Monitor

- A monitor is a software construct that serves two purposes:
  - enforces mutual exclusion of concurrent accessto shared data objects
    - Processes have to acquire a lock to access such a shared resource
  - Support conditional synchronisation between processes accessing shared data
    - Multiple processes may use monitorspecific wait()/signal() mechanisms to wait for particular conditions to hold

Monitor

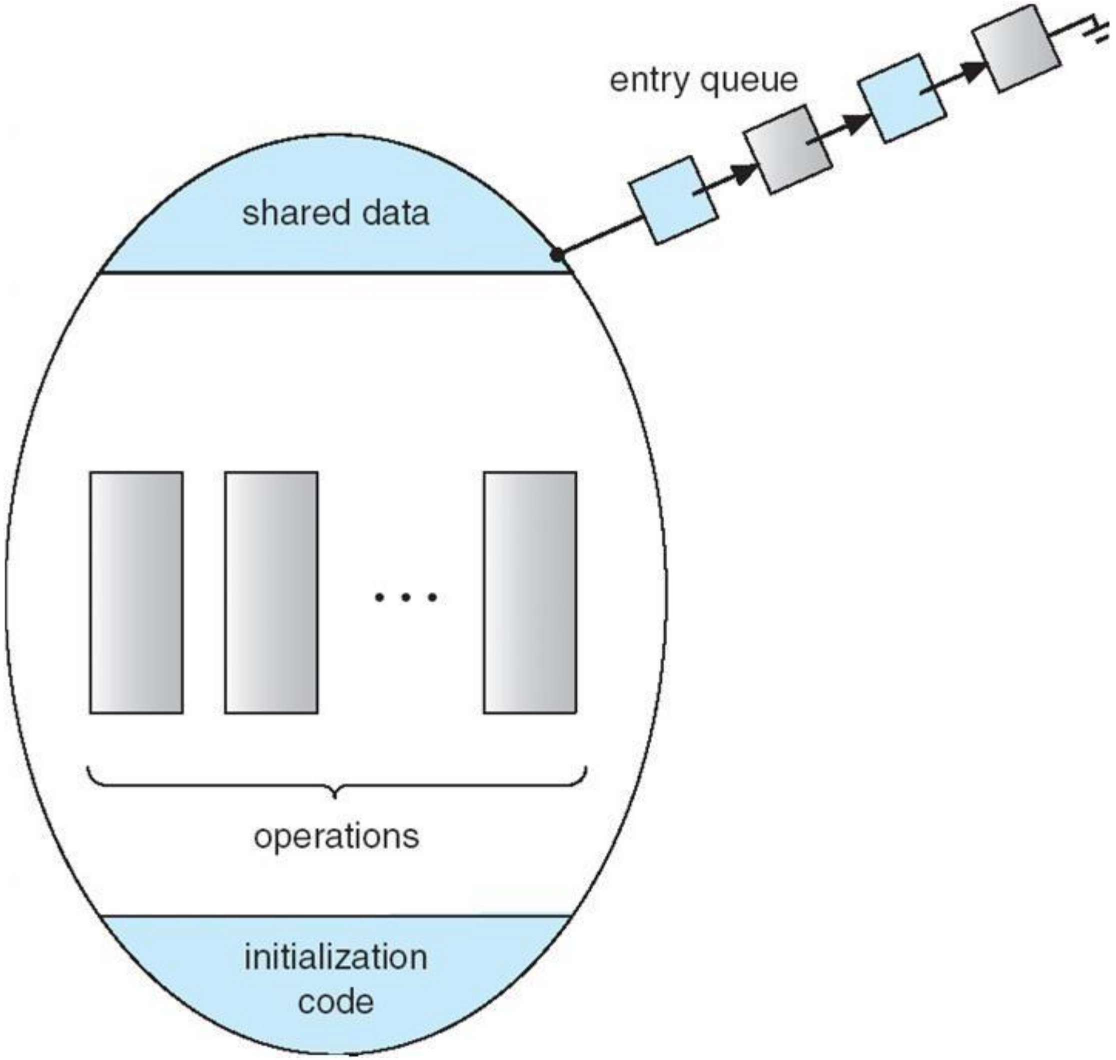
- Monitors are typically supported by a programming language
  - Languagespecific software construct
  - Prominent implementation by Java classes
- Programs using monitors are supposed to allow easier implementation of mutual exclusion and synchronisation

Monitor Characteristics

- A monitor is a software construct consisting of
  - One or more procedures
  - Some local data that can only be accessed via these procedures
- Objectoriented concepts
  - Local variables accessible only by the monitor’s procedures (methods)
- Processes “enter” monitor when they invoke one of the monitor’s procedures
- Mutual exclusion:
  - Only one process at a time may call one of these procedures and “enter” the monitor
  - All other processes have to wait

