LECTURE 15

MONITORS

SUBJECTS

Introduction to monitors

Comparison between monitors and semaphores

Condition variables

- Condition variables' operations

Monitor examples

MONITOR

A monitor is a set of routines which are protected by a mutual exclusion lock

- None of the routines in the monitor can be executed by a thread until that thread acquires the lock

Any other threads must wait for the thread that is currently executing to give up control of the lock

When a monitor is used, the competition synchronization code is added by the compiler

- Why is this an advantage?

MONITOR

A thread can actually suspend itself inside a monitor and then wait for an event to occur

- If this happens, then another thread is given the opportunity to enter the monitor

Usually, a thread suspends itself while waiting for a condition

- During the wait, the thread temporarily gives up its exclusive access
- It must reacquire it after the condition has been met.

MONITOR VS SEMAPHORE

uOttawa
L'Université canadienne
Canada's university

A semaphore is a simpler construct than a monitor because it's just a lock that protects a shared resource

- Not a set of routines like a monitor

An task must acquire (or wait for) a semaphore before accessing a shared resource

A task must simply call a routine (or procedure) in the monitor in order access a shared resource

- When done, you do not have to release anything
- Remember you have to release semaphores (if you forget ➔deadlock)

COMPETITION SYNCHRONIZATION

One of the most important features of monitors is that shared data is resident in the monitor

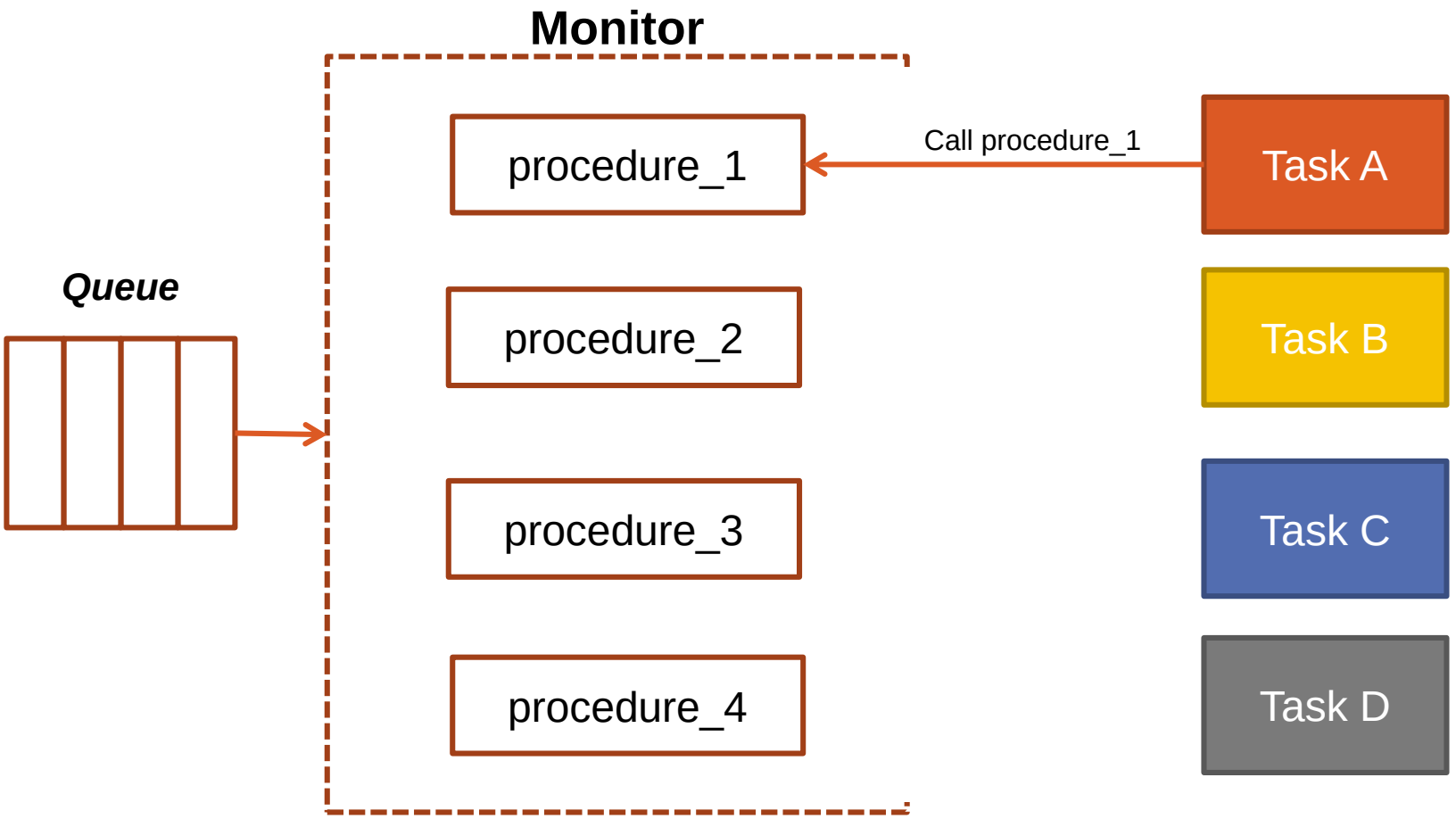
- All synchronization code is centralized in one location
- This is in contrast to being in the competing tasks

The monitor guarantees synchronization by allowing access to only one task at time

- Remember that using counting semaphores, we are able to allow multiple tasks access, not necessarily only one

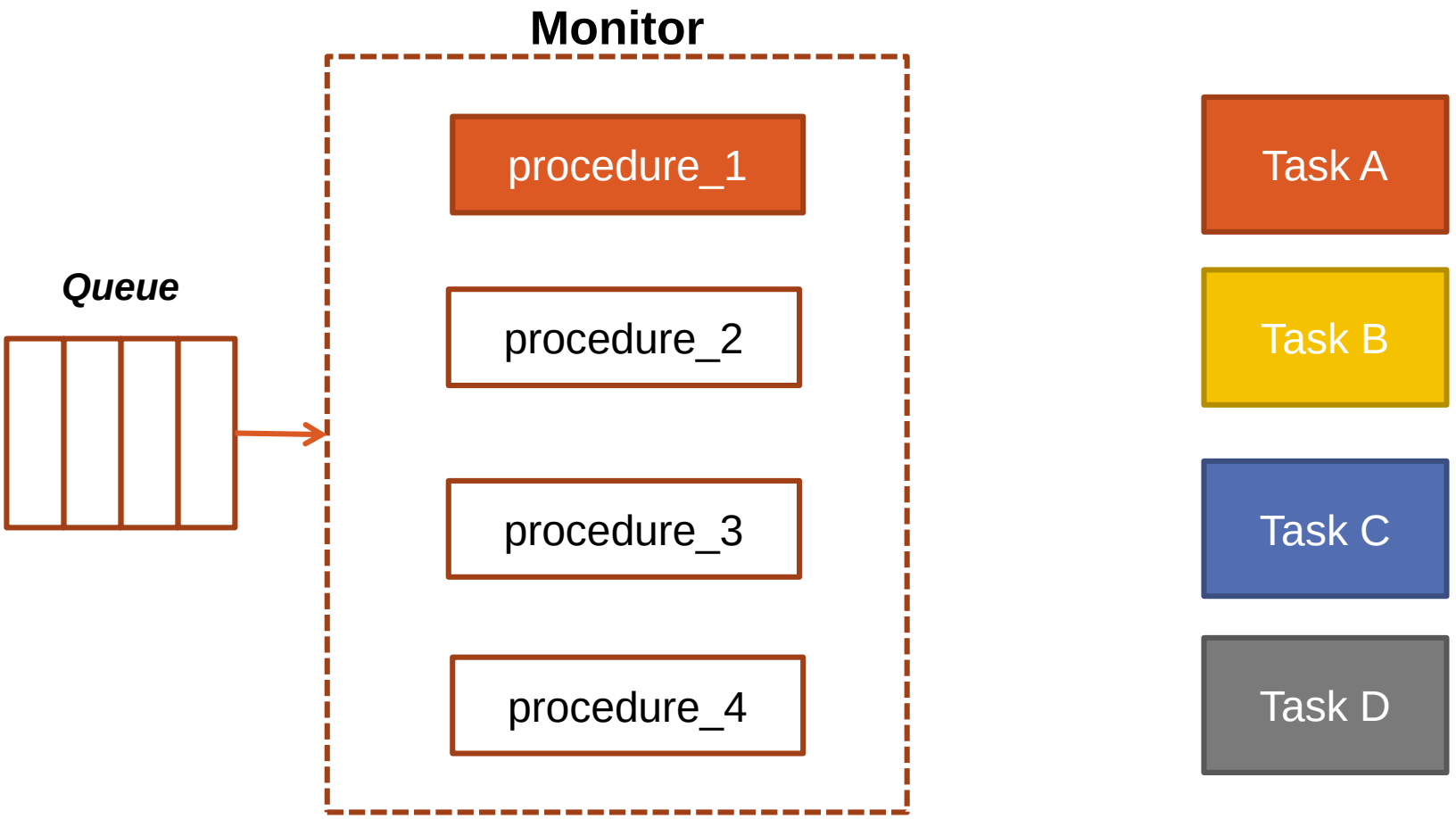
Calls to monitor procedures are implicitly queued if the monitor is busy at the time of the call

