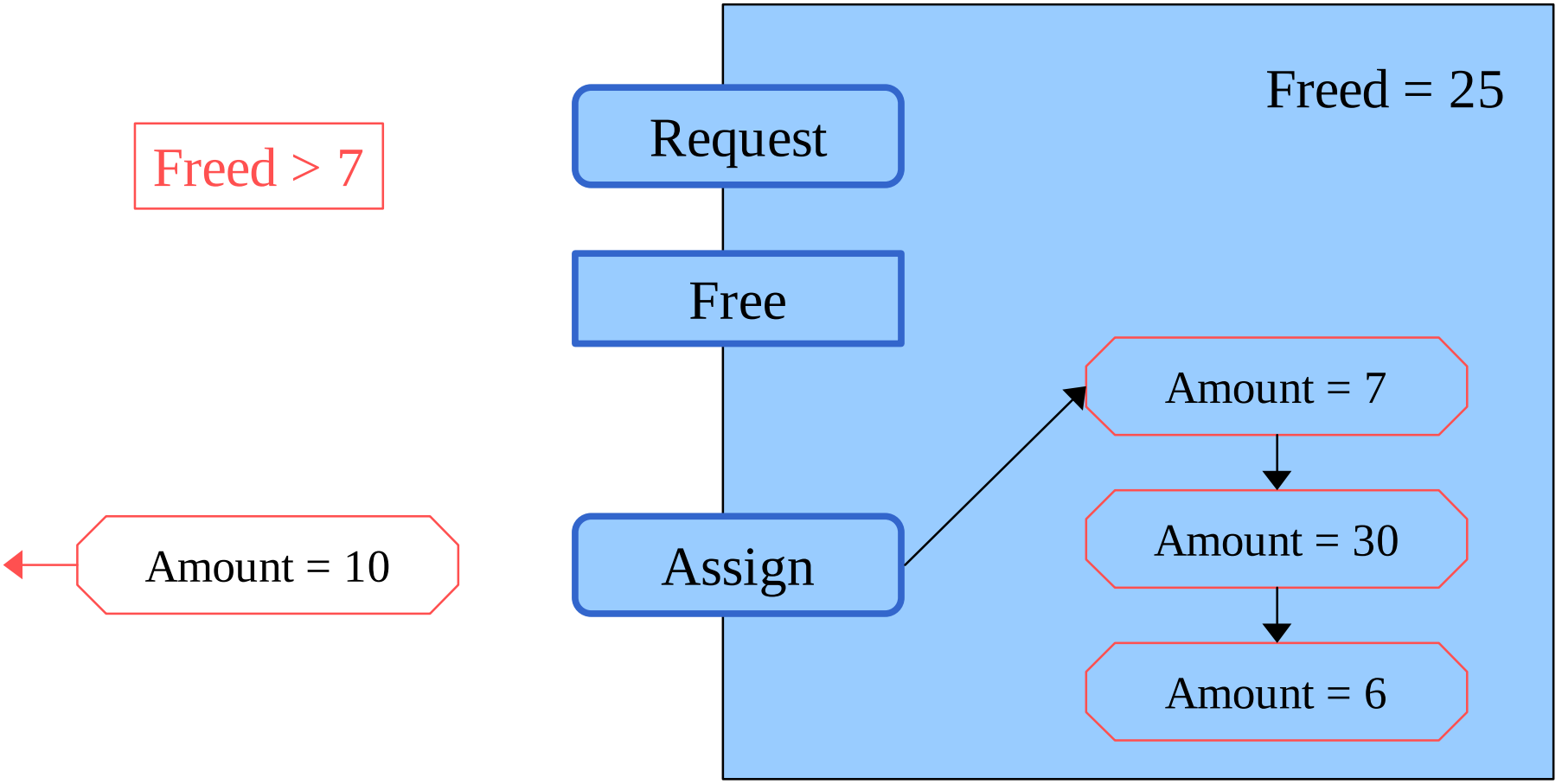
Illustration



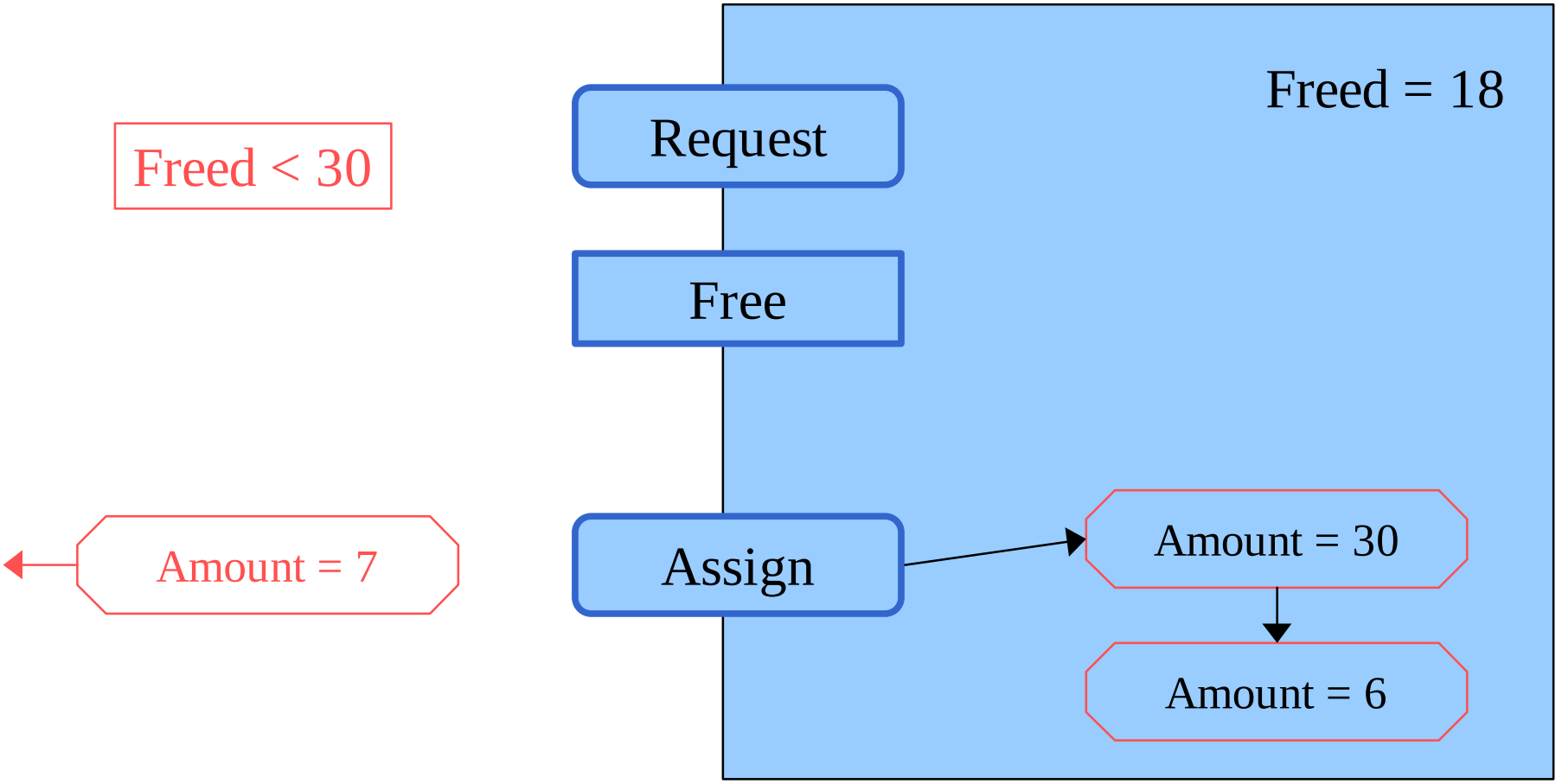
Illustration



Illustration

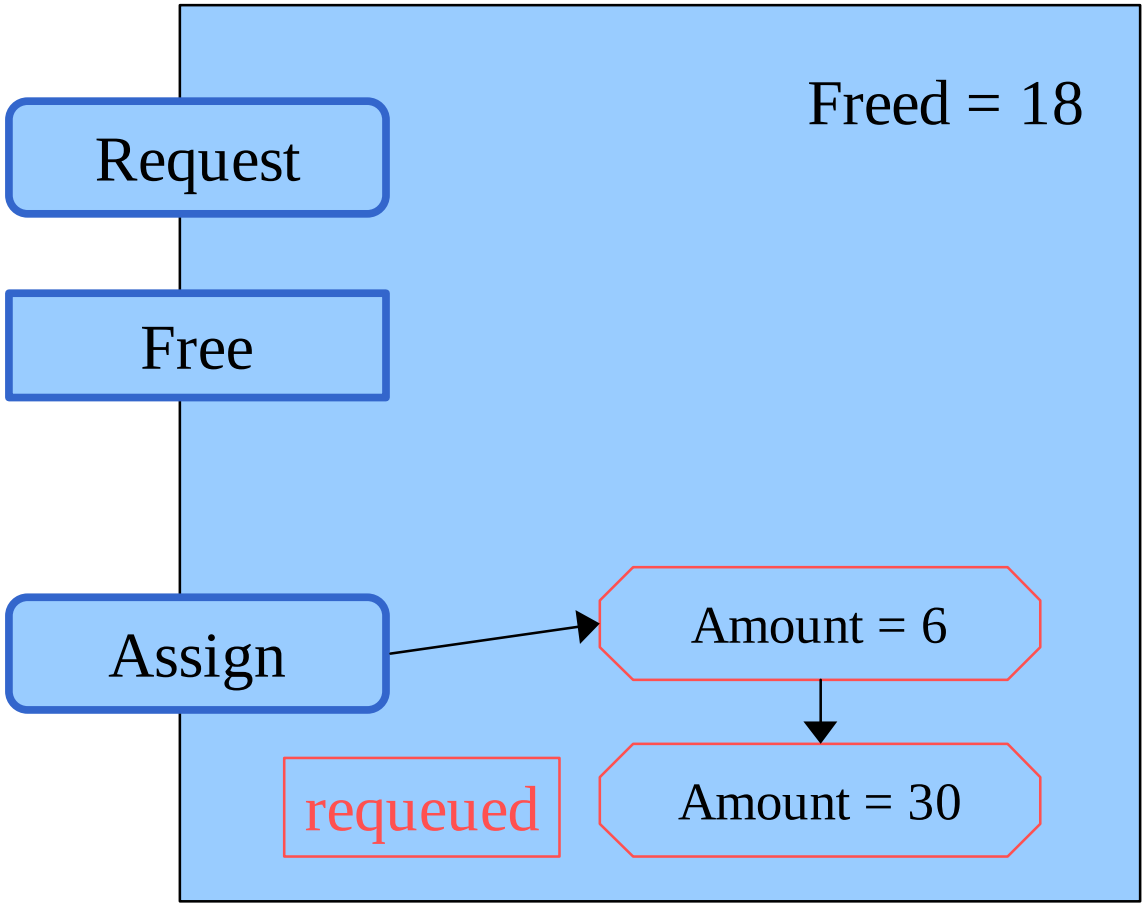