Monitor: Entry

- A monitor has an entry queue
- Processes calling monitor procedures may be added to waiting queue and suspended



Mutual Exclusion

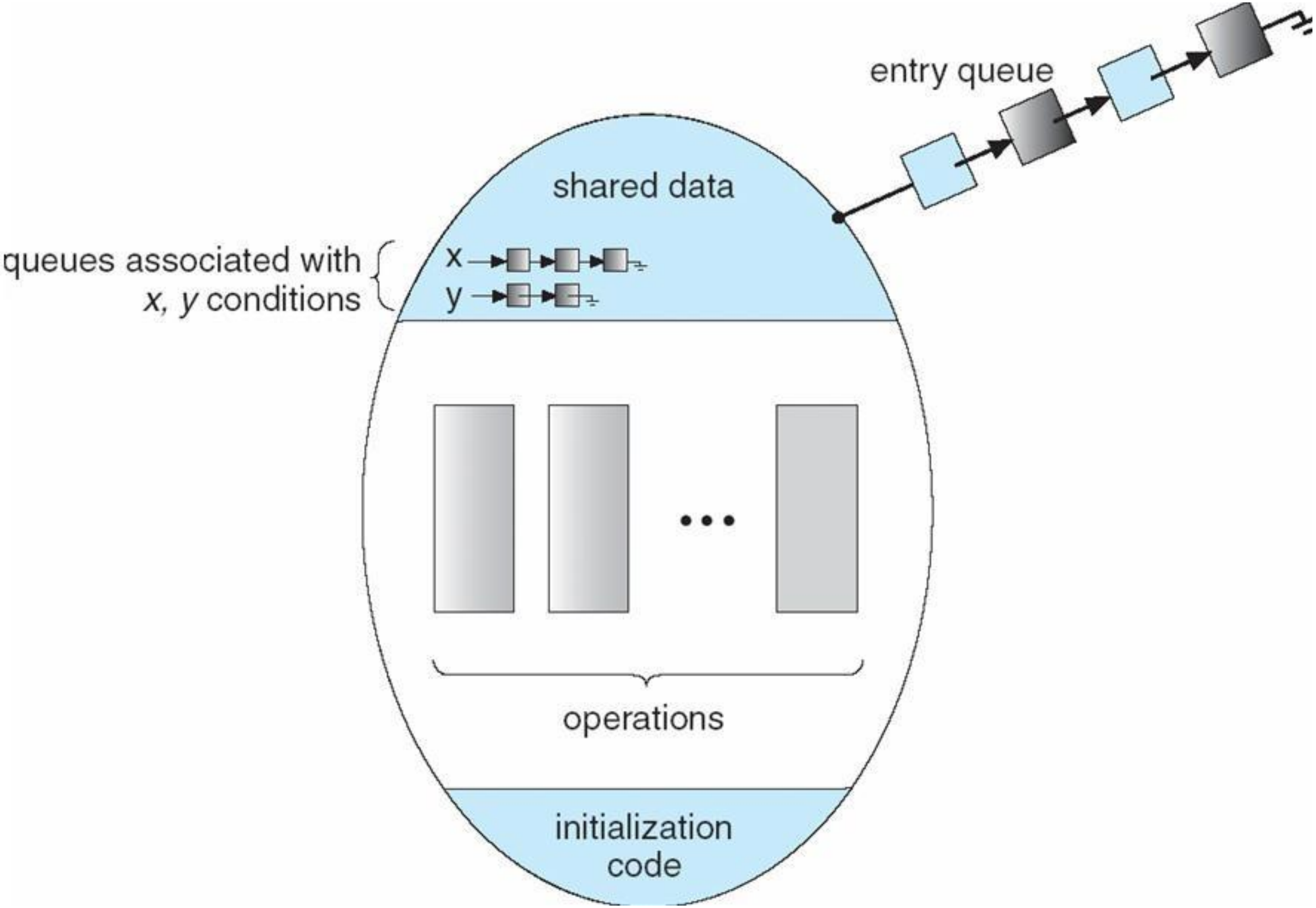
- When a process calls one of these procedures, it “enters” the monitor
  - The process has to acquire a monitor lock first
- The monitor guarantees that
  - Only one process at a time may call one of these procedures and “enter” the monitor
  - All other processes have to wait

```
monitor SharedBuffer
   buffer b[N] ;
   int in, out ;

   procedure  append()
   . . .
   end;

   procedure take()
   . . .
   end;
end monitor;
```

