# 2.5 Resources

## Representing resources

The OS maintains numerous resources that processes can request and release at runtime. Ex: Hardware resources include keyboards, pointing devices, printers, disk and tape drives, cameras, speakers, and many other types of devices. Software resources include input/output buffers, locks on database tables, messages, timers, and many others.

Most resources can be represented and handled uniformly using the same data structure and operations. Similar to a PCB, a ***resource control block (RCB)*** is a data structure that represents a resource. The specific implementation and the contents of the RCB vary between different OSs but the following is a generic structure that can handle many types of resources:

Table 2.5.1: A generic RCB of resource r.

| RCB field | Explanation |
|---|---|
| resource_description | Contains the description of r's properties and capabilities. |
| state | Shows the current availability of r. If r is a single-unit resource then this field indicates whether r is currently free or allocated to some process. If r has multiple identical units, such as a pool of identical buffers, then this field keeps track of how many units are currently free and how many are allocated. |
| waiting_list | When a process requests r and r is currently unavailable then the process's PCB is removed from the ready list and added to r's waiting list. The process is moved back to the ready list when r becomes available and is allocated to the process. |

Feedback?

**PARTICIPATION ACTIVITY** | 2.5.1: RCB implementation. ✓

1) With a single-unit resource, the state is just a Boolean.

[✓ True]

False

**Correct** ✓

The Boolean indicates whether the resource is free or allocated.

2) A waiting list can be implemented as a linked list where each element points to a PCB.

[✓ True]

False

**Correct** ✓

The waiting list can be a linked list implemented outside of the RCB. But to avoid dynamic memory management, the links can also be distributed over the RCBs, similar to distributing child pointers over PCBs.

Feedback?

**PARTICIPATION ACTIVITY** | 2.5.2: Organization of RCBs. ✓

The set of all RCBs can be implemented in different ways, depending on whether the number of RCBs is fixed or variable, and whether all RCB fields can be represented by only integers. Find the best implementation for each case.

Select the definition that matches each term

1) 2-D array of integers

● # of RCBs is fixed, all fields contain only integers

**Correct**

All RCBs can be pre-allocated and all fields can be represented by integers. Thus a 2-D array fixed in both dimensions is sufficient.

2) Linked list of integer arrays

● # of RCBs is variable, all fields contain only integers

**Correct**

RCBs are allocated and freed dynamically and thus a linked list of RCBs is necessary. All fields can be represented by integers and thus each RCB is a 1-D array.

3) Array of structures

● # of RCBs is fixed, some fields contain values other than integers

**Correct**

All RCBs can be pre-allocated and thus a 1-D array of RCBs is sufficient. Some RCB fields are not integers and thus each RCB must be a structure.

4) Linked list of structures

● # of RCBs is variable, some fields contain values other than integers

**Correct**

RCBs are allocated and freed dynamically and thus a linked list of RCBs is necessary. Some fields are not integers and thus each RCB must be a structure.

Reset

Feedback?

## Requesting a resource

A resource is ***allocated*** to a process if the process has access to and is able to utilize the resource. A resource is ***free*** if the resource may be allocated to a requesting process.

When the currently running process needs a resource, the process invokes a request() function. The ***request resource function*** allocates a resource r to a process p or blocks p if r is currently allocated to another process.

- If r is currently free, the state of r is changed to allocated and a pointer to r is inserted into the list of other_resources of p.
- If r is not free, the calling process p is blocked. p's PCB is moved from the RL to the waiting_list of r. Since the calling process, p, is now blocked, the scheduler must be called to select another process to run.

Figure 2.5.1: The request resource function.

```
request(r) {
   if (r.state == free) {
      r.state = allocated
      Insert r into self.other_resources
   }
   else {
      self.state = blocked
      Move self from RL to r.waiting_list
      scheduler()
   }
}
```

Feedback?

PARTICIPATION ACTIVITY | 2.5.3: Implementing request.

1) A process can be blocked on more than one resource.

○ True
● False

**Correct**

When a process requests a resource that is not available, the process is blocked. The process will not be able to call request() again until the first resource becomes available and the process is allowed to run again.

2) A request can be initiated by any process, regardless of the process's current state.

○ True
● False

**Correct**

Only a running process can initiate any actions.

3) Instead of blocking the calling process when the resource is not available, the function could return immediately and let the process continue executing.

● True
○ False

**Correct**

Such a call is generally known as a non-blocking call. The function returns a value to let the process know whether the resource has been allocated or not and to decide how to proceed.

Feedback?