Illustration

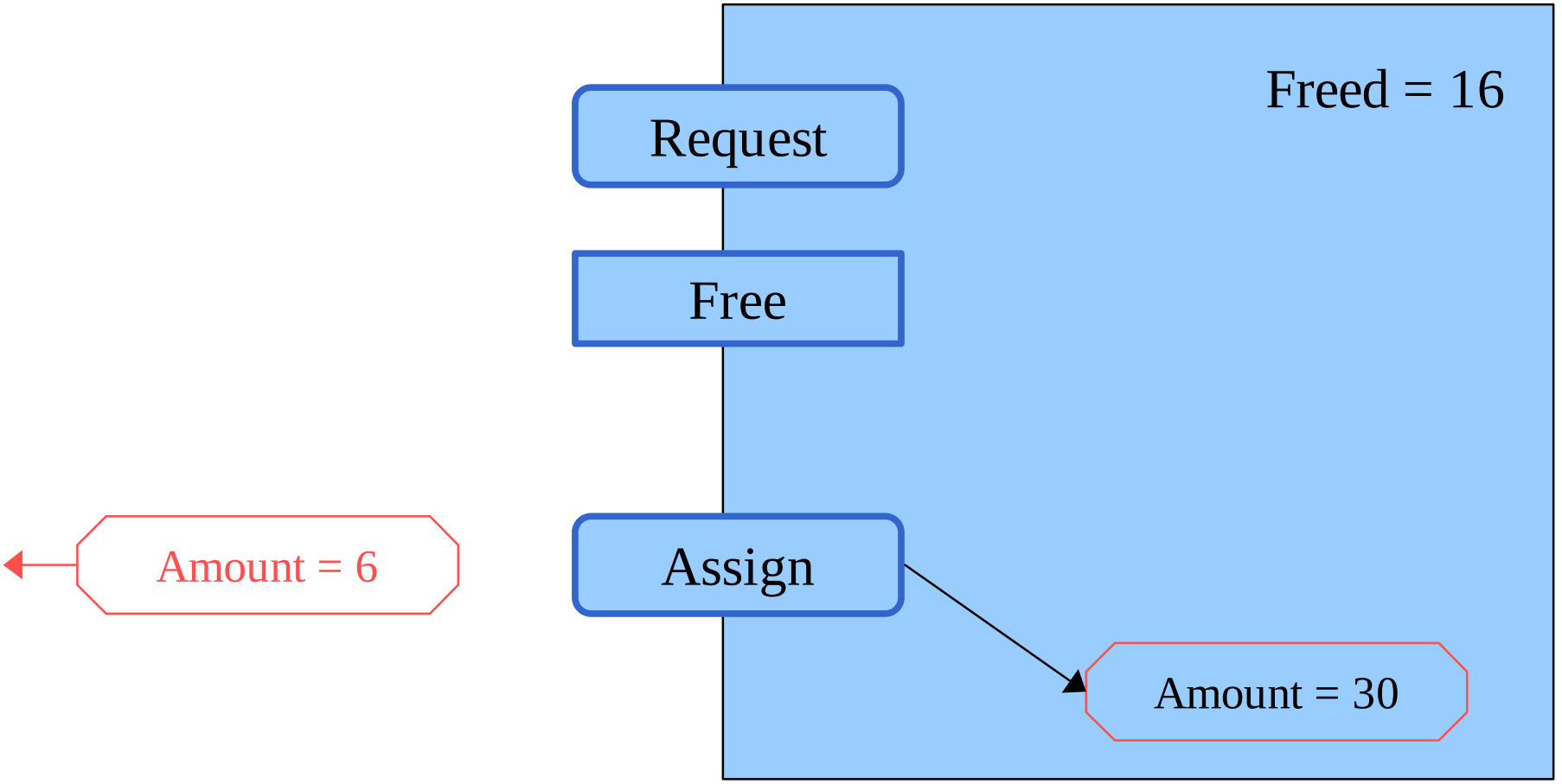


Illustration

Freed > 6



Illustration



Semantics of Requeue

- Requeue is not a simple entry call
- If an entry call is requeued, the call is completed

```
if Requeuing then
  requeue Entry_Name;
  -- Not Executed
end if;
```