Process Synchronisation

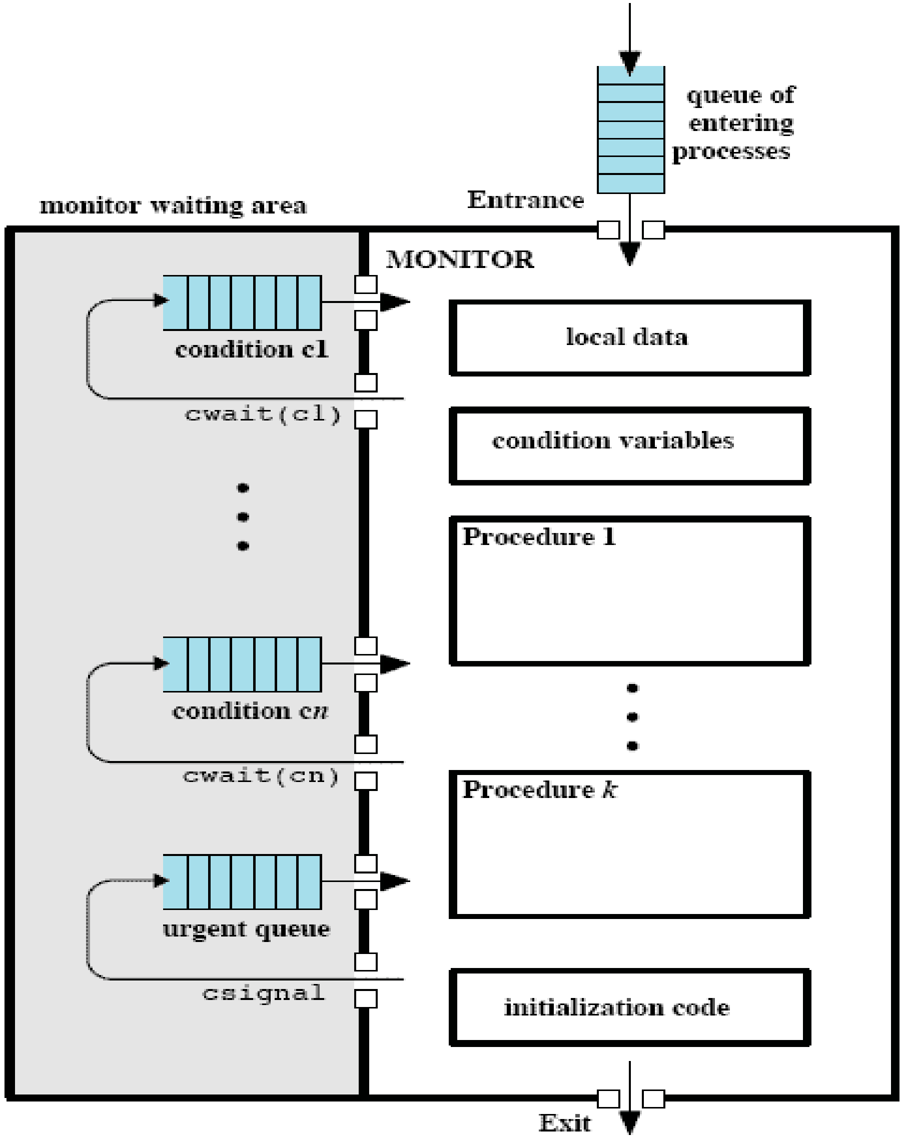
- A monitor also supports process synchronisation with condition variables
  - Only accessible within the monitor with the functions wait(condition) and signal(condition)
- A monitor may maintain a set of these condition variables
- For each condition variable, the monitor maintains a waiting queue

Condition Variables



Process Synchronisation with Condition Variables

- Signalling mechanism:
  - Process may call wait(condition variable)
    - made to wait for a condition in a condition queue
  - Process may call signal(condition variable)
    - This resumes one of the processes waiting for this conditional signal