## Releasing a resource

When the currently running process, p, wishes to release a resource r, the process invokes a release() function. The ***release resource function*** allocates the resource r to the next process on the r's waiting list. If the waiting list is empty, r is marked as free.

- If r's waiting_list contains no processes then the state of r is changed to free and p continues executing.
- If the waiting list of r is not empty then the process q at the head of the list is allocated r, the state of q is changed to ready, and q is moved from the waiting_list to RL. Since a new process (q) is now on RL, the scheduler must be called to decide which process (p or q) should continue to run.

Figure 2.5.2: The release resource function.

```
release(r) {
   Remove r from self.other_resources
   if (r.waiting_list == empty)
      r.state = free
   else
      Remove process q from the head of r.waiting_list
      Insert r into q.other_resources
      q.process_state = ready
      Move q from r.waiting_list to RL
      scheduler()
}
```

Feedback?

PARTICIPATION ACTIVITY | 2.5.4: Number of processes reactivated by release.

1) The release function always reactivates ________.

**Correct**

exactly one waiting process
zero or more processes
[✓ zero or one process]

If the waiting list is not empty then the next process is reactivated, otherwise no process is reactivated.

Feedback?

PARTICIPATION ACTIVITY | 2.5.5: Release with multiple-unit resources.

1) Assume the release function is modified to handle resources with multiple identical units. Each call to release could reactivate ________.

[✓ zero or more processes]
zero or one process

**Correct**

Release could reactivate any number of processes.
Ex: 3 units are released.
Case 1: Process at the head of waiting_list needs 4 units -- no process is reactivated.
Case 2: Process at the head of waiting_list needs 3 units -- one process is reactivated.
Case 3: The next 3 processes on waiting_list need 1 unit each -- all 3 processes are reactivated.

Feedback?

PARTICIPATION ACTIVITY | 2.5.6: Implementing release.

1) The release function may cause the calling process to stop running. To avoid penalizing the process for releasing a resource, a fairer approach would be not to call the scheduler as part of the release.

True
[✓ False]

**Correct**

The process waiting for the resource could have a higher priority than the running process and so should run as soon as the resource becomes available.

Feedback?

## The scheduler

The ***scheduler function*** determines which process should run next and starts the process. The scheduler function is called at the end of each of the process and resource management functions: create, destroy, request, release.

Assuming the RL is implemented as a priority list, the scheduler() function performs the following tasks:

1. Find the highest priority process q on the RL.
2. Perform a context switch from p to q if either of the following conditions is met:
   - The priority of the running process p is less than the priority of q. This condition can be true when scheduler() is called from create() or from release(). In both cases the process q could have a higher priority than p.
   - The state of the running process p is blocked. This condition is true when scheduler() is called from request() and the resource is not available. The request function changes the status of p to blocked, but the process continues executing the request and scheduler functions until the context switch is performed at the end.

Figure 2.5.3: The scheduler function.

```
scheduler() {
   Find highest priority process q on RL
   if (p.priority < q.priority) or (p.process_state == blocked)
      perform context switch from p to q
}
```

Feedback?

PARTICIPATION ACTIVITY | 2.5.7: Possibility of context switch. ✓

The calling process p could be preempted by some other process q when scheduler() is called from ________.

1) create()
- [✓ True]
- False

**Correct**

p could create q with a higher priority. ✓

2) destroy()
- [✓ True]
- False

**Correct**

destroy() could free up a resource r held by a destroyed process. If a process q with priority higher than p is blocked on r, then q could preempt p. ✓

3) request()
- [✓ True]
- False

**Correct**

If p blocks then some other process q is resumed. ✓

4) release()
- [✓ True]
- False

**Correct**

release() could unblock a process q with a higher priority than p. ✓

Feedback?

PARTICIPATION ACTIVITY | 2.5.8: Scheduler with a single-priority RL. ✓

If RL is implemented as a single list without priority levels and every new process is entered at the tail of RL then the following functions do not need to call the scheduler:

1) create()
- [✓ True]
- False

**Correct**

The new process is inserted at the tail of RL. The calling process continues to run and create() does not need to call scheduler(). ✓

2) destroy()
- [✓ True]
- False

**Correct**

As long as the process cannot call destroy() on itself, the process will continue to run and scheduler() need not be called. ✓

3) request()
- True
- [✓ False]

**Correct**

When the resource is not available, the calling process is blocked and scheduler() must be called to select the next processes to run. ✓

4) release()

**Correct** ✓

[✓ True]
False

The released resource is marked as free or is allocated to a waiting process, which is entered at the tail of RL. The calling process continues to run and scheduler() need not be called.