EXAMPLE OF USING MONITOR



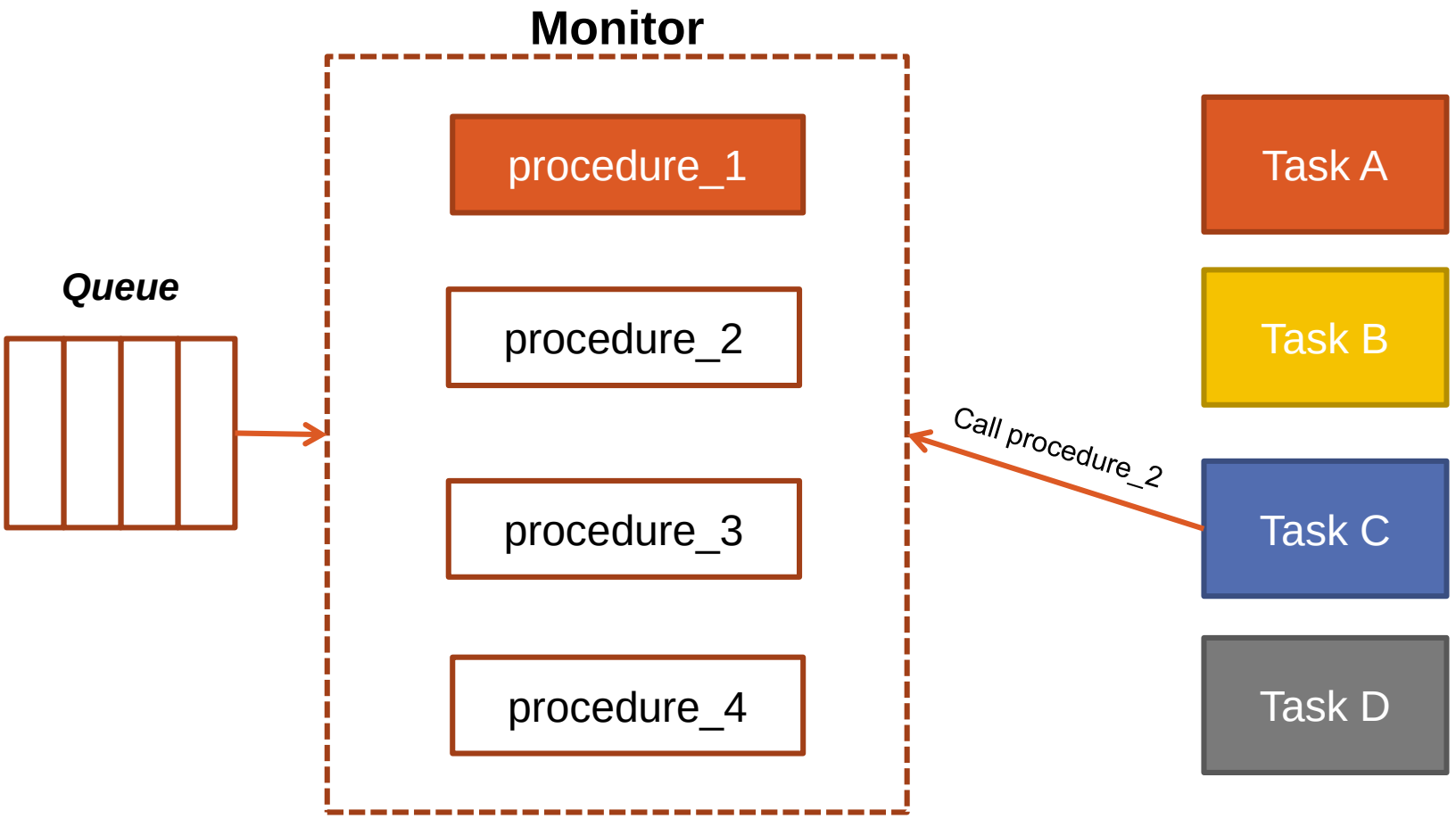
Monitor Owner: None

EXAMPLE OF USING MONITOR



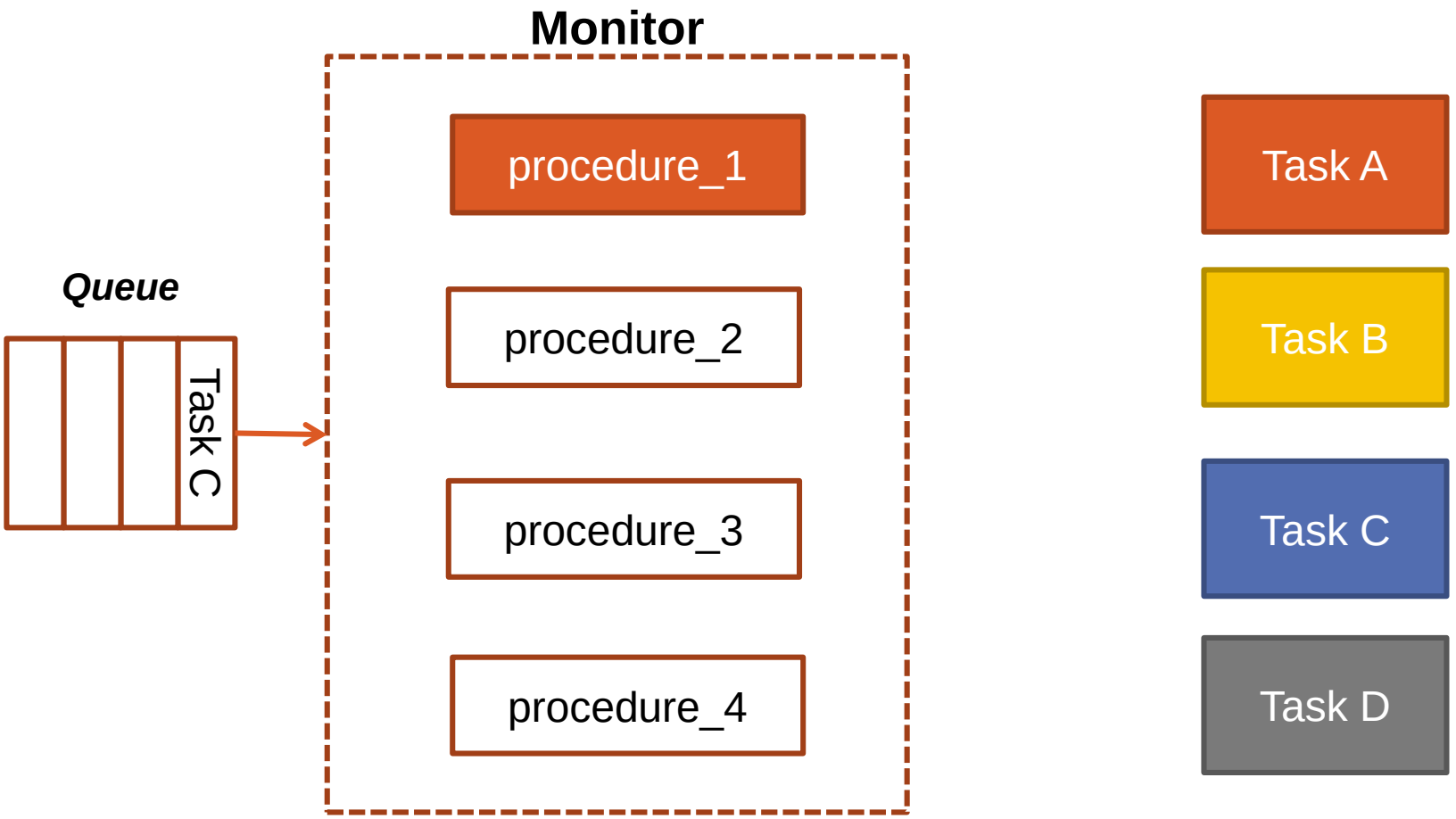
Monitor Owner: Task A

EXAMPLE OF USING MONITOR



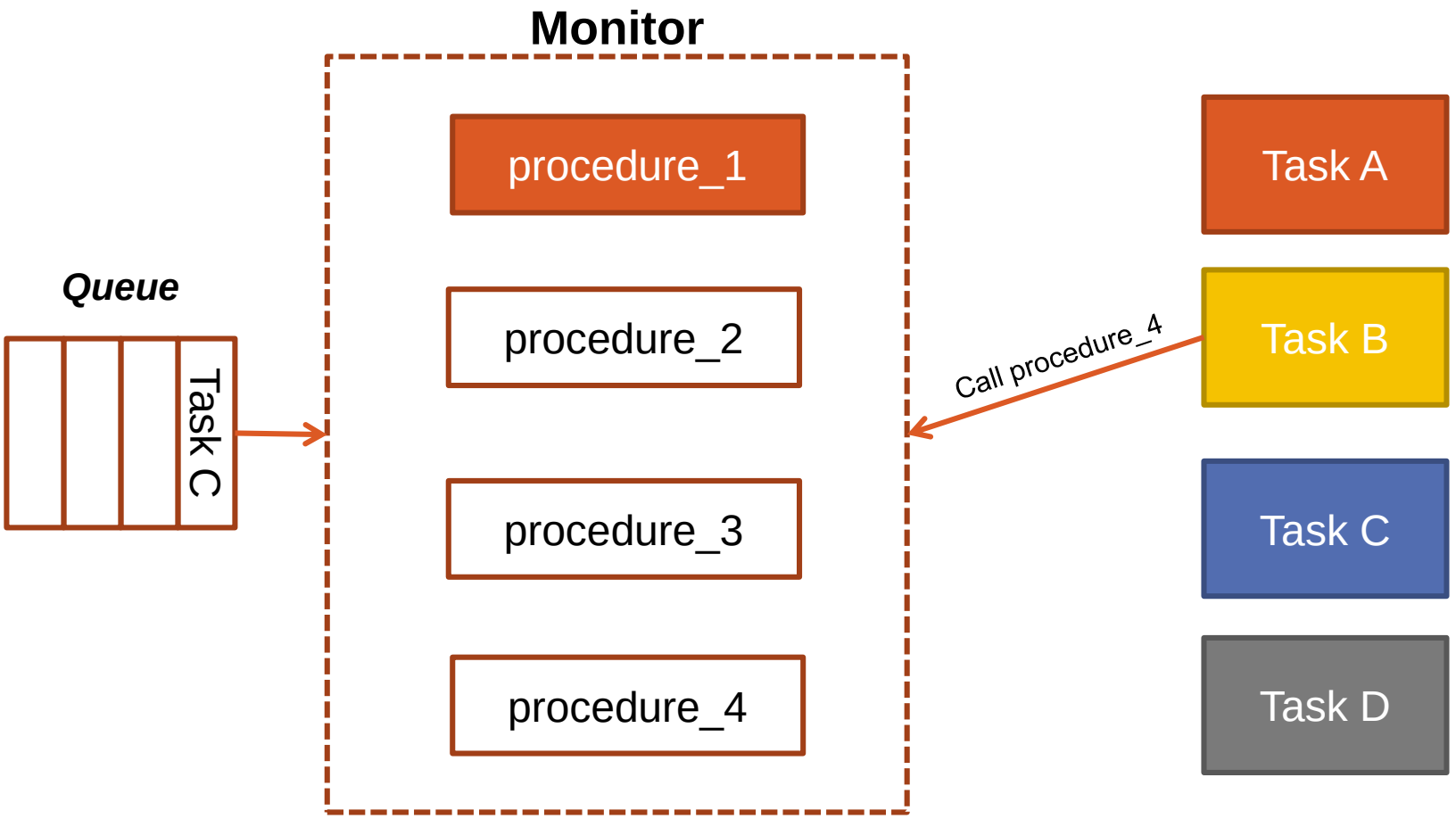
Monitor Owner: Task A

EXAMPLE OF USING MONITOR



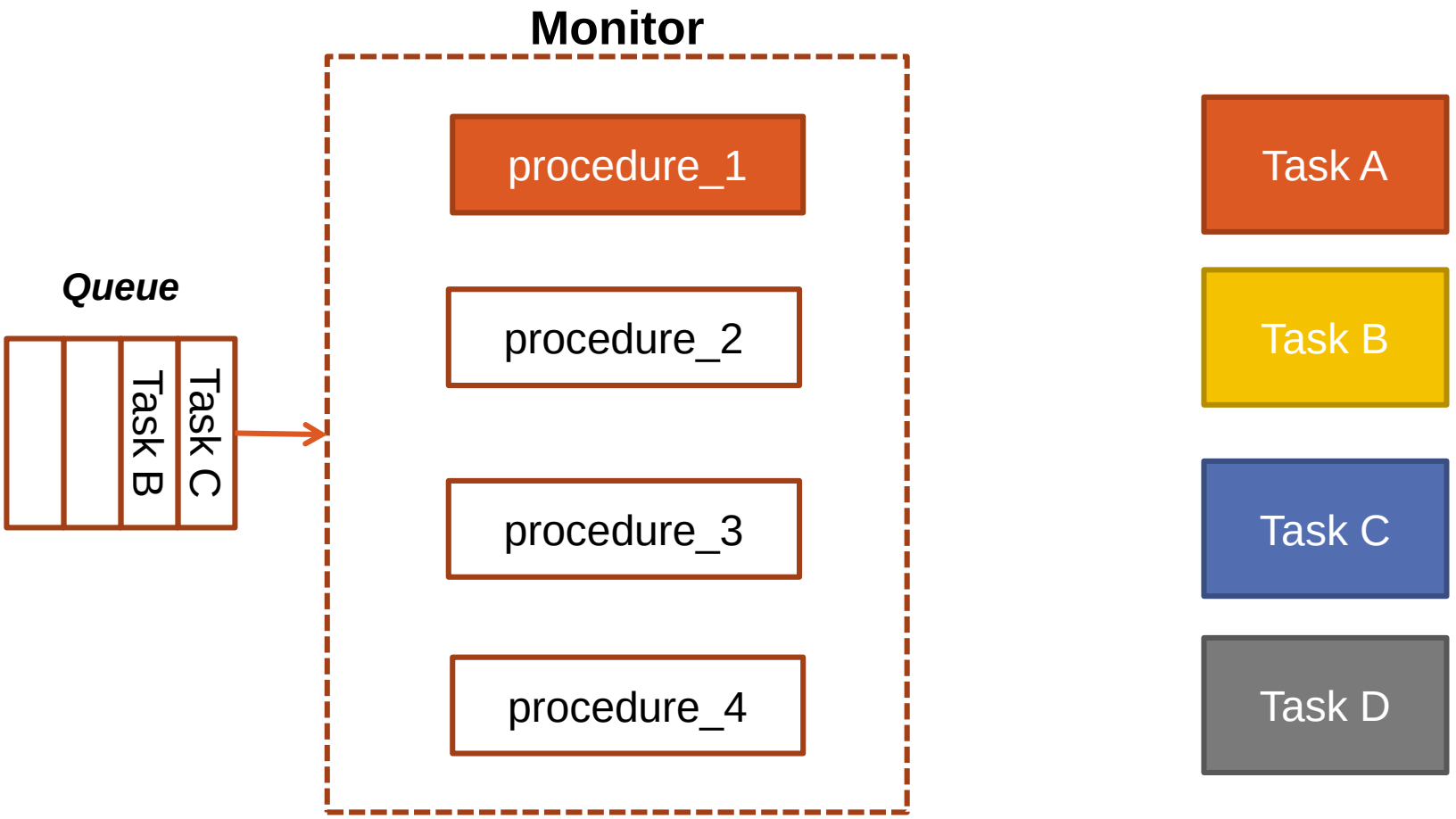
Monitor Owner: Task A

EXAMPLE OF USING MONITOR



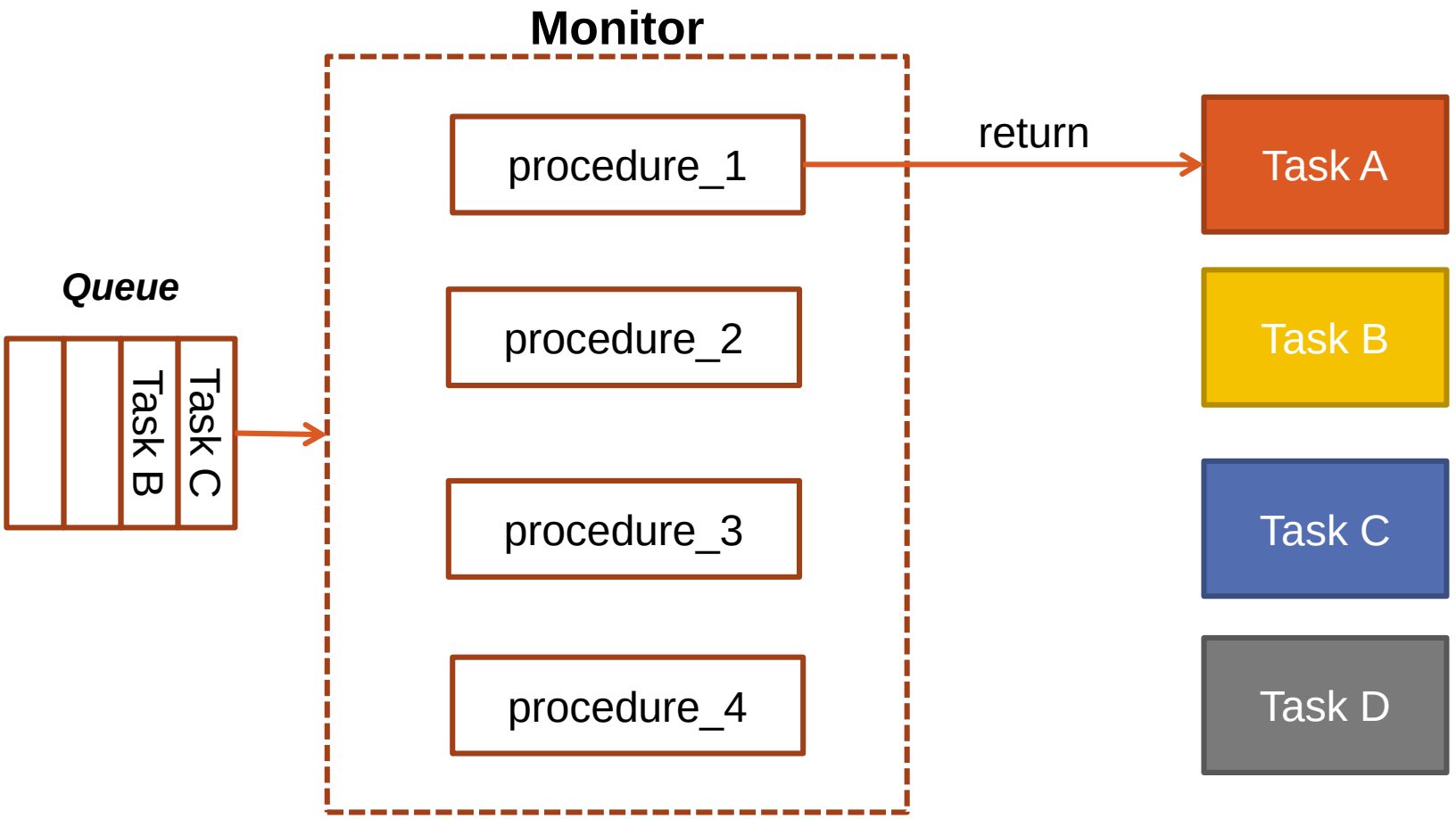
Monitor Owner: Task A

EXAMPLE OF USING MONITOR



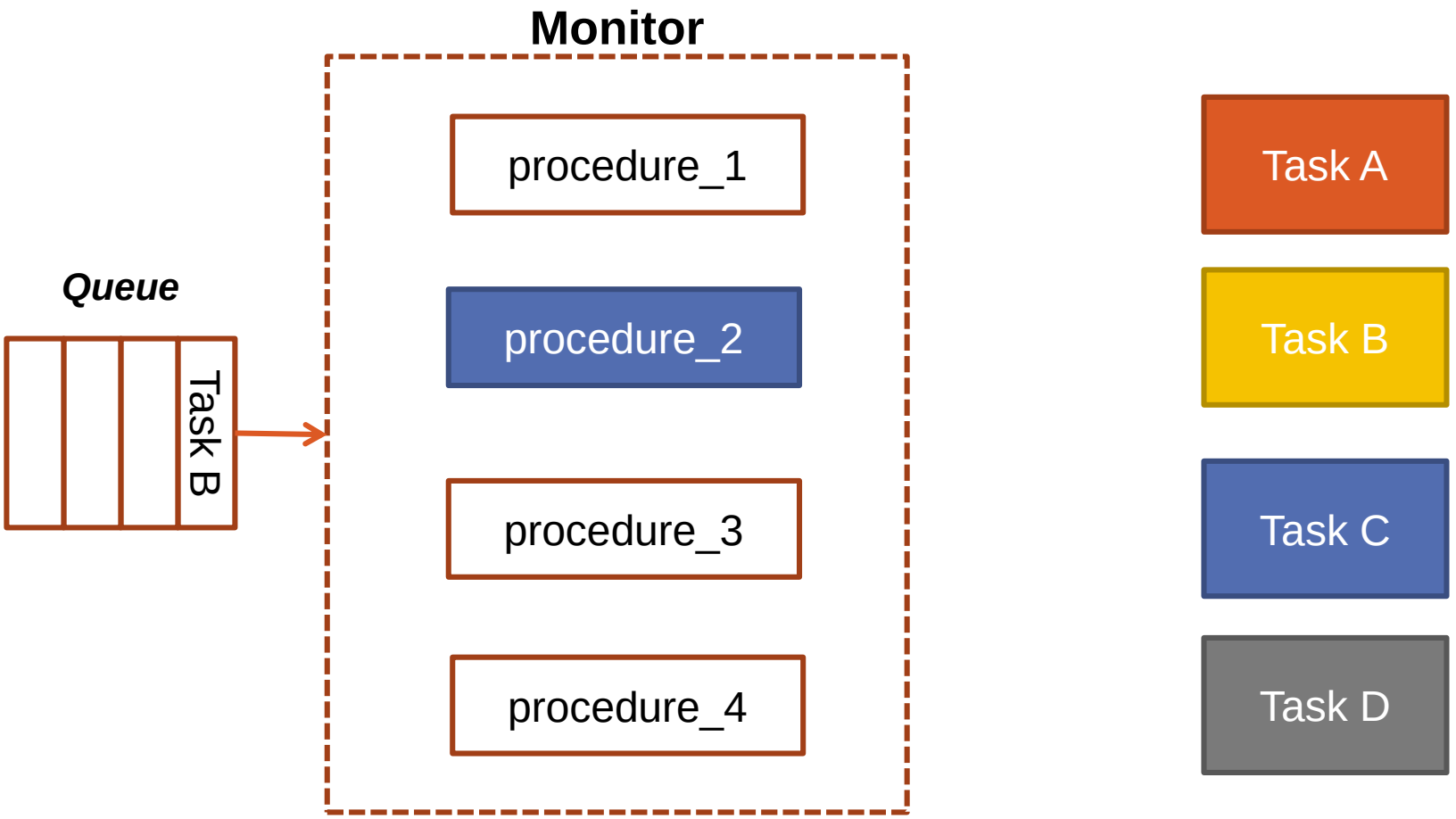
Monitor Owner: Task A

EXAMPLE OF USING MONITOR



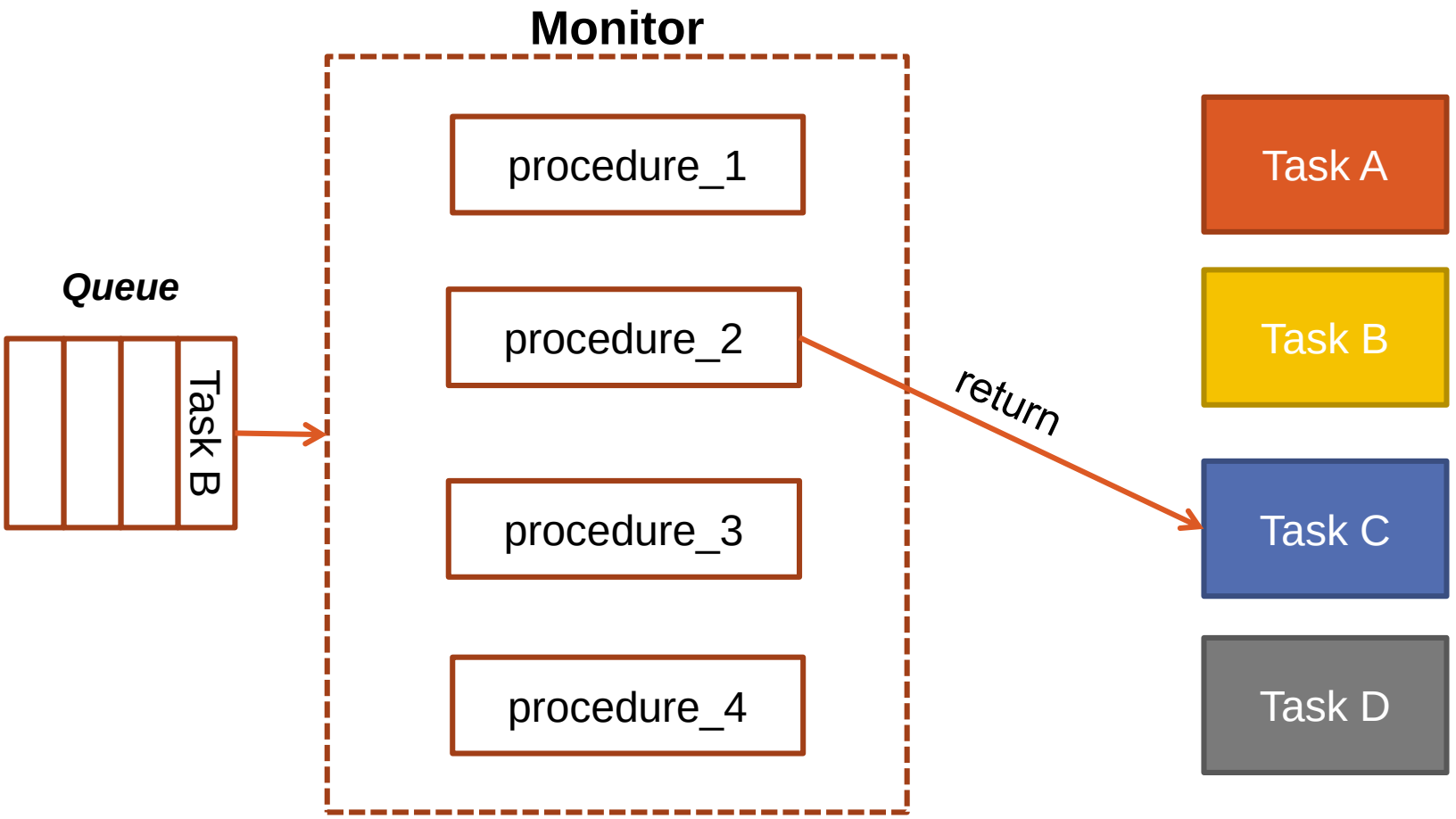
Monitor Owner: None

EXAMPLE OF USING MONITOR



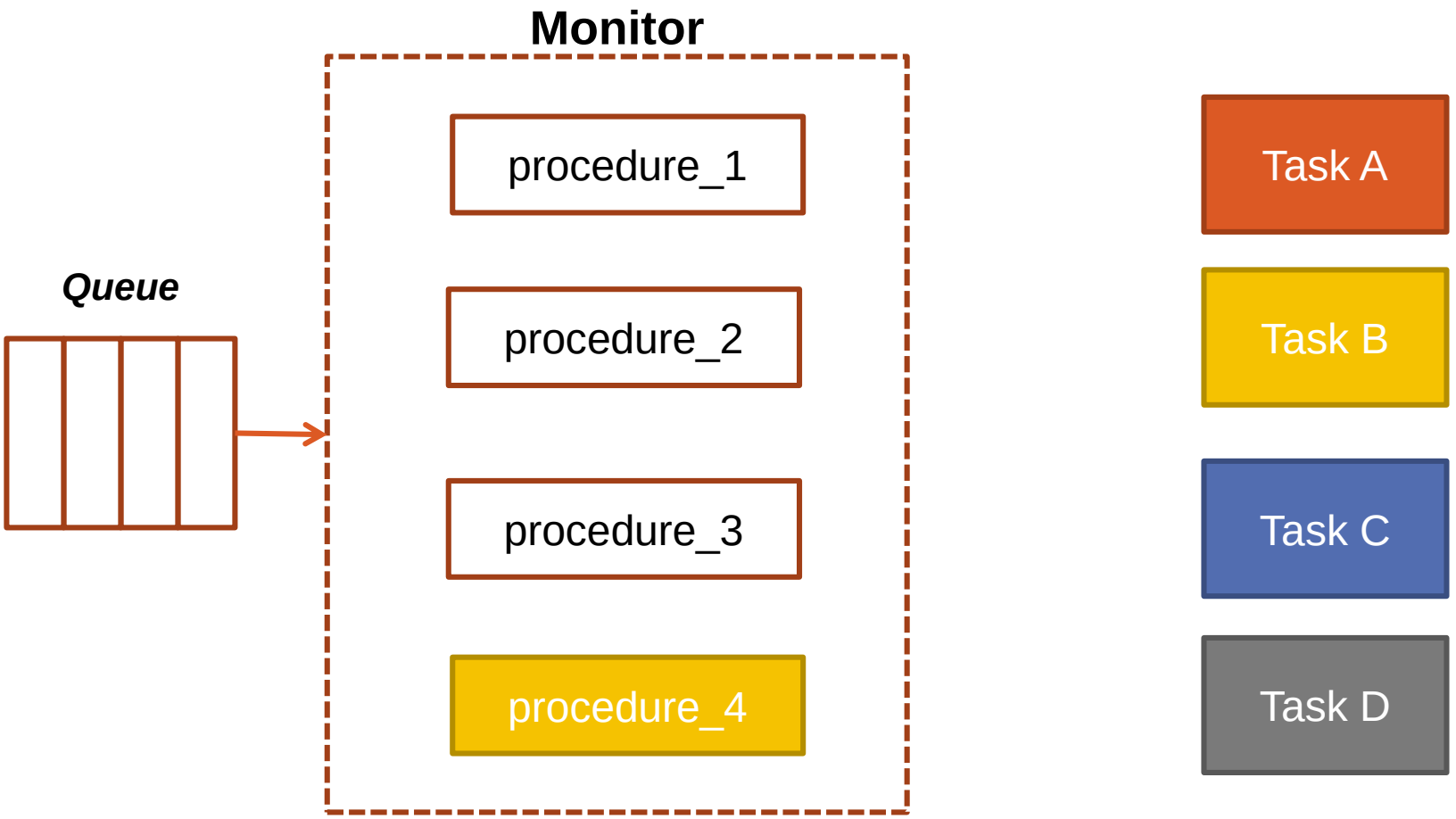
Monitor Owner: Task C

EXAMPLE OF USING MONITOR



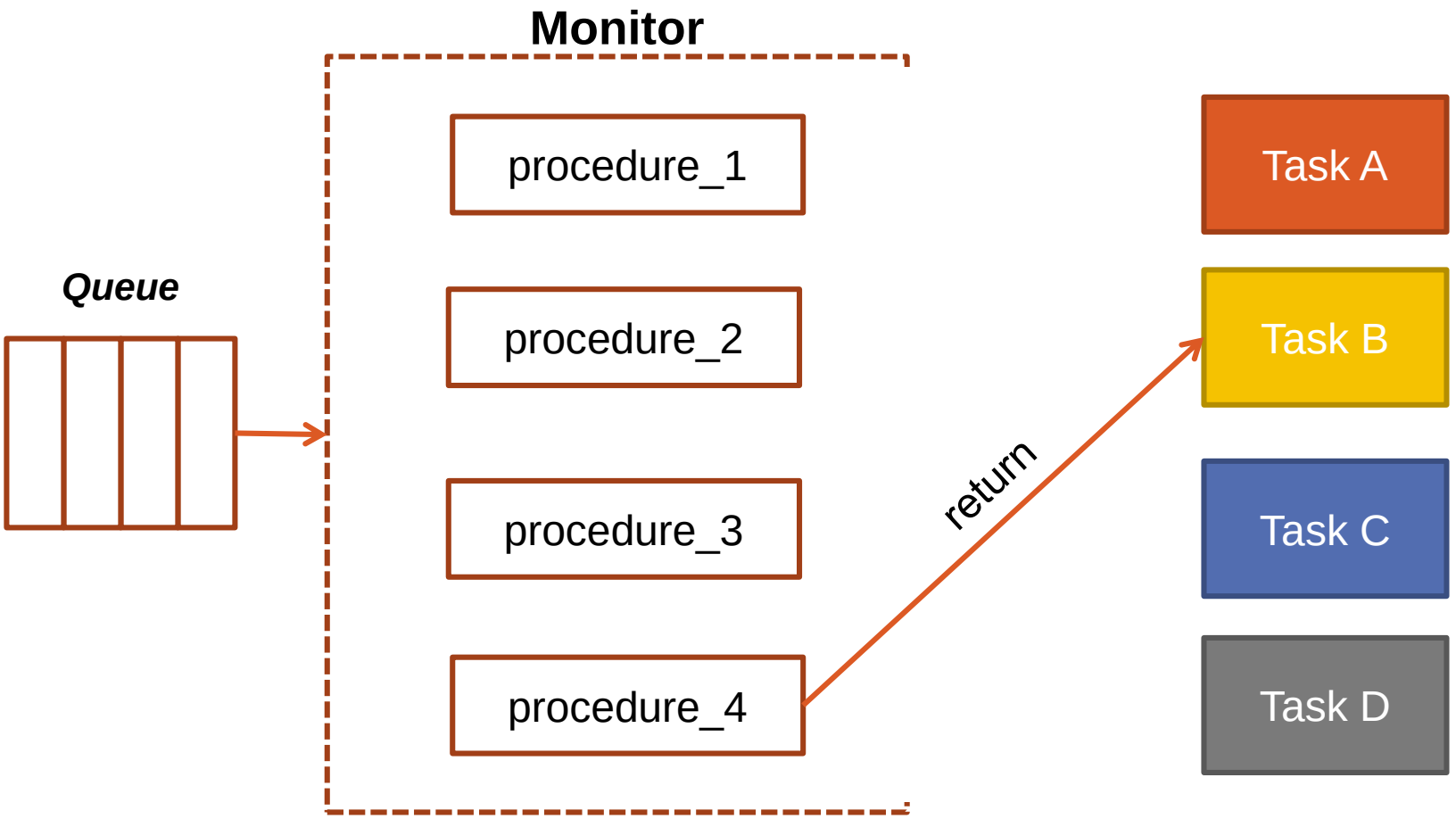
Monitor Owner: None

EXAMPLE OF USING MONITOR



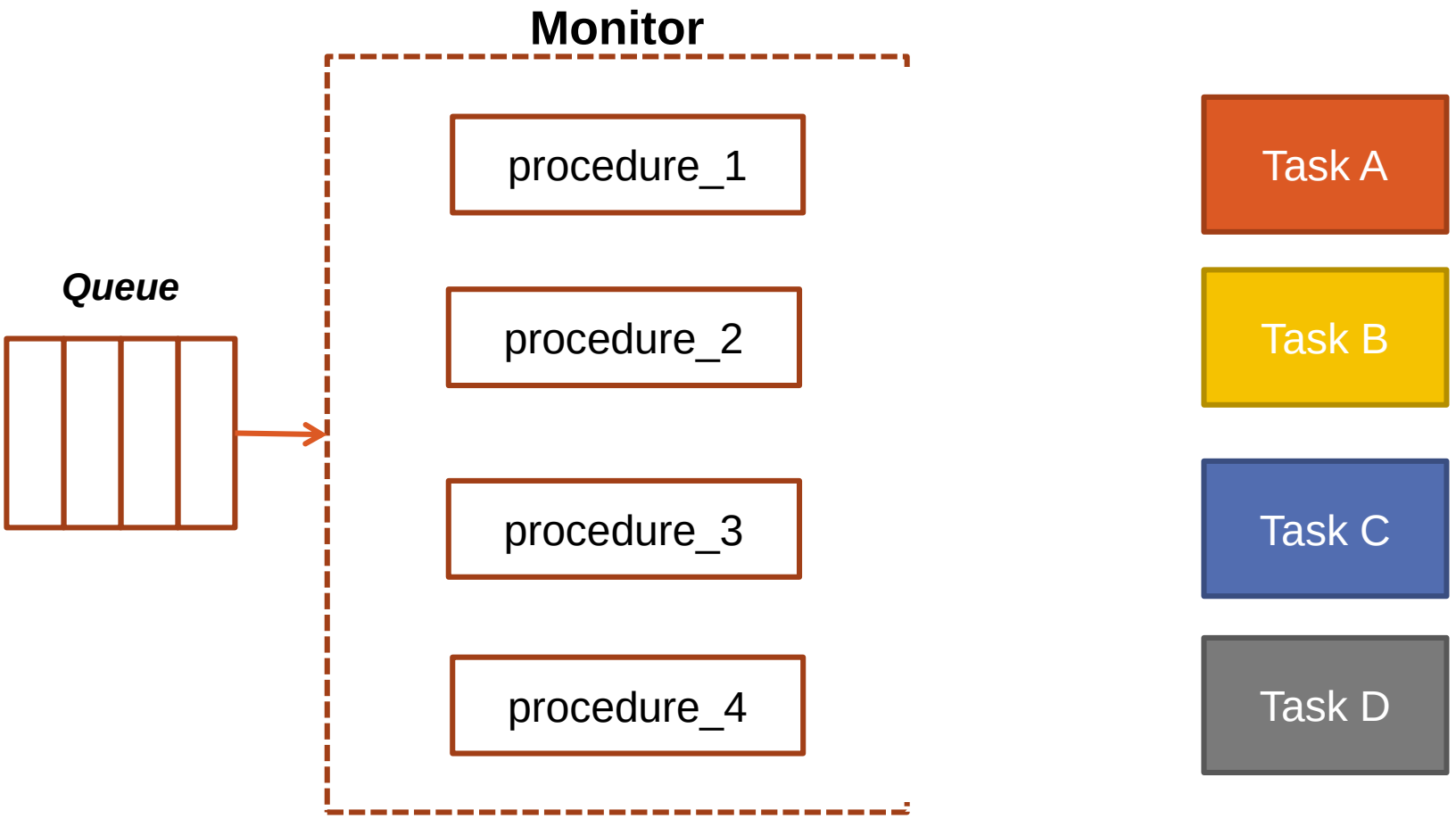
Monitor Owner: Task B

EXAMPLE OF USING MONITOR



Monitor Owner: None

EXAMPLE OF USING MONITOR



Monitor Owner: None