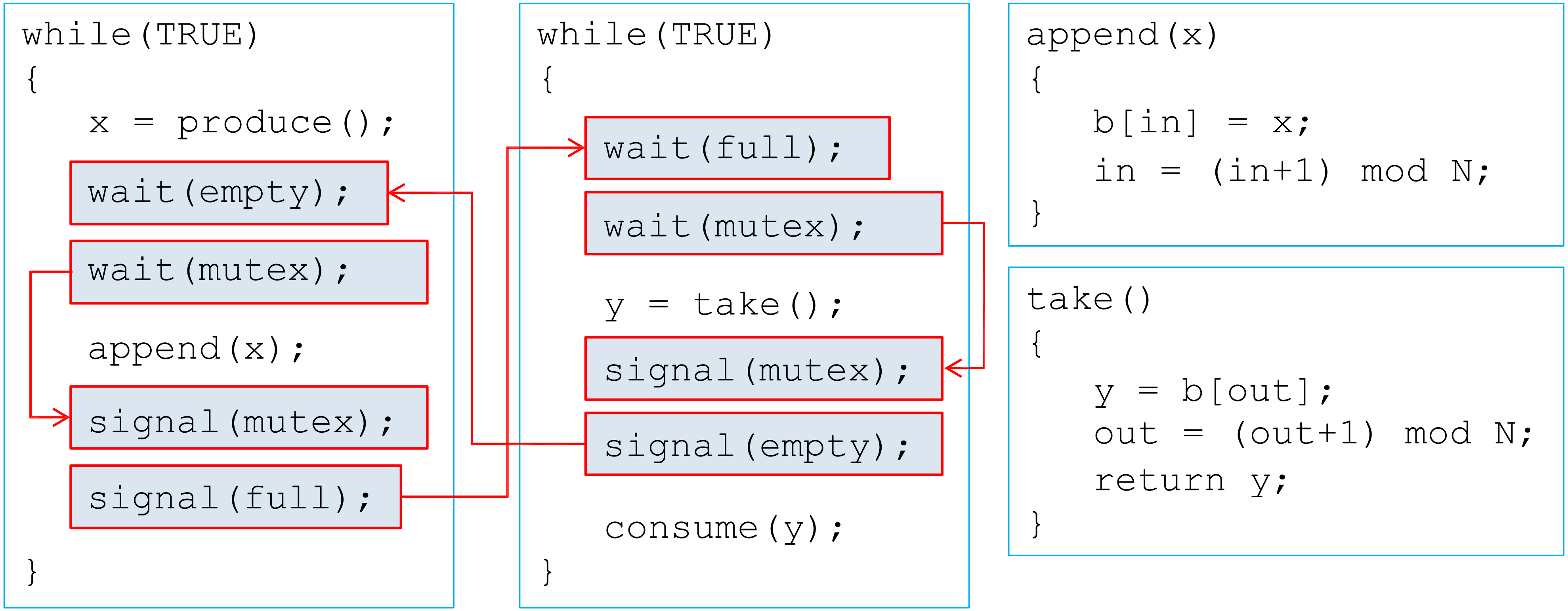
Process Synchronisation

- For each condition variable, the monitor maintains a waiting queue
- wait(condition variable) :
  - a process calling this functions is suspended
  - releases monitor lock
  - waits until a signal based on condition c is received, re acquires lock
- signal(condition variable) :
  - resume one of the processes waiting

Producer – Consumer Ring Buffer Semaphores

```
init(mutex,1); init(full,0); init(empty, N);
in = 0; out = 0; buffer[N] ;
```

Producer Consumer



```
while(TRUE)
{
   x = produce();
   wait(empty);
   wait(mutex);
   append(x);
   signal(mutex);
   signal(full);
}
```

```
while(TRUE)
{
    wait(full);
    wait(mutex);
    y = take();
    signal(mutex);
    signal(empty);
    consume(y);
}
```

```
append(x)
{
   b[in] = x;
   in = (in+1) mod N;
}
```

```
take()
{
   y = b[out];
   out = (out+1) mod N;
   return y;
}
```

- Bounded buffer:
  - –Buffer limited to N places, is managed as a circular buffer

Producer – Consumer Ringbuffer Monitor

Producer

```
while(TRUE)
{
    x = produce();
    append(x);
}
```

Consumer

```
while(TRUE)
{
    y = take();
    consume(y);
}
```

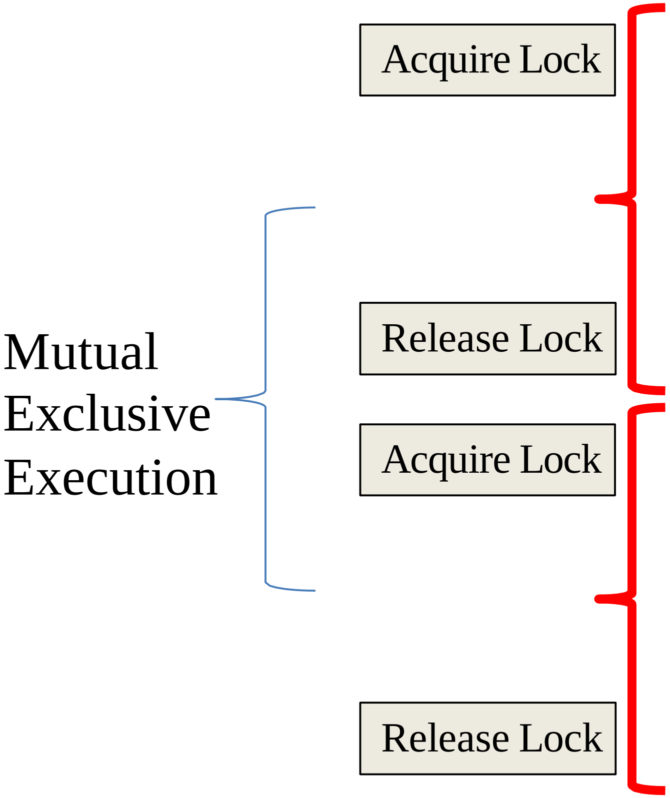
RingBuffer

```
in = 0;
out = 0;
count = 0;
buffer[N];
notfull;
notempty;
```

```
void append(char item) {

    if (count == N) wait(notfull);
    b[in] = item;
    in = (in+1) mod N;
    count++;
    signal(notempty);
}
```

```
char take() {
    if (count == 0) wait(notempty);
    item = b[out];
    out = (out+1 ) mod N;
    count--;
    signal(notfull);
    return item;
}
```

