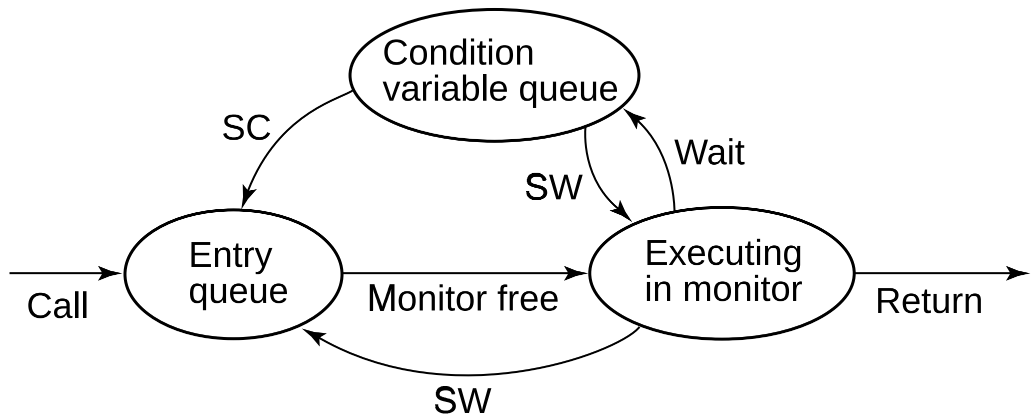

Figure 5.1 State diagram for synchronization in monitors.

```
monitor Semaphore {
  int s = 0;  ## s >= 0
  cond pos;   #  signaled when s > 0
  procedure Psem() {
    while (s == 0) wait(pos);
    s = s-1;
  }
  procedure Vsem() {
    s = s+1;
    signal(pos);
  }
}
```

Figure 5.2 Monitor implementation of a semaphore.

```
monitor FIFOsemaphore {
  int s = 0;  ## s >= 0
  cond pos;   #  signaled when s > 0

  procedure Psem() {
    if (s == 0)
      wait(pos);
    else
      s = s-1;
  }

  procedure Vsem() {
    if (empty(pos))
      s = s+1;
    else
      signal(pos);
  }
}
```

Figure 5.3 FIFO semaphore using passing the condition.

```
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.

```
monitor RW_Controller {

  int nr = 0, nw = 0;  ## (nr == 0 ∨ nw == 0) ∧ nw <= 1
  cond oktoread;    #  signaled when nw == 0
  cond oktowrite;   #  signaled when nr == 0 and nw == 0

  procedure request_read() {
    while (nw > 0) wait(oktoread);
    nr = nr + 1;
  }

  procedure release_read() {
  nr = nr - 1;
  if (nr == 0) signal(oktowrite);  # awaken one writer
  }

  procedure request_write() {
    while (nr > 0 || nw > 0) wait(oktowrite);
    nw = nw + 1;
  }

  procedure release_write() {
    nw = nw - 1;
    signal(oktowrite);      # awaken one writer and
    signal_all(oktoread);   # all readers
  }

}
```

Figure 5.5 Readers/writers solution using monitors.

```
monitor Shortest_Job_Next {

  bool free = true;  ## Invariant SJN:  see text
  cond turn;         #  signaled when resource available

  procedure request(int time) {
    if (free)
      free = false;
    else
      wait(turn, time);
  }

  procedure release() {
    if (empty(turn))
      free = true
    else
      signal(turn);
  }

}
```

Figure 5.6 Shortest-job-next allocation with monitors.

```
monitor Timer {

  int tod = 0;   ## invariant CLOCK -- see text
  cond check;    #  signaled when tod has increased

  procedure delay(int interval) {
    int wake_time;
    wake_time = tod + interval;
    while (wake_time > tod) wait(check);
  }

  procedure tick() {
    tod = tod + 1;
    signal_all(check);
  }

}
```

Figure 5.7 Interval timer with a covering condition.