BANK ACCOUNT EXAMPLE

```
MONITOR: Account
          double balance


          procedure double withdraw(amount )
          begin
             balance = balance – amount


             return balance
          end procedure
```

withdraw (amount)
balance = balance - amount

withdraw (amount)

withdraw (amount)

return balance

balance = balance – amount
return balance

balance = balance – amount
return balance

COOPERATION SYNCHRONIZATION

Although mutually exclusive access to shared data is intrinsic with a monitor:

- Cooperation between tasks is still the responsibility of the programmer

Programmer must guarantee that a shared buffer does not experience underflow or overflow

CONDITION VARIABLES

Condition variables provide a mechanism to wait for events

Condition variables support three operations:

- Wait: release monitor lock and wait for condition variable to be signaled
- Signal: wakeup one waiting thread
- Broadcast: wakeup all waiting threads

Each condition variable has a queue associated with it

- A task waiting on that condition is blocked and its descriptor is stored in the queue

PRODUCER CONSUMER EXAMPLE – SEMAPHORE VS MONITOR

Producer

```
task producer;
  loop
    -- produce VALUE --
    bufferMonitor.deposit(VALUE)
  end loop;
end producer;
```

Consumer

```
task consumer;
  loop
    VALUE = bufferMonitor.fetch()
    -- consume VALUE --
  end loop;
end consumer;
```

PRODUCER CONSUMER EXAMPLE – SEMAPHORE VS MONITOR

Producer

```
task producer;
  loop
    -- produce VALUE --
    bufferMonitor.deposit(VALUE)
  end loop;
end producer;
```

Consumer

```
task consumer;
  loop
    VALUE = bufferMonitor.fetch()
    -- consume VALUE --
  end loop;
end consumer;
```

```
task producer;
  loop
  -- produce VALUE --
  wait(emptyspots);      { wait for a space }
  wait(access);          { wait for access }
  DEPOSIT(VALUE);
  release(access);       { relinquish access }
  release(fullspots);    { increase filled spaces }
  end loop;
end producer;
```

```