Producer Consumer



Acquire Lock

Release Lock

Mutual Exclusive Execution

Acquire Lock

Release Lock

```
monitor boundedBuffer
{
     char b[N]; int count, in, out ;
     condition notfull, notempty;

     void append(char item) {
          if (count == N) wait(notfull) ;
          b[in] = item;
          in = (in+1) mod N;
          count++;
          signal(notempty);
     }
     char take() {
          if (count == 0) wait(notempty) ;
          item = b[out];
          out = (out+1) mod N;
          count--;
          signal(notfull);
     }
}
```

Behaviour of signal()
Resuming Processes waiting in Monitor

- Monitor ensures that only one process is active when within the monitor
  - All other processes will wait
- How will the monitor behave if a process calls signal(condition variable)?
  - Invoking signal(condition variable) will wake up and resume exactly one process waiting in the queue of the condition variable
- The issuing of “signal()” and the rescheduling of a process waiting for this signal has to be atomic

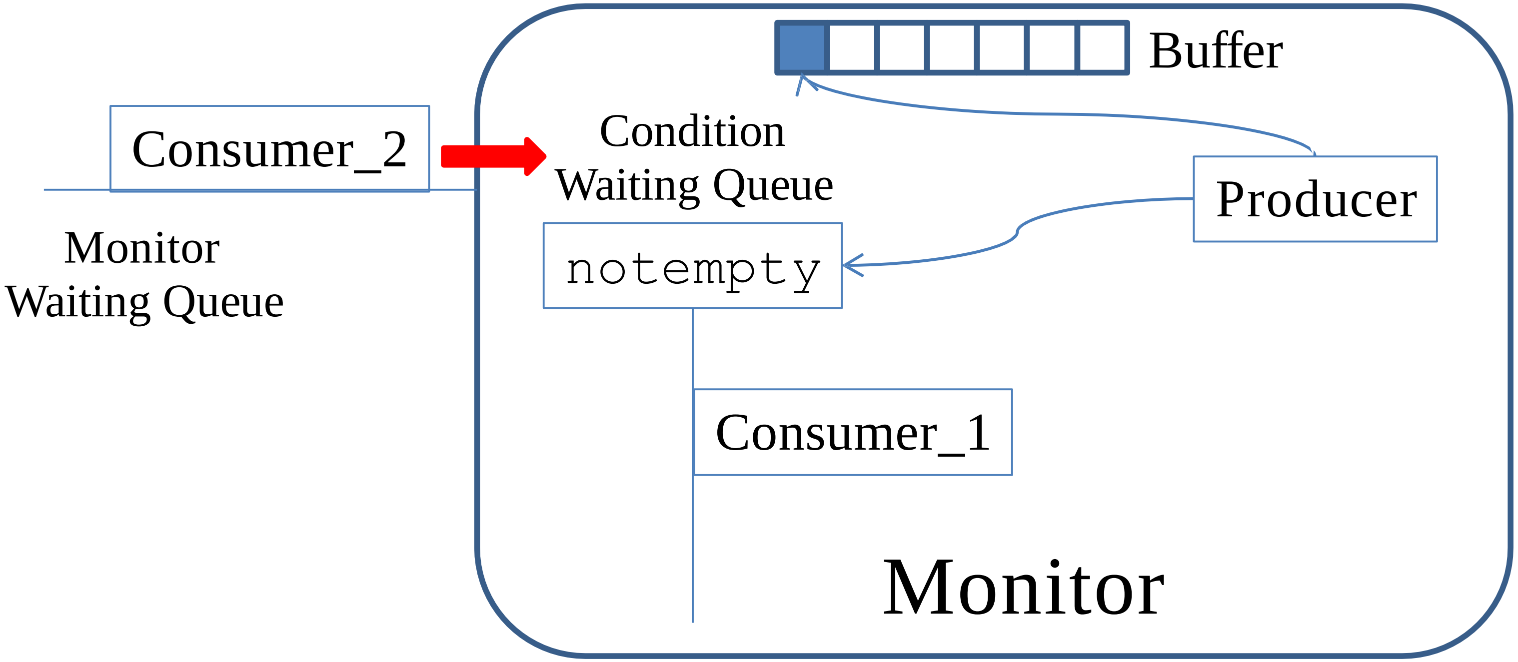
Behaviour of signal()
Resuming Processes waiting in Monitor