```
monitor Timer {
  int tod = 0;   ## invariant CLOCK -- see text
  cond check;    #  signaled when minrank(check)<=tod
  procedure delay(int interval) {
    int wake_time;
    wake_time = tod + interval;
    if (wake_time > tod) wait(check, wake_time);
  }
  procedure tick() {
    tod = tod+1;
    while (!empty(check) && minrank(check) <= tod)
      signal(check);
  }
}
```

Figure 5.8 Interval timer with priority wait.

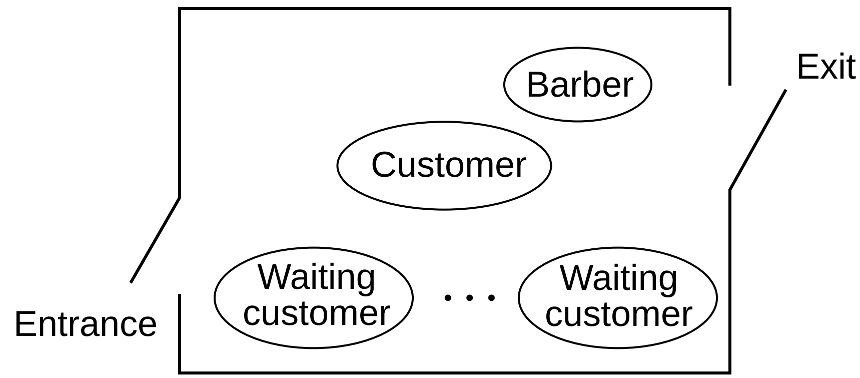

Figure 5.9 The sleeping barber problem.

```
monitor Barber_Shop {
  int barber = 0, chair = 0, open = 0;
  cond barber_available;    # signaled when barber > 0
  cond chair_occupied;      # signaled when chair > 0
  cond door_open;           # signaled when open > 0
  cond customer_left;       # signaled when open == 0
  procedure get_haircut() {
    while (barber == 0) wait(barber_available);
    barber = barber - 1;
    chair = chair + 1; signal(chair_occupied);
    while (open == 0) wait(door_open);
    open = open - 1; signal(customer_left);
  }
  procedure get_next_customer() {
    barber = barber + 1; signal(barber_available);
    while (chair == 0) wait(chair_occupied);
    chair = chair - 1;
  }
  procedure finished_cut() {
    open = open + 1; signal(door_open);
    while (open > 0) wait(customer_left);
  }
}
```

Figure 5.10 Sleeping barber monitor.

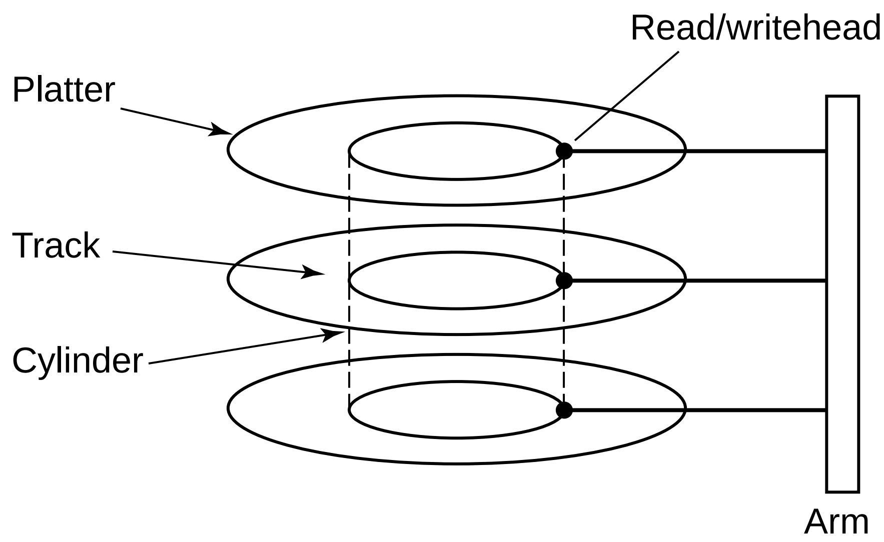

Figure 5.11 A moving-head disk.

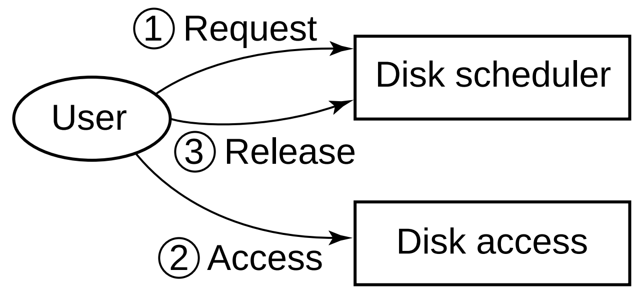

Figure 5.12 Disk scheduler as separate monitor.