- The full syntax for requeue is

```
requeue Entry_Name [with abort];
```

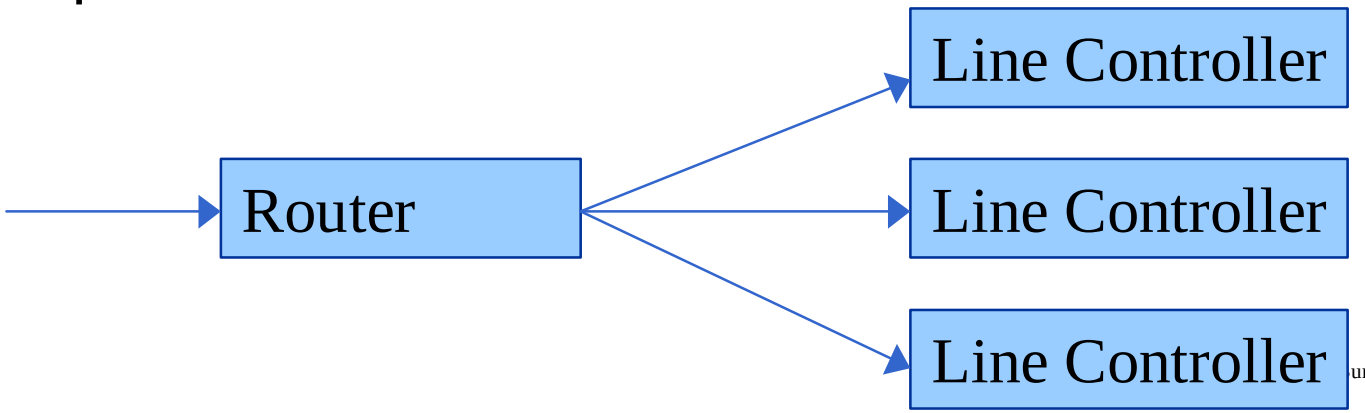
- When a task has any entry call accepted or begins executing a protected entry, any associated time-out is lost. Hence if the call is requeued indefinitely no time-out will occur. Also once the call have been requeued, by default, it is unabortable (Why?);

With abort

- **`requeue with abort`**
  - reinstates any timeout
  - allows the queued task to be aborted
- **`requeue`** (with no abort)
  - cancels any timeout
  - marks the queued task as non abortable
- The server task/protected object must decide on whether it expects the queued task to remain on the queue, or whether it can tolerate the task being removed

Requeuing to other Entries

- Any entry call can be requeued to another task or another protected unit as long as the parameters are type conformant
- Consider a network router which has a choice of three lines on which to forward messages: Line A is the preferred route, but if it becomes overloaded Line B can be used, if this becomes overloaded Line C can be used
- Each line is controlled by a server task
- A protected unit acts as an interface to the router



Router Example

```
type Line_Id is (Line_A, Line_B, Line_C);
type Line_Status is array (Line_Id) of Boolean;

task type Line_Controller(Id : Line_Id) is
  entry Request(...);
end Line_Controller;

protected Router is
  entry Send(...);
  procedure Overloaded(Line : Line_Id);
  procedure Clear(Line : Line_Id);
private
  OK : Line_Status := (others => True);
end Router;

LA: Line_Controller(Line_A);
LB: Line_Controller(Line_B);
LC: Line_Controller(Line_C);
```

Router Example II

```
protected body Router is
  entry Send(...) when OK(Line_A) or OK(Line_B) or
                       OK(Line_C) is
  begin
    if OK(Line_A) then
      requeue LA.Request with abort;
    elsif OK(Line_B) then
      requeue LB.Request with abort;
    else
      requeue LC.Request with abort;
    end if;
  end Send;
```

Router Example III

```
  procedure Overloaded(Line : Line_Id) is
  begin
    OK(Line) := False;
  end Overloaded;

  procedure Clear(Line : Line_Id) is
  begin
    OK(Line) := True;
  end Clear;
end Router;
```

Router Example IV

```
task body Line_Controller is
  …
begin
  loop
    select
      accept Request ( ... ) do
        -- service request
      end Request;
    or
      terminate;
    end select;
    -- housekeeping including possibly
    Router.Overloaded(Id)
    -- or
    Router.Clear(Id);
  end loop;
end Line_Controller;
```

Real-Time Resource Controller

```
protected Resource_Controller is
   entry Allocate(R: out Resource;
                  Amount : Request_Range);
   procedure Release(R: Resource;
                     Amount : Request_Range);
private
   Free : Request_Range := ...;
   Queued : Natural := 0;
end Resource_Controller;
```

Real-Time Resource Controller II

```
protected body Resource_Controller is
  entry Allocate( ... ) when Free > 0 and
                        Queued /= Allocate'Count is
  begin
    if Amount < Free then
      Free := Free - Amount;
      Queued := 0;
    else
      Queued := Allocate'Count + 1;
      requeue Allocate;
    end if;
  end Allocate;
```