- E.g. signal(notempty) indicates "buffer is not empty any more"
    - A consumer waiting in the "notempty" queue should wake up and consume it
- Producer releases the monitor lock, a new consumer could enter before the waiting consumer
    - The new consumer will then acquire the monitor lock before the waiting consumer , pass the wait(notempty) and consume the buffer content
- Waiting consumer is rescheduled, comes out of wait(notempty) , but will find an empty buffer

```
void append(char item) {

    if (count == N) wait(notfull);

    b[in] = item;
    in = (in+1) mod N;
    count++;
    signal(notempty);
}
```

```
char take() {
    if (count == 0) wait(notempty);
    item = b[out];
    out = (out+1 ) mod N;
    count--;

    signal(notfull);

    return item;
}
```

Behaviour of signal()
Resuming Processes waiting in Monitor



- Nonatomicity of signal() and process resumption
  - Producer signals consumer 1
  - Before consumer 1 is scheduled, consumer 2 enters monitor and consumes buffer content
  - When consumer 1 is finally scheduled, it will leave wait() and find that buffer is already empty

Behaviour of signal()
Hoare's Definition of Monitors

- Monitor as originally defined by Hoare
- If a process issues a signal(condition variable)
  - it is immediately suspended to free monitor
- If there is another process waiting in the queue of the condition variable
  - it has to be rescheduled immediately

Behaviour of signal()
Hoare's Definition of Monitors

- Drawbacks
  - The process performing the signal() may not be finished, it must be rescheduled and gain access to monitor again
    - Multiple process switches become necessary
  - When a signal() is issued, a process waiting in the corresponding condition queue must be activated immediately
    - No other arbitrary process is allowed to enter the monitor
    - Why: because such a process could change the condition that led to the activation of the waiting process

Monitors with Notify and Broadcast

- The signal(c) is replaced by a notify(c)
  - When a signalling process issues the notify(c) for condition queue c, it continues to execute
    - It notifies the queue because a particular condition currently holds, e.g. “buffer is not empty any more”
  - The next process to be executed from the condition queue c will be rescheduled when the monitor becomes available
  - Rescheduled process has to recheck condition
    - E.g. “is buffer still not empty?”
    - This is necessary, because another process could be scheduled before and interfere

Monitors with notify() and Broadcast

- Note the "while" loop:
  - Rechecking the condition after wakeup
- Allows for non atomicity between notify() and wakeup()

```
monitor boundedBuffer
{
     char b[N]; int count, in, out ;
     condition notfull, notempty;

     void append(char item) {
          while (count == N) wait(notfull);
          b[in] = item;
          in = (in+1) % N;
          count++;
          notify(notempty);
     }
     char take() {
          while (count == 0) wait(notempty);
          item = b[out];
          out = (out+1) % N;
          count--;
          notify(notfull);
     }
}
```

Monitor vs Semaphores

- Monitor
  - The monitor construct itself enforces mutual exclusion
  - Programmer not dealing with mutual exclusion issues
  - However, programmer has to place condition checks in program code to enforce condition synchronisation (e.g. Manage the bounded buffer read and writes)
- Semaphore
  - Programmer has to do both mutual exclusion and condition synchronisation programming
- Benefit of monitor
  - All synchronisation functionality confined to the monitor

Message Passing

- Enforce mutual exclusion
- Exchange information

```
send (destination, message)
receive (source, message)
```

Synchronization

- Sender and receiver may or may not be blocking (waiting for message)
- Blocking send, blocking receive
  - Both sender and receiver are blocked until message is delivered
  - Called a rendezvous