task consumer;
  loop
  wait(fullspots);       { make sure it is not empty }
  wait(access);          { wait for access }
  FETCH(VALUE);
  release(access);       { relinquish access }
  release(emptyspots);   { increase empty spaces }
  -- consume VALUE --
  end loop
end consumer;
```

PRODUCER CONSUMER EXAMPLE

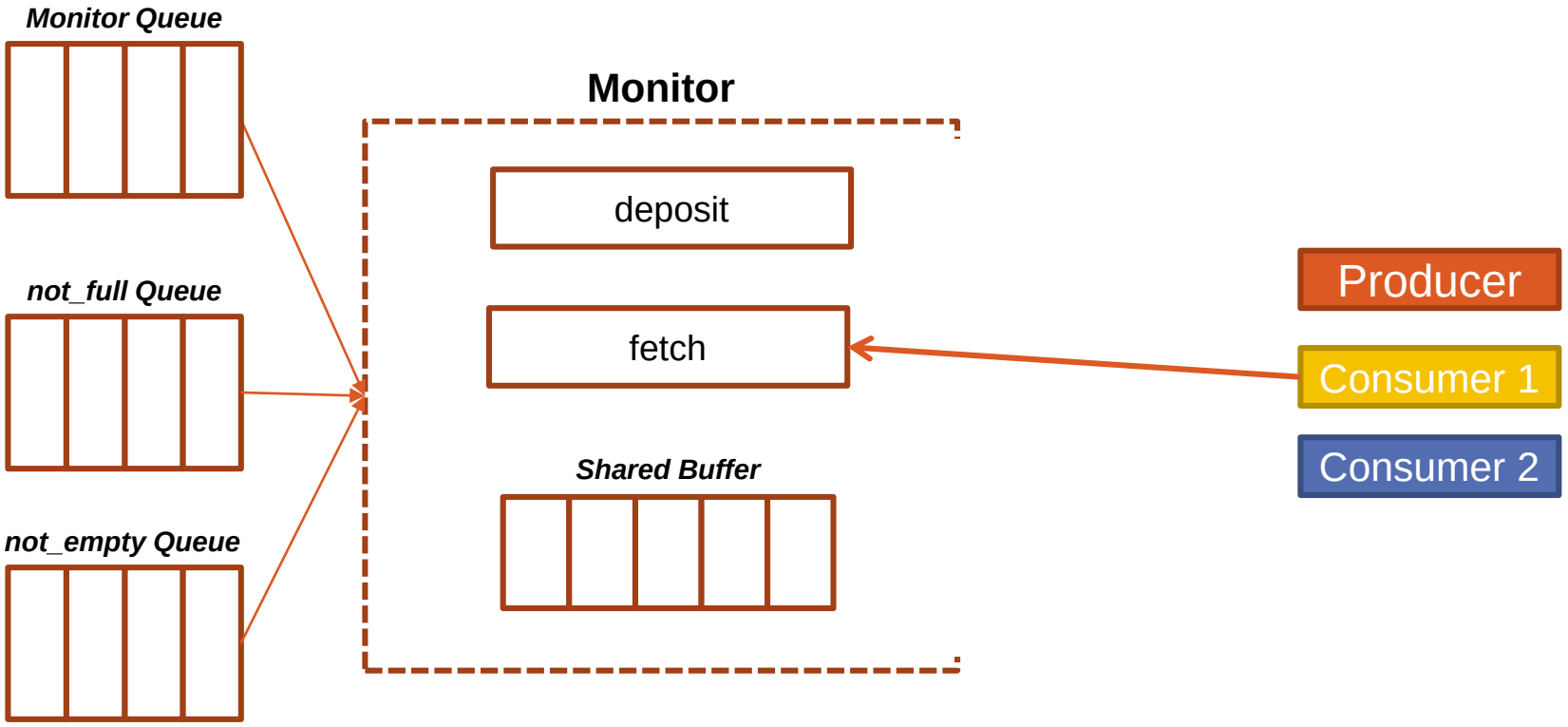
uOttawa
L'Université canadienne
Canada's university

```
MONITOR: BufferMonitor
        const bufferSize = 5
        buffer = array [0.. bufferSize-1]
        next_in = 0, next_out = 0, filled = 0
        condition not_full, not_empty

        procedure void deposit (item )
        begin
           while filled == bufferSize then
              wait(not_full) // block thread and place it in the not_full queue
           end
           buffer[next_in] = item
           next_in = (next_in + 1) mod bufferSize
           filled = filled + 1
           signal(not_empty) // free a task that has been waiting on not_empty
        end procedure
```