Feedback?

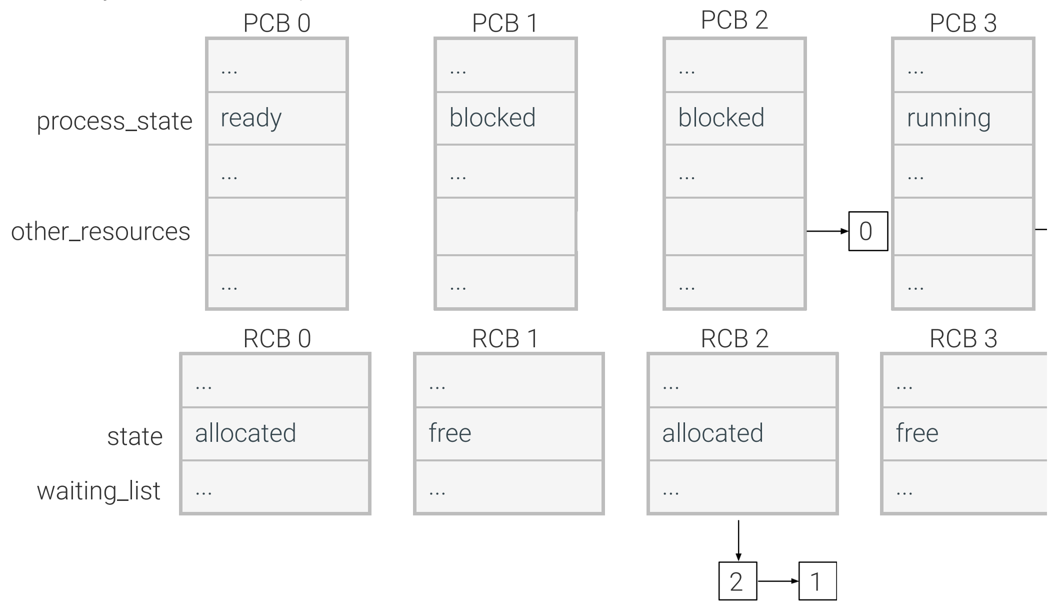
CHALLENGE ACTIVITY | 2.5.1: Resource handling.

The following diagram shows the process_state and other_resources fields of 4 processes in the PCB array. The RCB array shows the state and the waiting_list of 4 resources.

371440.2479476.qx3zqy7

Jump to level 1

The ready list RL contains processes 3 then 0.



Process 3 releases resource 2. After the operation:

| | | | |
|---|---|---|---|
| Process | 0 changes to ready | Resource 2 is | allocated |
| Process 3's resource list | contains resource 1 | Resource 2's waiting list | contains pr |
| Process 2's resource list | contains resource 2 | RL contains the processes | 3,0 |

1 2 3 4 **5**

Check Next

✗ Each incorrect answer is highlighted.
Expected:
Process 2 changes to ready
Process 3's resource list is empty
Process 2's resource list contains resource 2
Resource 2 is allocated
Resource 2's waiting list contains process 1
RL contains processes 3, 0, 2

The waiting list of resource 2 contains processes 2 and 1.

When process 3 releases resource 2, the resource is allocated to the first process, 2, on the wa is removed from the resource list of process 3 and is added to the resource list of process 2.

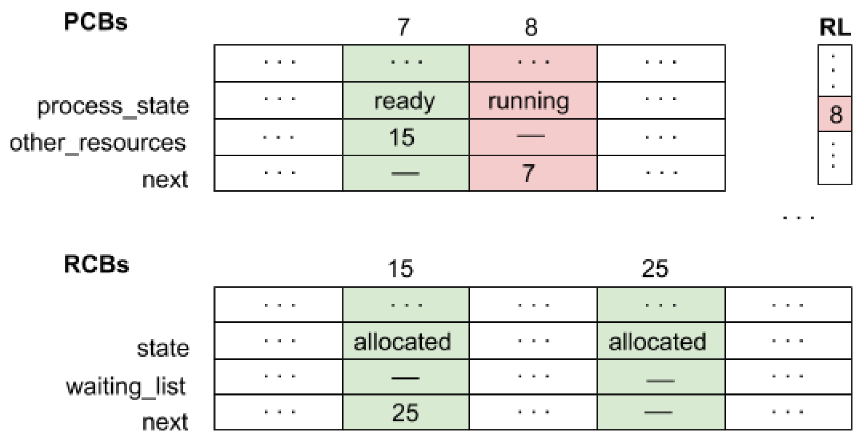
The waiting list of resource 2 contains the remaining process 1.