Synchronization

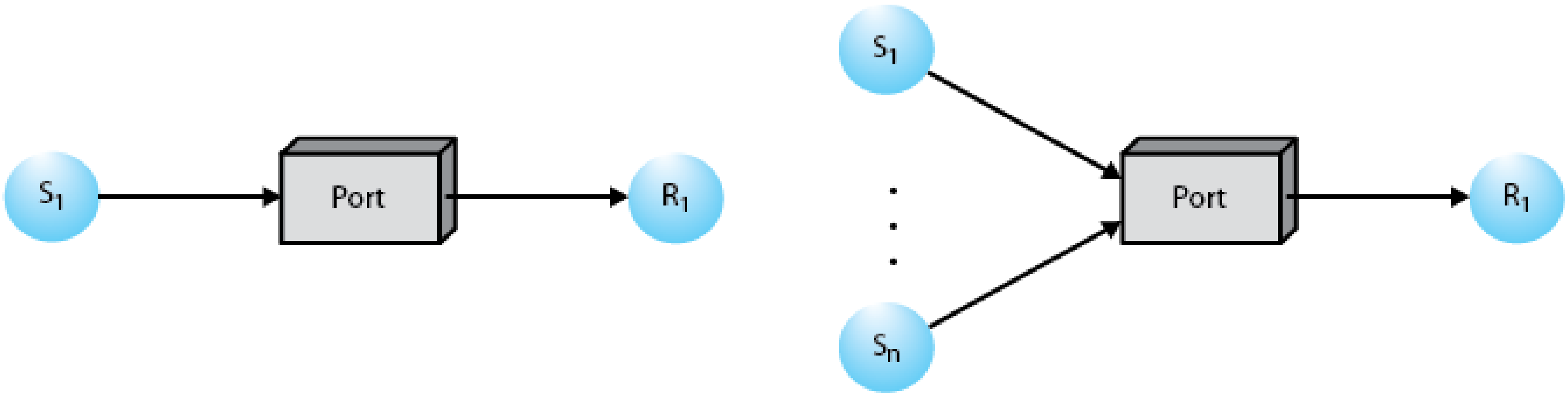
- Nonblocking send, blocking receive
  - Sender continues on
  - Receiver is blocked until the requested message arrives
- Nonblocking send, nonblocking receive
  - Neither party is required to wait

Addressing

- Direct addressing
  - Send primitive includes a specific identifier of the destination process
  - Receive primitive could know ahead of time which process a message is expected
  - Receive primitive could use source parameter to return a value when the receive operation has been performed

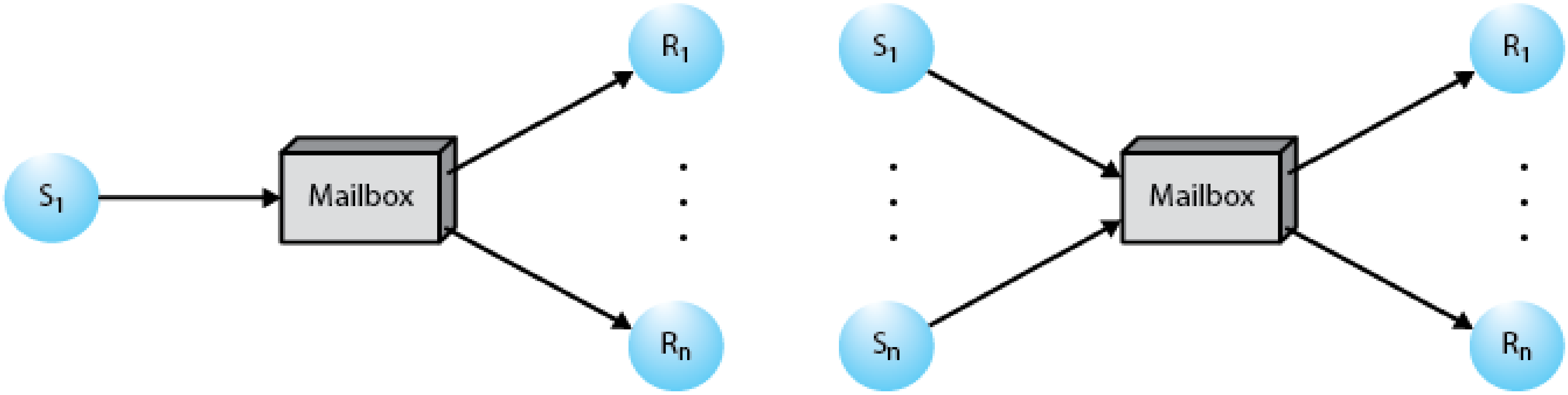
Addressing

- Indirect addressing
  - Messages are sent to a shared data structure consisting of queues
  - Queues are called mailboxes
  - One process sends a message to the mailbox and the other process picks up the message from the mailbox