PRODUCER CONSUMER EXAMPLE

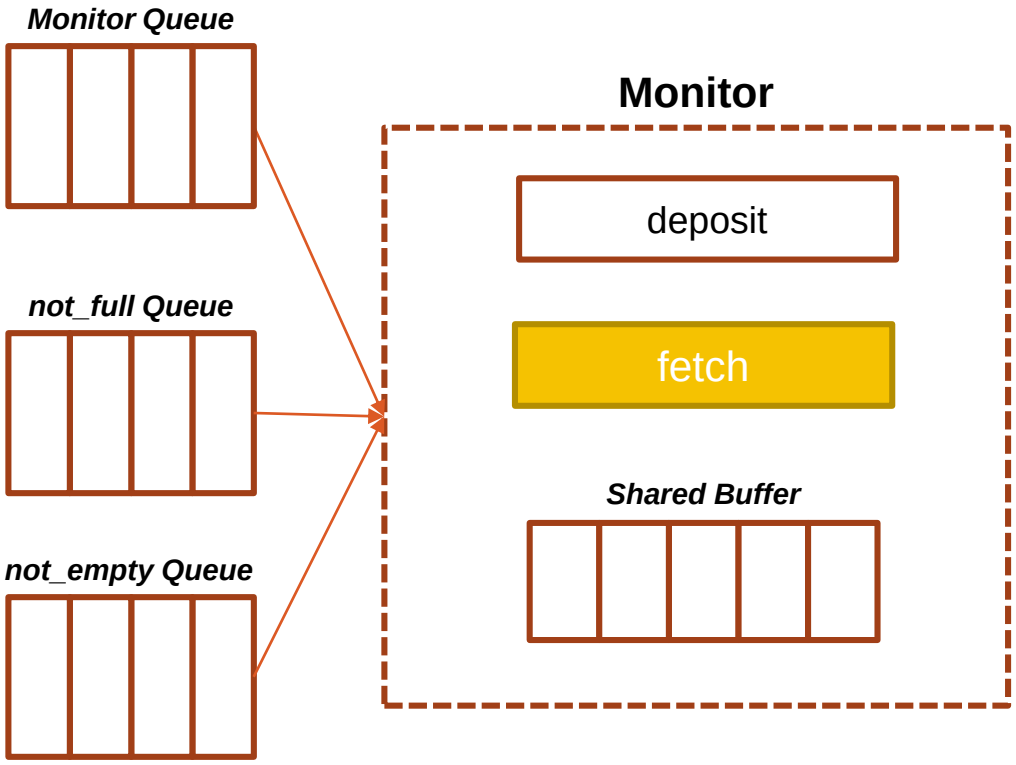

```
procedure Item fetch()
begin
   while filled == 0 then
      wait(not_empty) // block thread and place it in the not_empty queue

   end

   item = buffer[next_out]
   next_out = (next_out + 1) mod bufferSize
   filled = filled - 1

   signal(not_full) // free a task that has been waiting on not_full

   return item
end procedure
```

PRODUCER CONSUMER EXAMPLE



Monitor Owner: None

PRODUCER CONSUMER EXAMPLE



Producer

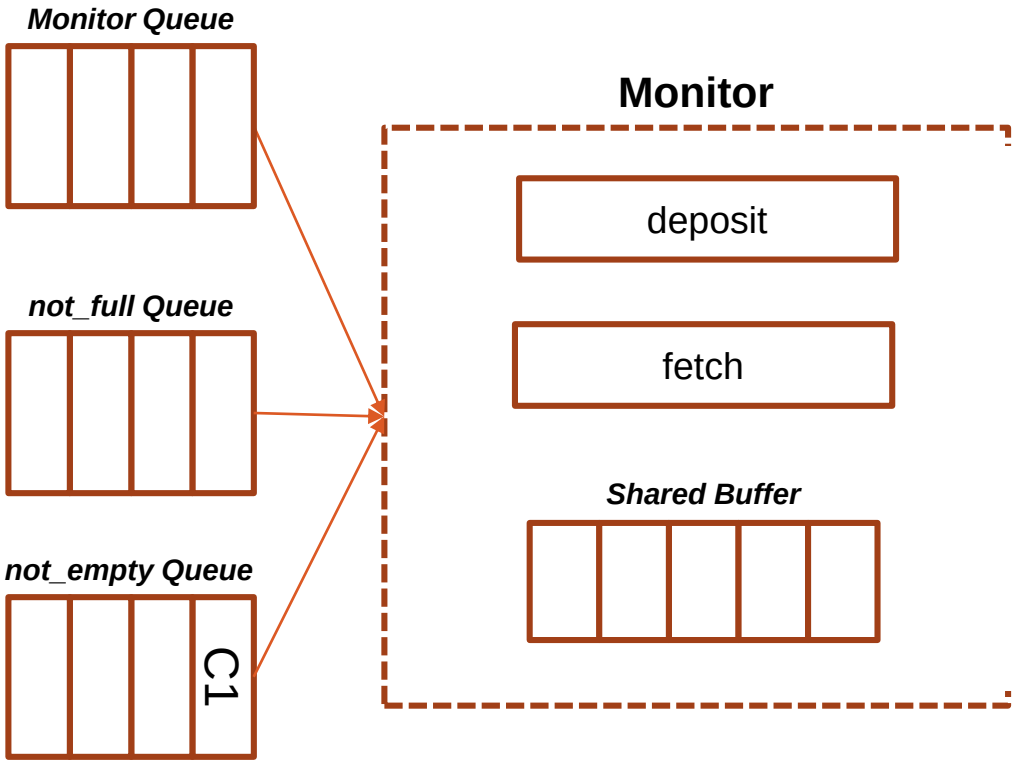
Consumer 1

Consumer 2

```
task producer;
  loop
    -- produce VALUE --
    bufferMonitor.deposit(VALUE)
  end loop;
end producer;
```