```
monitor Disk_Scheduler {  ## Invariant DISK
  int position = -1, c = 0, n = 1;
  cond scan[2];   # scan[c] signaled when disk released
  procedure request(int cyl) {
    if (position == -1)  # disk is free, so return
      position = cyl;
    elseif (position != -1 && cyl > position)
      wait(scan[c],cyl);
    else
      wait(scan[n],cyl);
  }
  procedure release() {
    int temp;
    if (!empty(scan[c]))
      position = minrank(scan[c]);
    elseif (empty(scan[c]) && !empty(scan[n])) {
      temp = c; c = n; n = temp;       # swap c and n
      position = minrank(scan[c]);
      }
    else
      position = -1;
    signal(scan[c]);
  }
}
```

Figure 5.13 Separate disk scheduler monitor.

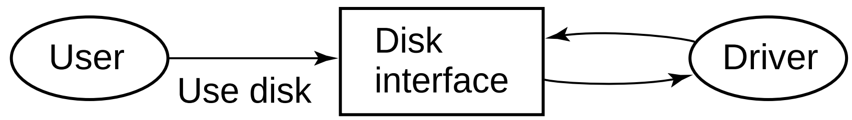

Figure 5.14 Disk scheduler as intermediary.

```
monitor Disk_Interface
  permanent variables for status, scheduling, and data transfer
  procedure use_disk(int cyl, transfer and result parameters) {
    wait for turn to use driver
    store transfer parameters in permanent variables
    wait for transfer to be completed
    retrieve results from permanent variables
  }
  procedure get_next_request(someType &results) {
    select next request
    wait for transfer parameters to be stored
    set results to transfer parameters
  }
  procedure finished_transfer(someType results) {
    store results in permanent variables
    wait for results to be retrieved by client
  }
}
```

Figure 5.15 Outline of disk interface monitor.

```
monitor Disk_Interface {
  int position = -2, c = 0, n = 1, args = 0, results = 0;
  cond scan[2];
  cond args_stored, results_stored, results_retrieved;
  argType arg_area; resultType result_area;
  procedure use_disk(int cyl; argType transfer_params;
                     resultType &result_params) {
    if (position == -1)
      position = cyl;
    elseif (position != -1 and cyl > position)
      wait(scan[c],cyl);
    else
      wait(scan[n],cyl);
    arg_area = transfer_params;
    args = args+1; signal(args_stored);
    while (results == 0) wait(results_stored);
    result_params = result_area;
    results = results-1; signal(results_retrieved);
  }
  procedure get_next_request(argType &transfer_params) {
    int temp;
    if (!empty(scan[c]))
      position = minrank(scan[c]);
    elseif (empty(scan[c]) && !empty(scan[n])) {
      temp = c; c = n; n = temp;      # swap c and n
      position = minrank(scan[c]);
      }
    else
      position = -1;
    signal(scan[c]);
```

Figure 5.17 Disk access using nested monitors.