Queue is priority ordered

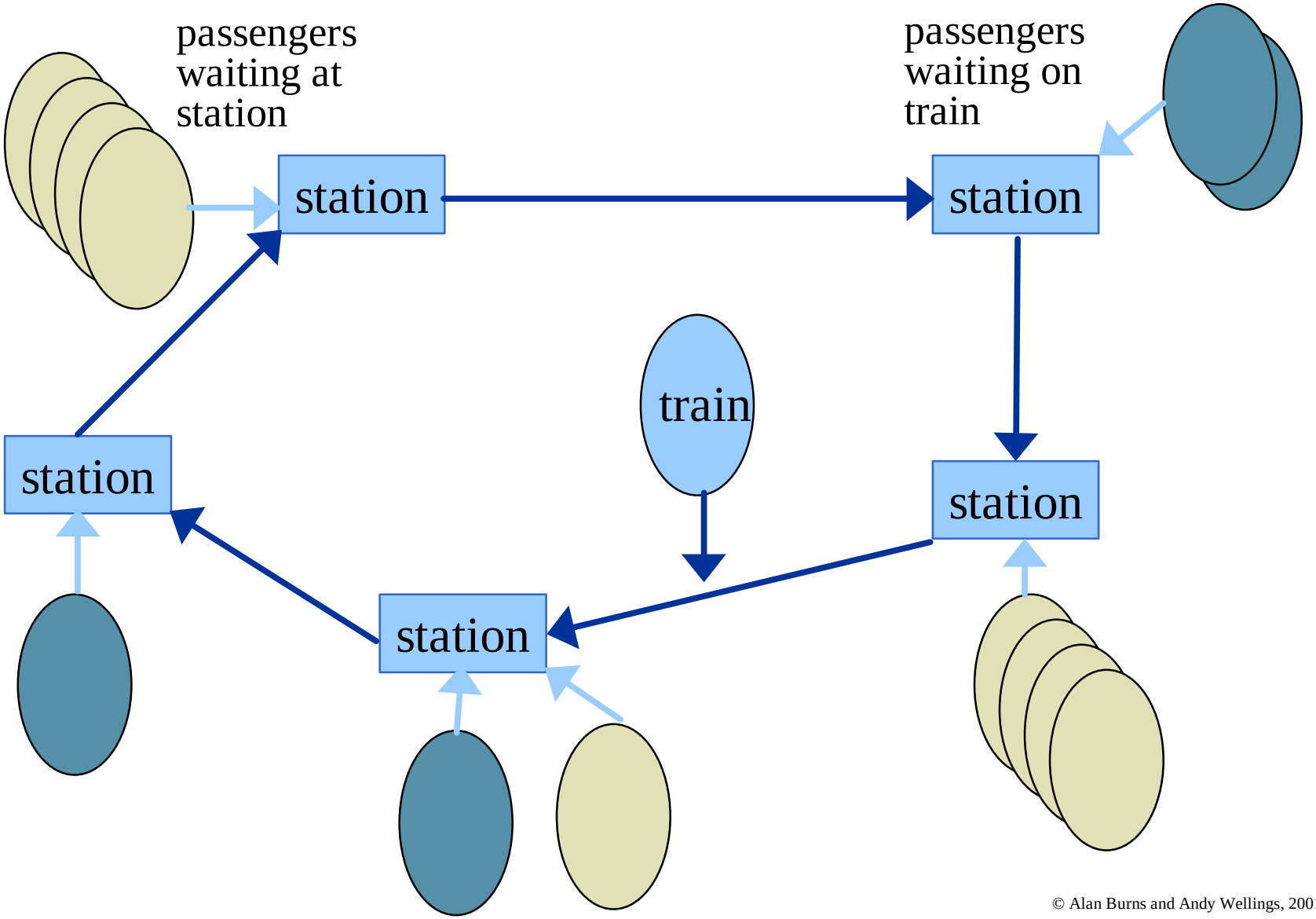
Real-Time Resource Controller III

```
    procedure Release (...) is
    begin
      Free := Free + Amount;
      Queued := 0;
    end Release;
end Resource_Controller;
```