Monitor Owner: Consumer 1

PRODUCER CONSUMER EXAMPLE



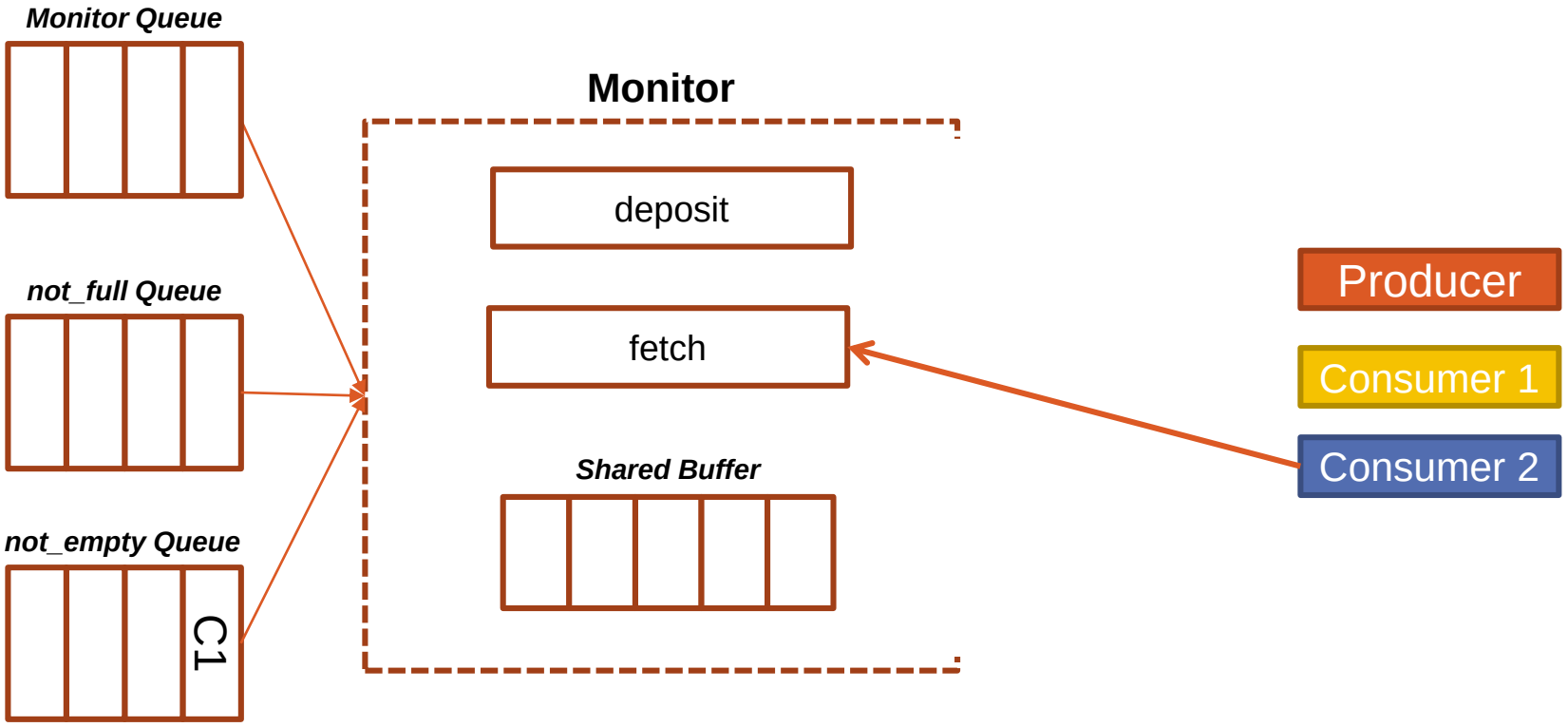
Producer

Consumer 1

Consumer 2

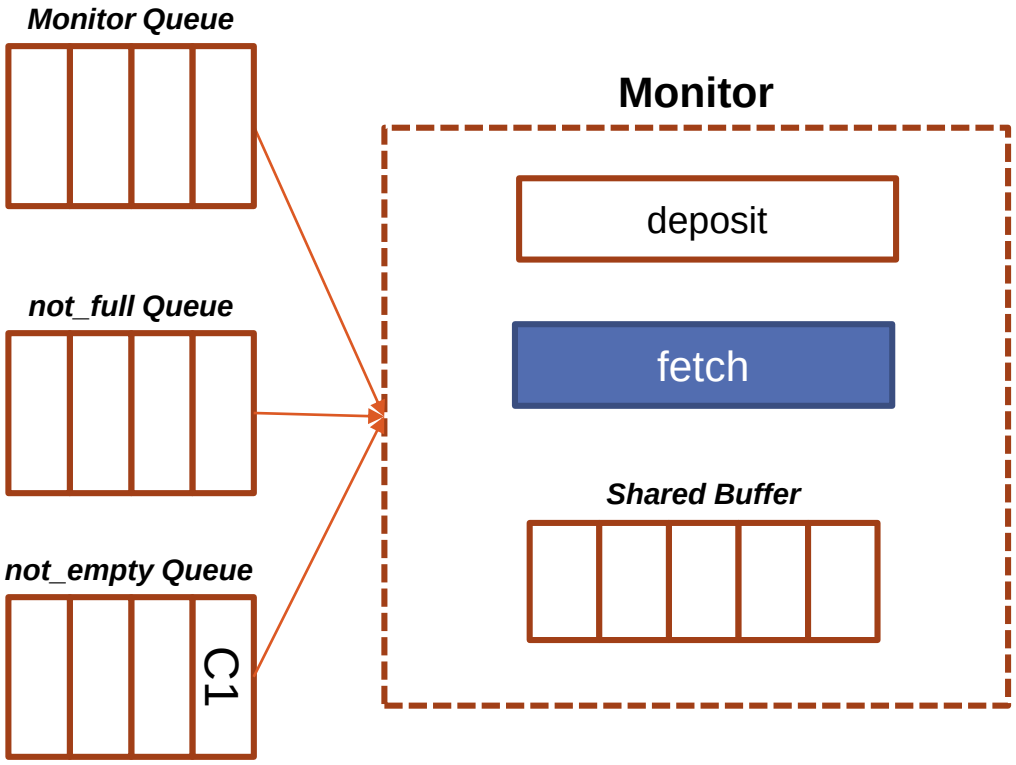
Monitor Owner: None

PRODUCER CONSUMER EXAMPLE



Monitor Owner: None

PRODUCER CONSUMER EXAMPLE



Producer

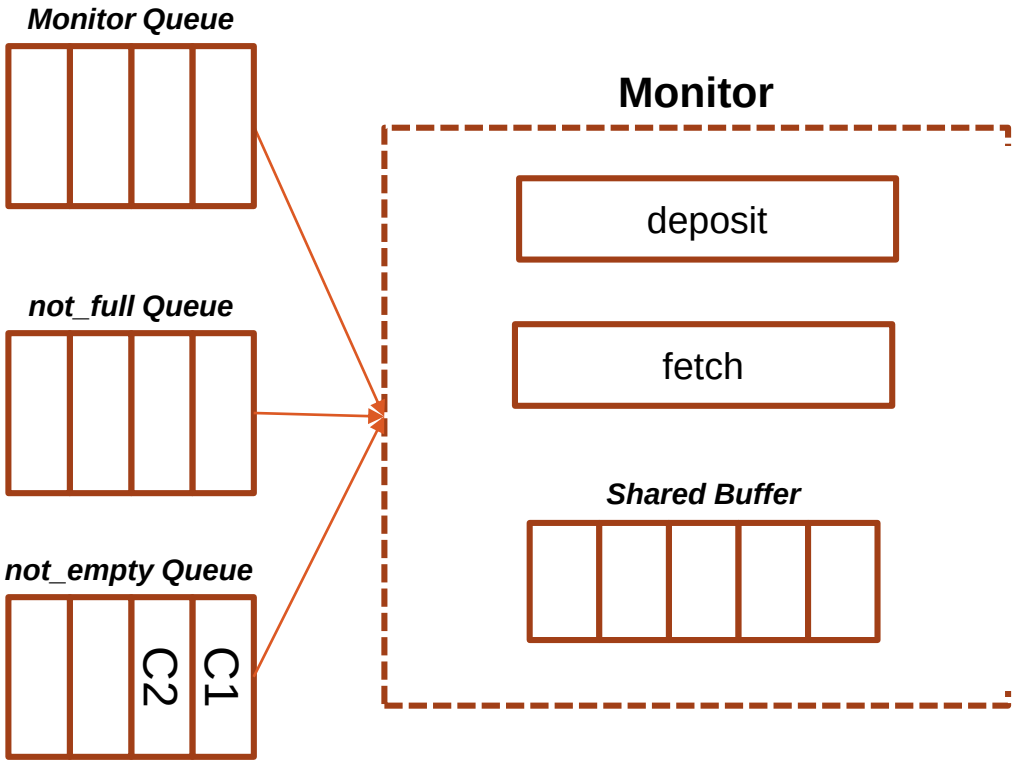
Consumer 1

Consumer 2

```
task producer;
  loop
    -- produce VALUE --
    bufferMonitor.deposit(VALUE)
  end loop;
end producer;
```