(a) One to one

(b) Many to one



(c) One to many

(d) Many to many

Figure 5.18 Indirect Process Communication

Message Format

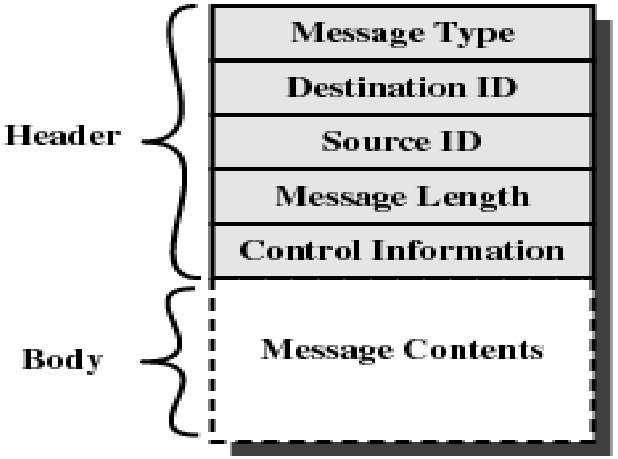
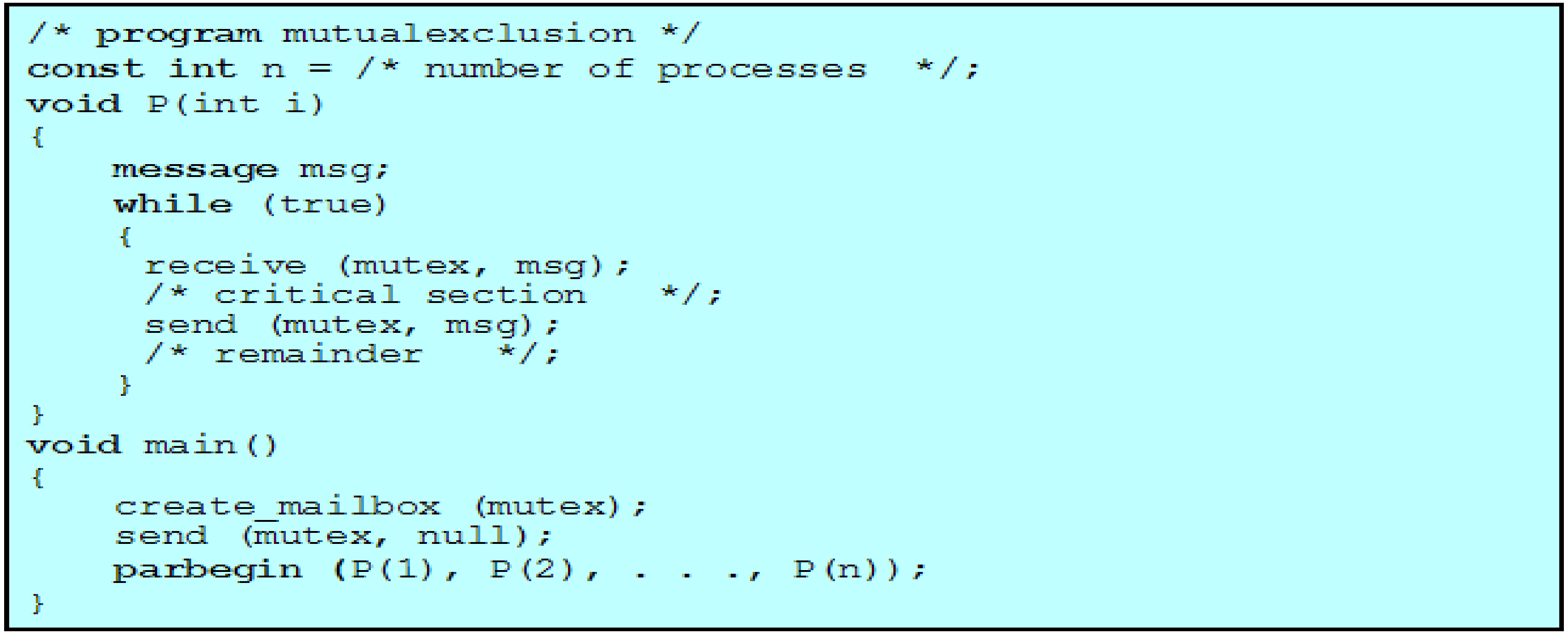


Figure 5.19 General Message Format



```
/* program mutualexclusion */
const int n = /* number of processes  */;
void P(int i)
{
    message msg;
    while (true)
    {
     receive (mutex, msg);
     /* critical section   */;
     send (mutex, msg);
     /* remainder   */;
    }
}
void main()
{
    create_mailbox (mutex);
    send (mutex, null);
    parbegin (P(1), P(2), . . ., P(n));
}
```

Figure 5.20 Mutual Exclusion Using Messages

```
const int
    capacity = /* buffering capacity */ ;
    null =/* empty message */ ;
int i;
void producer()
{   message pmsg;
    while (true)
    {
     receive (mayproduce, pmsg);
     pmsg = produce();
     send (mayconsume, pmsg);
    }
}
void consumer()
{   message cmsg;
    while (true)
    {
     receive (mayconsume, cmsg);
     consume (cmsg);
     send (mayproduce, null);
    }
}

void main()
{
    create_mailbox (mayproduce);
    create_mailbox (mayconsume);
    for (int i = 1; i <= capacity; i++)
        send (mayproduce, null);
    parbegin (producer, consumer);
}
```

Figure 5.21 A Solution to the Bounded-Buffer Producer/Consumer Problem Using Messages