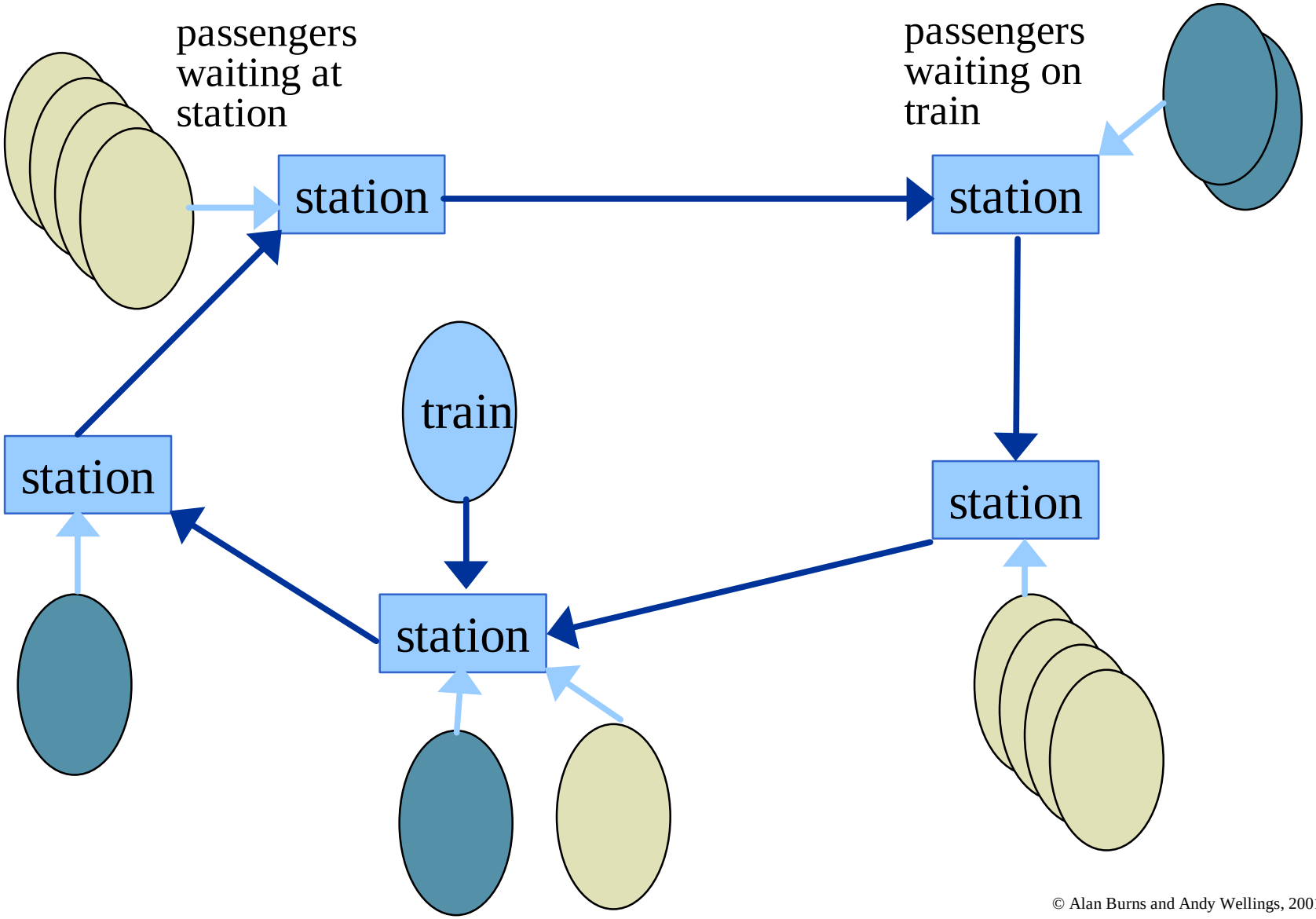
An Extended Example

- Consider the problem of simulating the behaviour of travellers on a circular railway (metro)
- There are N stations and one train (with a small finite capacity)
- Travellers arrive at one station and are transported to their requested destination
- Each station can be represented by a protected object, the train and travellers are tasks
- A passenger calls an entry at one station to catch the train, when the train arrives the passenger is requeued to another station and eventually released when the train arrives at that station