```
#include <pthread.h>
#include <stdio.h>
#define SHARED 1
#define MAXSIZE 2000   /* maximum matrix size */
#define MAXWORKERS 4   /* maximum number of workers */

pthread_mutex_t barrier;  /* lock for the barrier */
pthread_cond_t go;        /* condition variable */
int numWorkers;           /* number of worker threads */
int numArrived = 0;       /* number who have arrived */

/* a reusable counter barrier */
void Barrier() {
  pthread_mutex_lock(&barrier);
  numArrived++;
  if (numArrived < numWorkers)
    pthread_cond_wait(&go, &barrier);
  else {
    numArrived = 0;  /* last worker awakens others */
    pthread_cond_broadcast(&go);
  }
  pthread_mutex_unlock(&barrier);
}

void *Worker(void *);
int size, stripSize;   /* size == stripSize*numWorkers */
int sums[MAXWORKERS];  /* sums computed by each worker */
int matrix[MAXSIZE][MAXSIZE];

/* read command line, initialize, and create threads */
int main(int argc, char *argv[]) {
  int i, j;
  pthread_attr_t attr;
```

```
monitor Disk_Access {
   permanent variables as in Disk_Scheduler;
   procedure doIO(int cyl; transfer and result arguments) {
      actions of Disk_Scheduler.request;
      call Disk_Transfer.read or Disk_Transfer.write;
      actions of Disk_Scheduler.release;
   }
}
```

Disk access monitor when using nested calls.

```
// basic read or write; no exclusion
class RWbasic {
  protected int data = 0;  // the "database"
  public void read() {
    System.out.println("read:  " + data);
  }
  public void write() {
    data++;
    System.out.println("wrote:  " + data);
  }
}

class Reader extends Thread {
  int rounds;
  RWbasic RW;  // a reference to an RWbasic object
  public Reader(int rounds, RWbasic RW) {
    this.rounds = rounds;
    this.RW = RW;
  }
  public void run() {
    for (int i = 0; i < rounds; i++) {
      RW.read();
    }
  }
}

class Writer extends Thread {
  int rounds;
  RWbasic RW;
  public Writer(int rounds, RWbasic RW) {
    this.rounds = rounds;
    this.RW = RW;
```

```java
// mutually exclusive read and write methods
class RWexclusive extends RWbasic {
  public synchronized void read() {
    System.out.println("read:  " + data);
  }
  public synchronized void write() {
    data++;
    System.out.println("wrote:  " + data);
  }
}

class Reader extends Thread {
  int rounds;
  RWexclusive RW;
  public Reader(int rounds, RWexclusive RW) {
    this.rounds = rounds;
    this.RW = RW;
  }
  public void run() {
    for (int i = 0; i < rounds; i++) {
      RW.read();
    }
  }
}
```

Exclusive readers/writers using Java.

```
// concurrent read or exclusive write
class ReadersWriters extends RWbasic {
  private int nr = 0;
  private synchronized void startRead() {
    nr++;
  }
  private synchronized void endRead() {
    nr--;
    if (nr == 0) notify(); // awaken waiting Writers
  }
  public void read() {
    startRead();
    System.out.println("read:  " + data);
    endRead();
  }
  public synchronized void write() {
    while (nr > 0)   // delay if any active Readers
      try { wait(); }
        catch (InterruptedException ex) {return;}
    data++;
    System.out.println("wrote:  " + data);
    notify();    // awaken another waiting Writer
  }
}
```

True readers/writers using Java.

`wait(cv)`	wait at end of queue
`wait(cv, rank)`	wait in order of increasing value of rank
`signal(cv)`	awaken process at front of queue then continue
`signal_all(cv)`	awaken all processes on queue then continue
`empty(cv)`	true if wait queue is empty; false otherwise
`minrank(cv)`	value of rank of process at front of wait queue

Table 5.1 Operations on condition variables.