Monitor Owner: Consumer 2

PRODUCER CONSUMER EXAMPLE



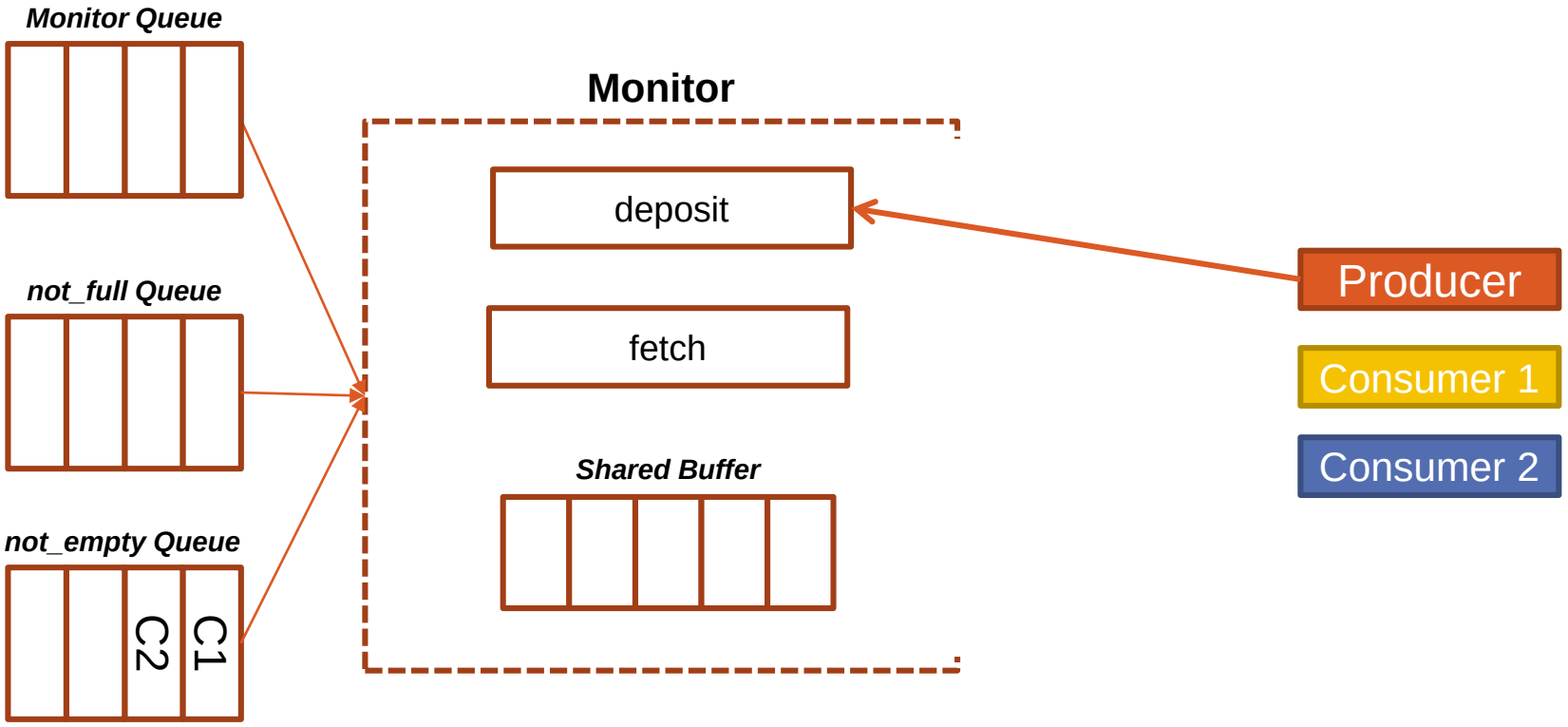
Producer

Consumer 1

Consumer 2

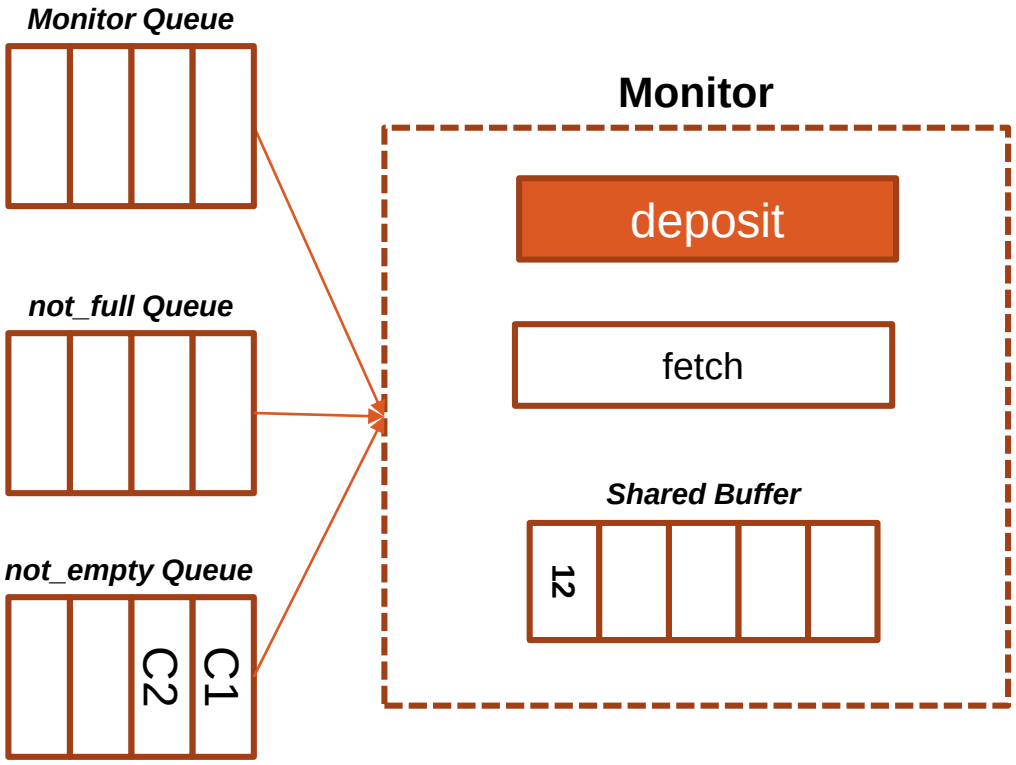
Monitor Owner: None

PRODUCER CONSUMER EXAMPLE



Monitor Owner: None

PRODUCER CONSUMER EXAMPLE



Producer

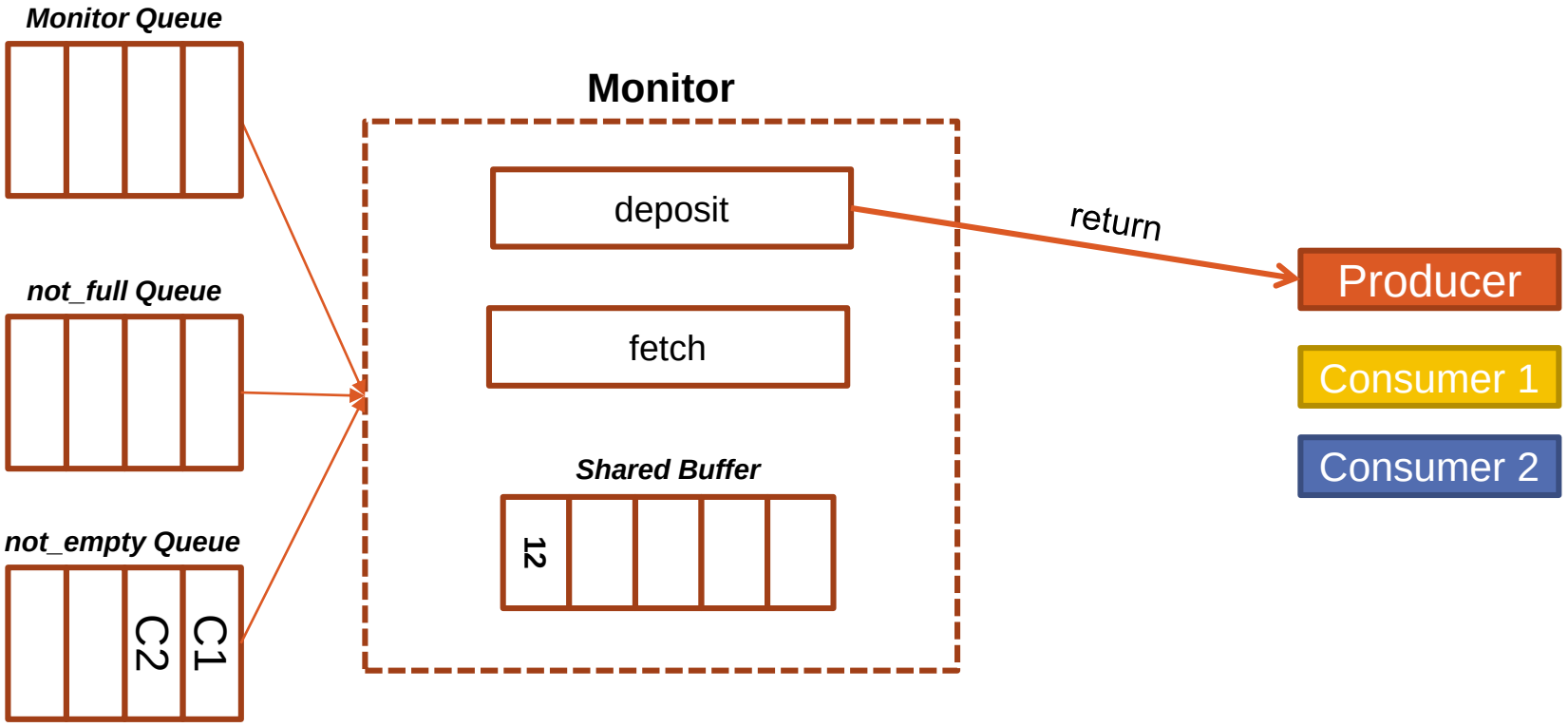
Consumer 1

Consumer 2

```
task consumer;
  loop
    VALUE = bufferMonitor.fetch()
    -- consume VALUE --
  end loop;
end consumer;
```

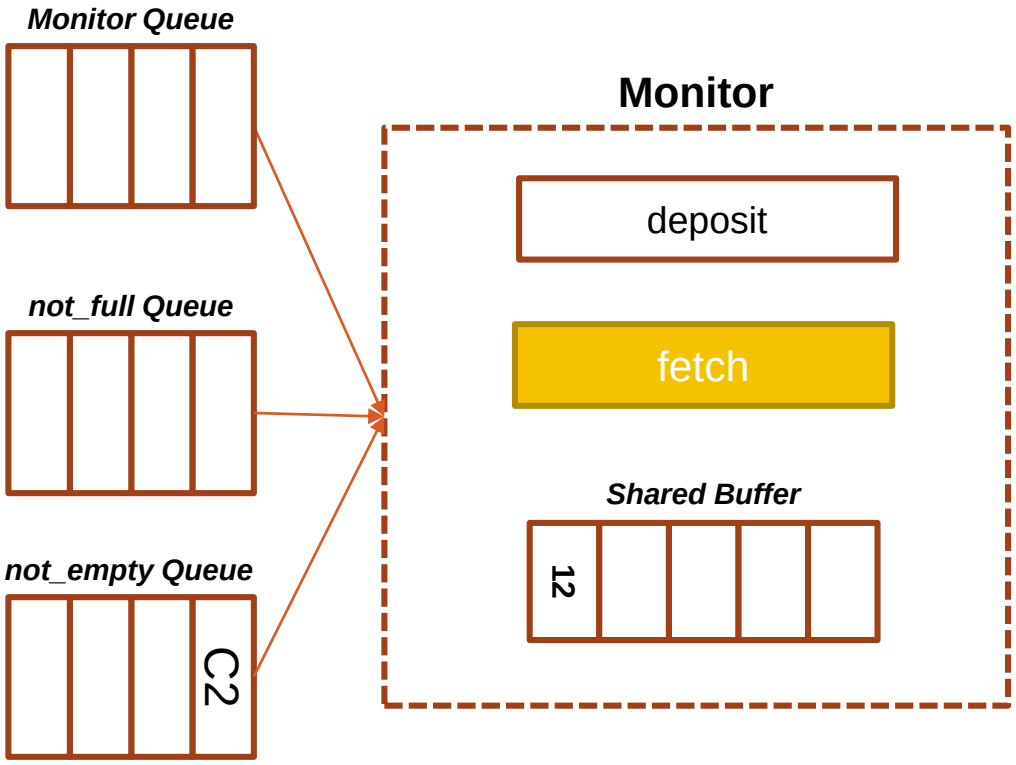
Monitor Owner: Producer

PRODUCER CONSUMER EXAMPLE



Monitor Owner: None

PRODUCER CONSUMER EXAMPLE



Producer

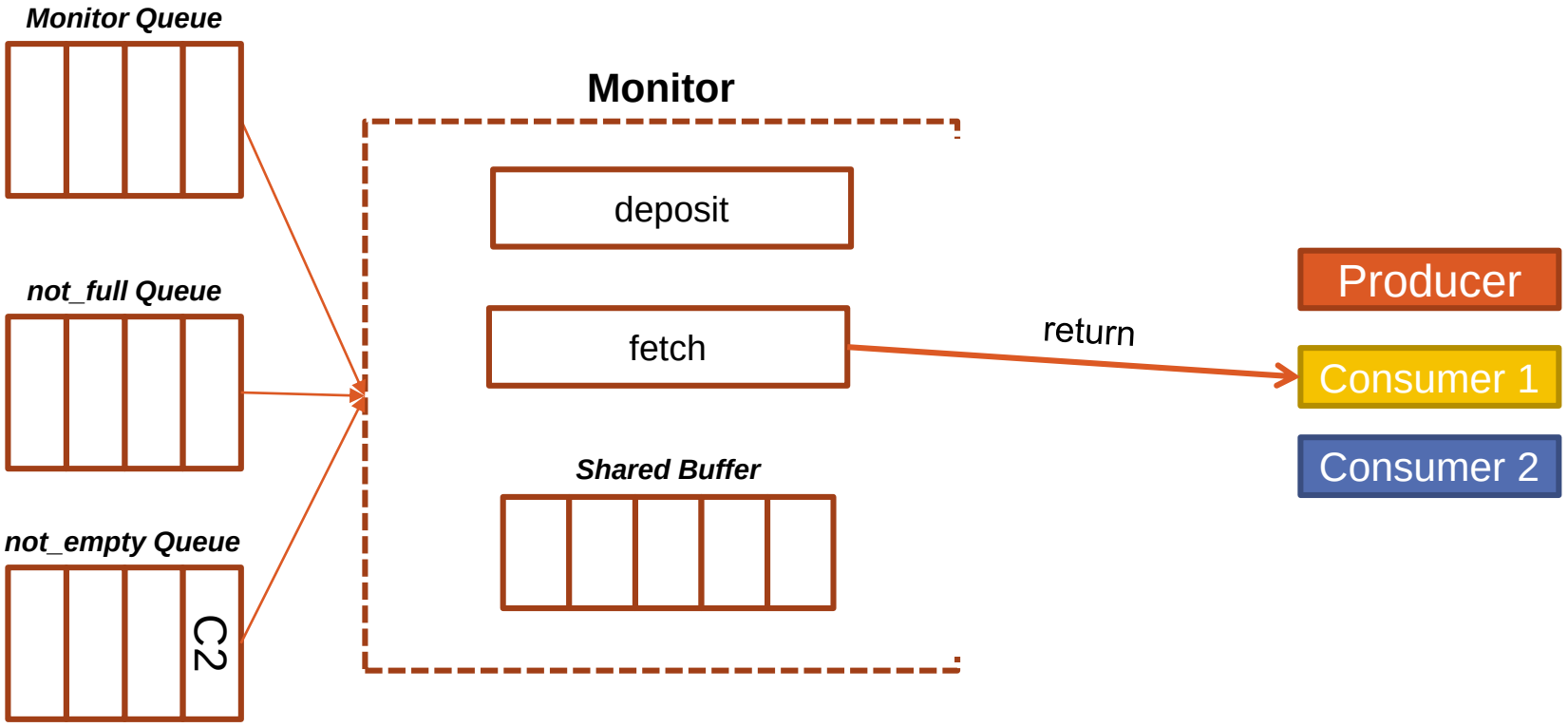
Consumer 1

Consumer 2

```
task producer;
  loop
    -- produce VALUE --
    bufferMonitor.deposit(VALUE)
  end loop;
end producer;
```

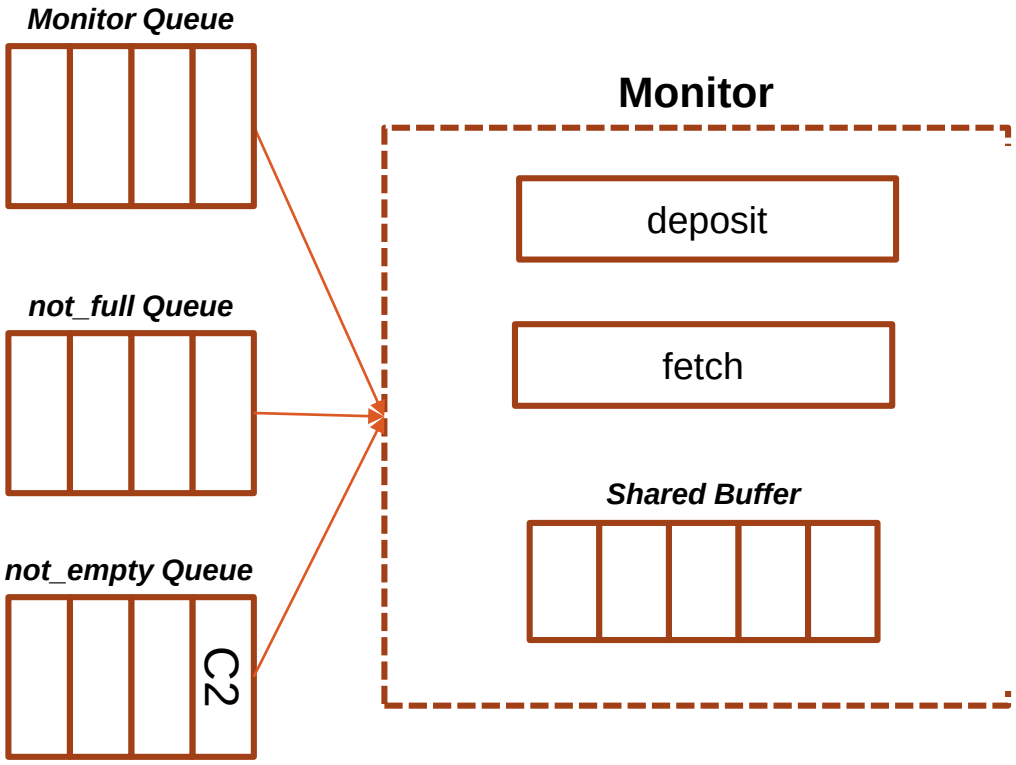
Monitor Owner: Consumer 1

PRODUCER CONSUMER EXAMPLE



Monitor Owner: None

PRODUCER CONSUMER EXAMPLE



Producer

Consumer 1

Consumer 2

Monitor Owner: None

THANK YOU!

QUESTIONS?