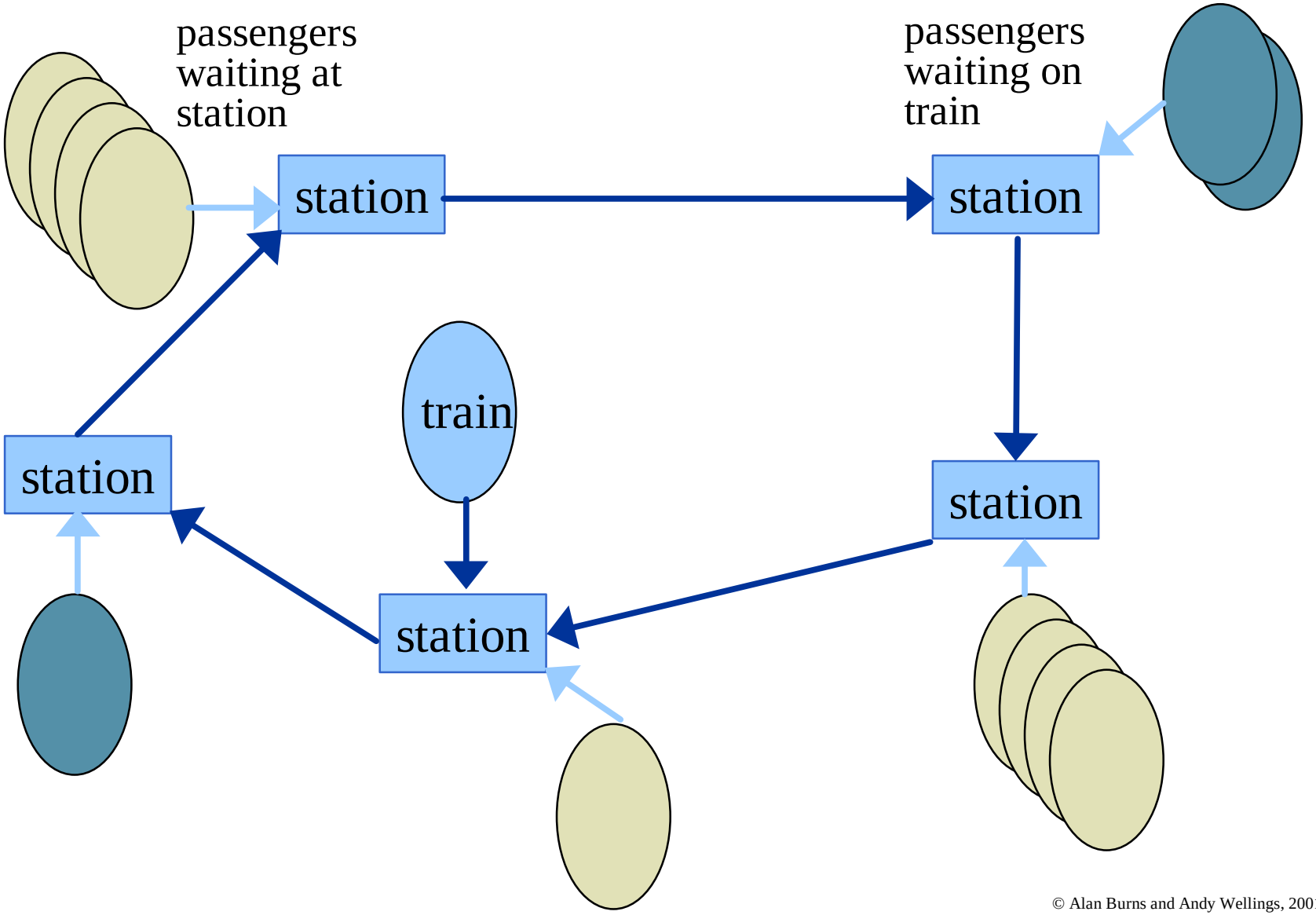
The Requeue Metro



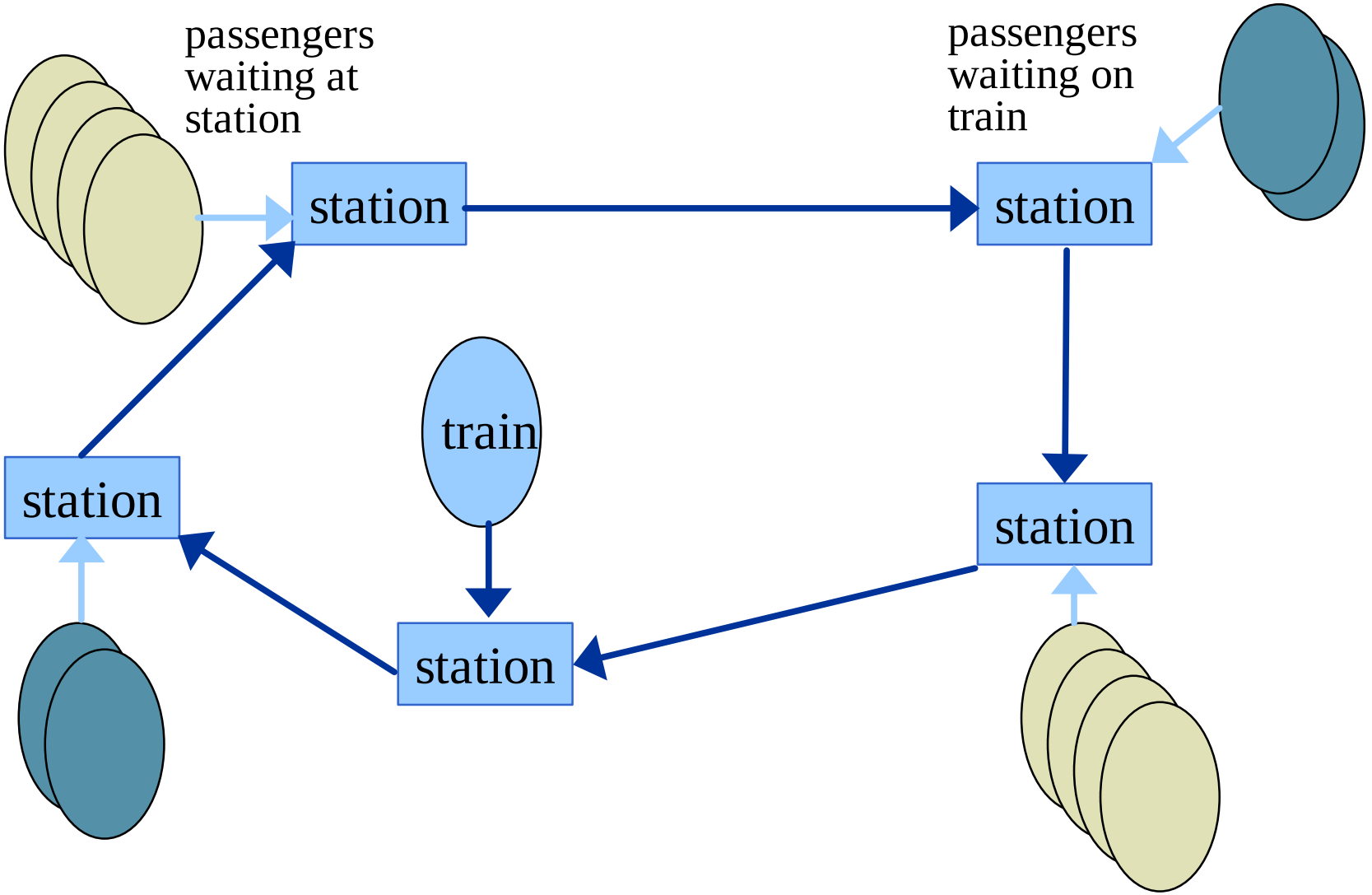
The Requeue Metro



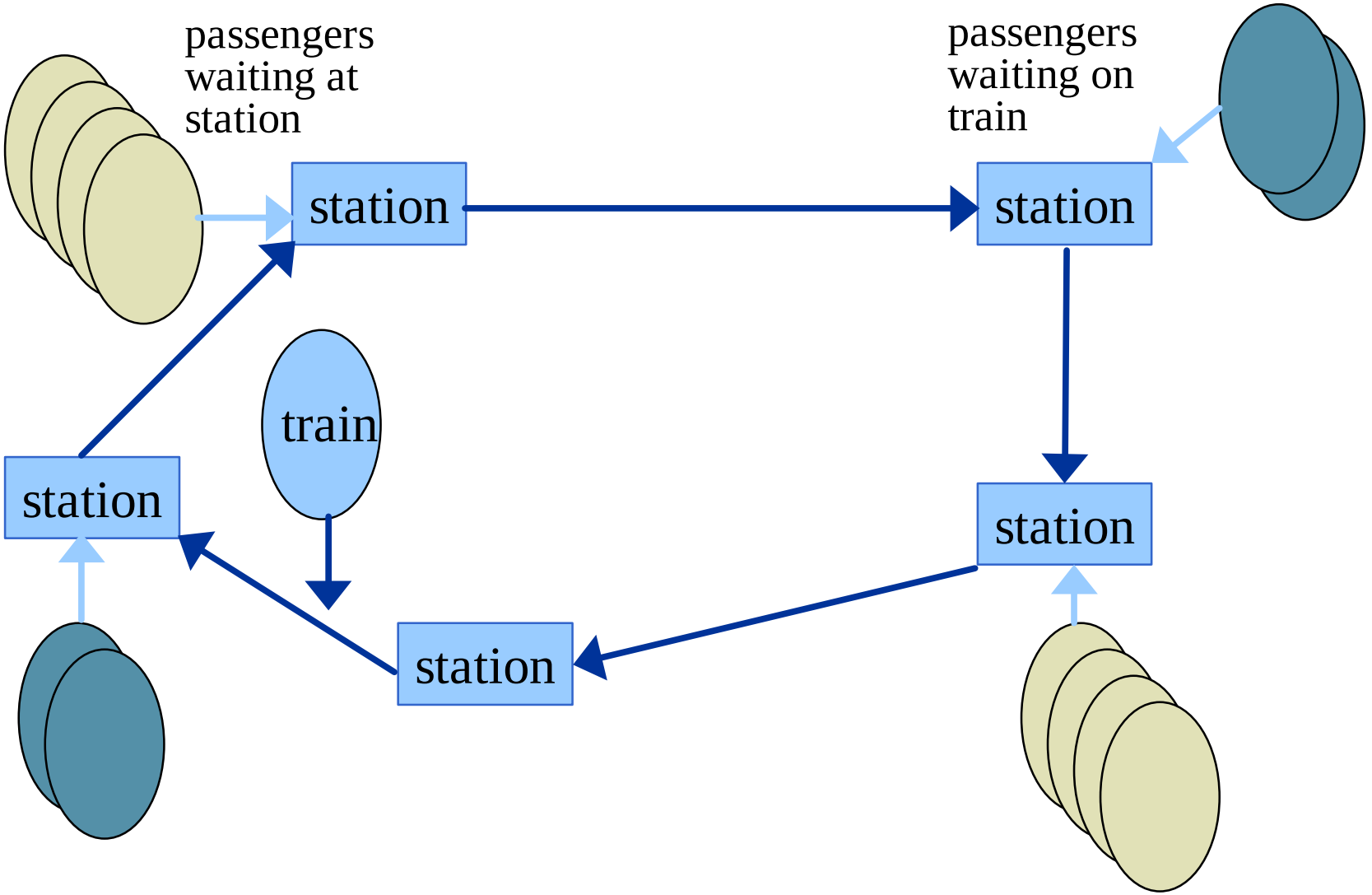
The Requeue Metro



The Requeue Metro



The Requeue Metro



Metro

```
type Station_Address is range 1 .. N;
type Passengers is range 0..Max;

Capacity : constant Passenger := ...;

protected type Station is
   entry Arrive(Destination : Station_Address);
   procedure Stopping(P : Passengers);
   entry Alight(Destination : Station_Address);
   procedure Boarding;
   procedure Closedoors(P : out Passengers);
private
   On_Train : Passengers;
   Trainboarding : Boolean := False;
   Trainstopped : Boolean := False;
end Station;

Stations : array(Station_Address) of Station;
```

Metro II

```
protected body Station is
  entry Arrive(Destination : Station_Address)
    when Trainboarding and then
         On_Train < Capacity is
  begin
    On_Train := On_Train + 1;
    requeue Stations(Destination).Alight;
  end Arrive;
  procedure Stopping(P: Passengers) is
  begin
    On_Train := P;
    Trainstopped := True;
  end Stopping;
```

Metro III

```
   entry Alight(Destination : Station_Address) is
      when Trainstopped is
   begin
     On_Train := On_Train - 1;
   end Alight;
   procedure Boarding is
   begin
     Trainstopped := False;
     Trainboarding := True;
   end Boarding;
   procedure Closedoors(P: out Passengers) is
   begin
     P := On_Train;
     Trainboarding := False;
   end Closedoors;
end Stations;
```

Metro IV

```
task type Client;
task body Client is
  Home, Away : Station_Address;
begin
  -- choose Home
  loop
    -- choose Away