The state of process 2 changes from blocked to ready and the process is added to RL.

Feedback?

EXERCISE 2.5.1: Request and release functions in action.

- Two processes (7 and 8) are currently on RL. Process 8 is at the head of the list and is running. The next field of process 8 points to process 7, which is ready.
- Process 7 is currently holding two resources (15 and 25). The other_resources field of process 7 points to resource 15, which in turn points to the next resource, 25.



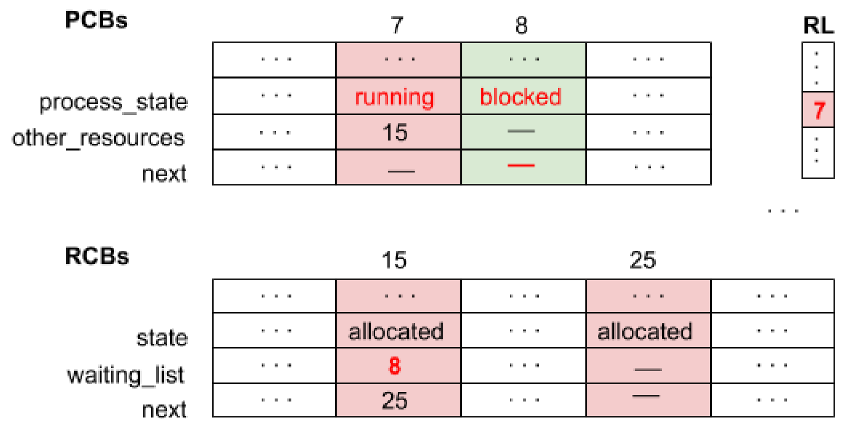
| | | 7 | 8 | |
|---|---|---|---|---|
| | ... | ... | ... | ... |
| process_state | ... | ready | running | ... |
| other_resources | ... | 15 | — | ... |
| next | ... | — | 7 | ... |

| | | 15 | | 25 | |
|---|---|---|---|---|---|
| | ... | ... | ... | ... | ... |
| state | ... | allocated | ... | allocated | ... |
| waiting_list | ... | — | ... | — | ... |
| next | ... | 25 | ... | — | ... |

(a) Show all changes to the data structures after process 8 requests resource 15.

**Solution**

- Process 8 is blocked on resource 15. Process 7 is running and is the only process on RL.



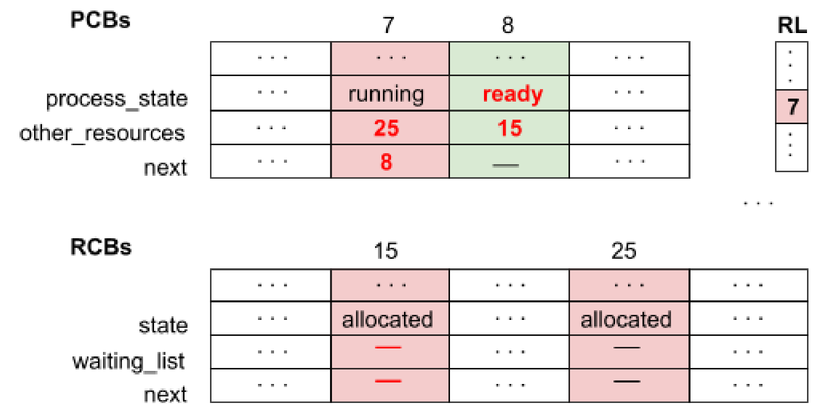
| | | 7 | 8 | |
|---|---|---|---|---|
| | ... | ... | ... | ... |
| process_state | ... | running | blocked | ... |
| other_resources | ... | 15 | — | ... |
| next | ... | — | — | ... |

| | | 15 | | 25 | |
|---|---|---|---|---|---|
| | ... | ... | ... | ... | ... |
| state | ... | allocated | ... | allocated | ... |
| waiting_list | ... | 8 | ... | — | ... |
| next | ... | 25 | ... | — | ... |

(b) Show all changes to the data structures after process 7 releases resource 15. Assume that processes are entered into RL at the end of the list.

**Solution**

- Process 8 gets resource 15, becomes ready, and is inserted on RL behind process 7. Process 7 continues running and holding resource 25.



Feedback?

EXERCISE 2.5.2: Effect of operations on processes and resources.