    Stations(Home).Arrive(Away);
    Home := Away;
  end loop;
end Client;
```

Metro V

```
task Train;
task body Train is
  Volume : Passengers := 0;
  Travel_Times : array(Station_Addresses) := ...;
begin
  loop
    for S in Station_Address loop
      Stations(S).Stopping(Volume);
      Stations(S).Boarding;
      Stations(S).Closedoors(Volume);
      delay Travel_Times(S);
    end loop;
  end loop;
end Train;
```

Summary

- Algorithms are required which manage the resource allocation procedures and which guarantee that resources are allocated according to a predefined behaviour
- They are also responsible for ensuring that processes cannot deadlock
- The synchronization facilities provided by a real-time language must have sufficient expressive power to allow a wide range of synchronization constraints to be specified.
  - the type of service request
  - the order in which requests arrive
  - the state of the server and any objects it manages
  - the parameters of a request
  - the priority of the client

Summary

- Monitors (with condition synchronization) deal well with request parameters
- Avoidance synchronization in message-based servers or protected objects cope adequately with request types
- If insufficient expressive power, processes are often forced into a double interaction with a resource manager
- This must be performed as an atomic action, otherwise the client process may be aborted between the interactions
- Requeuing extends the expressive power of avoidance synchronization
- Client tasks can be requeued within the same server/ protected object or across servers/protected objects