Consider a system of processes and resources and make the following assumptions:

- The Ready List (RL) has 3 priority levels, 0, 1, and 2 (0 being the lowest).
- The create function is simplified: create(n) creates a new child process with the priority n. The process is placed at the end of the corresponding RL queue.
- Only a single CPU exists. The currently running process continues running until the process:
  - blocks by requesting an unavailable resource, or
  - is interrupted by a function timeout(), which moves the process from the head of the current RL queue to the end of the same queue; the process currently at the head of the RL queue starts running
- The system has 3 resources, 0, 1, 2.
- At the start, only a single process, described by PCB 0, exists and is running at priority 2.

(a) Assume the following sequence of functions is issued (each function is executed by the currently running process). The create function always uses the next available PCB 1, 2, 3, ...:

1. create(1)
2. create(2)
3. timeout()
4. create(0)
5. timeout()
6. create(2)

Determine how many child processes each process has at the end of the above sequence.

**Solution**

1. Process 0 creates child process 1 at priority level 1. Process 0 has priority 2 and thus continues running.
2. Process 0 creates a second child process 2 at priority level 2. The new process goes behind process 2 on the RL and thus process 0 continues running.
3. The timeout function moves process 0 to the end of the queue. Process 2 is running.
4. Process 2 creates a child process 3 at level 0 and continues running.

5. The next timeout function moves process 2 to the end of the queue. Process 0 resumes.
6. Process 0 creates a third child process 4 at level 2 and continues running.

At this time, process 0 has 3 children and process 2 has 1 child. All other processes have 0 children.

(b) Assume that the above sequence continues as follows:

1. timeout()
2. request(1)
3. timeout()
4. request(2)
5. request(1)
6. timeout()
7. request(2)

Determine which processes are blocked at the end of the above sequence.

**Solution**

1. RL now contains the processes 0, 2, 4. The timeout function moves 0 to the end and thus 2 is running.
2. Resource 1 is free. Process 2 acquires resource 1 and continues.
3. The next timeout function moves process 2 to the end of the queue. Process 4 starts running.
4. Resource 2 is free. Process 4 acquires 2 and continues.
5. Resource 1 is already allocated. Thus process 4 is blocked on resource 1. Process 0 resumes.
6. The next timeout function moves process 0 to the end of the queue. Process 2 resumes.
7. Resource 2 is already allocated. Thus process 2 is blocked on 2. Process 0 is running.

At this time, processes 2 and 4 are blocked.

(c) Assume that the currently running process executes destroy(2). Determine:

1. The total number of processes in the system.
2. Which resources are free.

**Solution**

1. Process 2 has one child, 3, which is destroyed along with 2. Thus 3 processes (0, 1, and 4) remain in the system.
2. Since process 2 was destroyed, resource 1 is reallocated to process 4. Thus only resource 0 is free.

**Feedback?**

**EXERCISE** 2.5.3: Managing multi-unit resources.

(a) Extend the request and release functions to work with resources having multiple identical units.

- p.other_resources is a list of pairs (r, k) where r is a resource and k is the number of units that p is holding.
- r.state is a counter that keeps track of the currently available units of r.
- r.waiting_list contains pairs (p, k) where p is the waiting process and k is the number of requested units.
- The request function has the form request(r, k) where r is the resource and k is the number of units. To simplify the algorithm, a process may request units of the same resource only once.

- The release function has the form release(r) where r is the resource. All k units of r are released at the same time.
- Note that a release of k units may enable more than one process from r.waiting_list.

**Solution**

```
request(r, k){
   if (r.state >= k)
      r.state = r.state - k
      insert (r, k) into self.other_resources
   else
      self.state = blocked
      remove self from RL
      insert (self, k) in r.waiting_list
      scheduler()
}

release(r){
   remove (r, k) from self.other_resources
   r.state = r.state + k
   while (r.waiting_list != empty and r.state > 0)
      get next (p, k) from r.waiting_list
      if (r.state >= k)
         r.state = r.state - k
         insert (r, k) into p.other_resources
         p.process_state = ready
         remove (p, k) from r.waiting_list
         insert p in RL
      else break
   scheduler()
}
```

(b) A process could remain stuck in r.waiting_list forever if the number of units requested exceeds the total number of units available initially in r. How could that problem be prevented?

**Solution**

RCB must record the initial number of units in addition to the currently available number of units. Any request that exceeds the initial number must be rejected, rather than inserted into r.waiting_list.

(c) A process p could starve by being skipped over by processes arriving after p but requesting smaller numbers of units than p. How could that problem be prevented?

**Solution**

The loop in the release function must stop as soon as the request by the process p at the head of the queue cannot be satisfied. Any process q arriving after p is delayed even if q's request could be satisfied.

Feedback?

How was this section?

Provide feedback