Monitors

Andrews, Chapter 05

- More structure than semaphores.
- Can be implemented easily.
- Access to monitor variables is only through interface.
- Mutual exclusion of all monitor procedures is implicit: procedures in the same monitor cannot be executed concurrently.
- Condition synchronization is by **condition variables**.

```
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.

- Example monitor

Monitors

- Programs with monitors usually use two kinds of modules:
  active processes and passive monitors.
- Shared variables are inside the monitors.
- Processes communicate by calling procedures in the same monitor.
- Provided by Java.
- Provided in Unix.

Monitors

- The book uses a simple syntax for static monitors:

```
monitor mname {
  declarations of permanent variables
  initialization statements
  procedures
}
```

- And calling:

```
  call mname.opname(arguments)
```

Three Properties of Monitors

- Only procedure names are visible outside the monitor.
- Monitors may not access variables declared outside the monitor.
- Permanent variables are initalized before any procedures are called.

Monitor Invariants

- Truth of the invariant should be established by the initialization.
- After any procedure is called, the invariant should remain true.

Monitor Mutual Exclusion and Synchronization

- Mutual exclusion is implicit:
  no two procedures in a monitor execute concurrently.
- Synchronization is explicit:
  uses **condition variables.**
- Condition variables are declared:

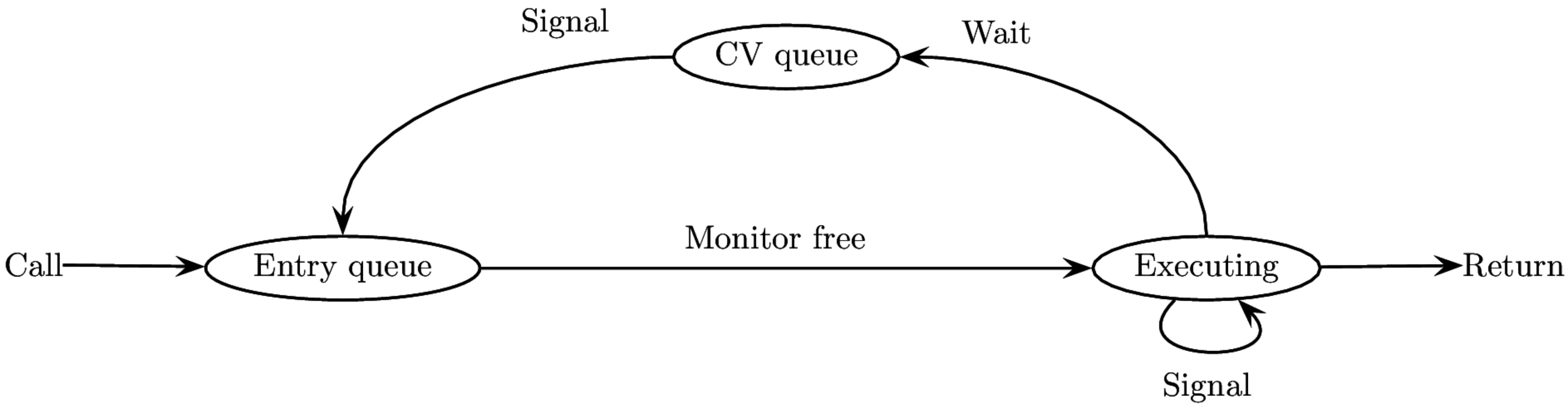
```
cond cv;
```

`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.

- Only called within the monitor.
- FIFO queue implicit, unless `rank` specified.

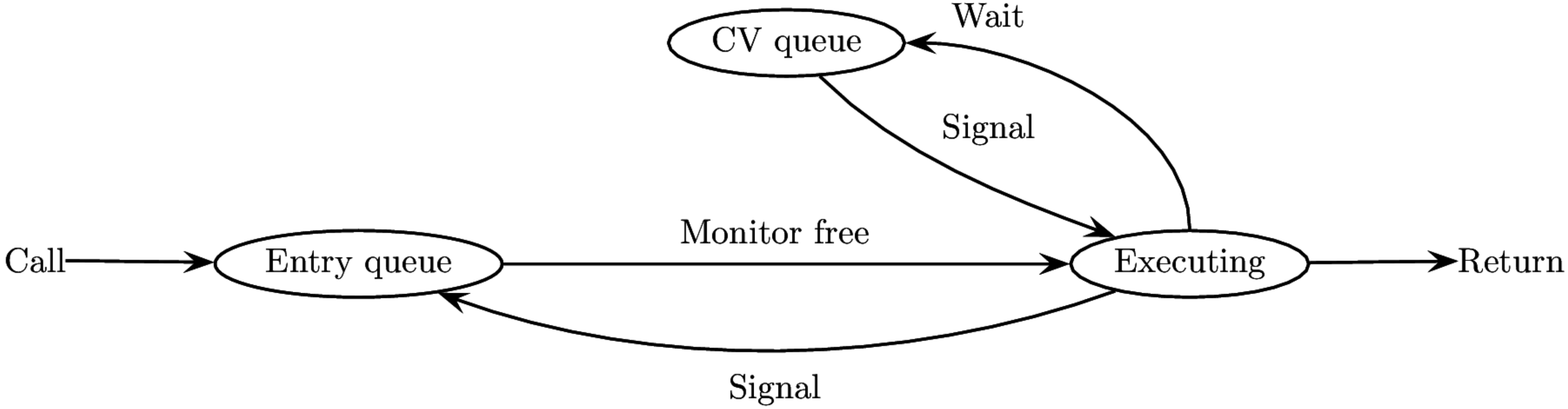
Signal and continue



Signal and continue

- The signaler continues and the signaled process waits.
- Nonpreemptive.

Signal and Wait



Signal and wait

- The signaler waits and the signaled process executes now.
- Preemptive.

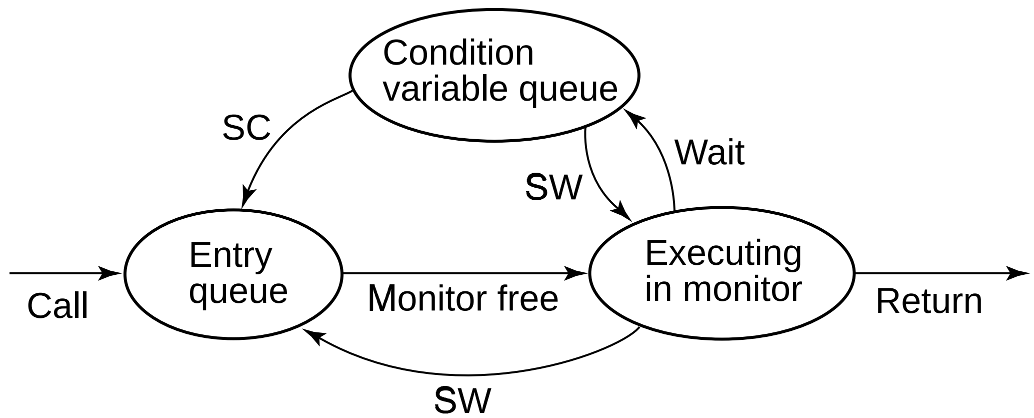


Figure 5.1 State diagram for synchronization in monitors.

- Signal and continue used in Java and Pthreads.
- If signaled process put at front of entry queue:
  signal and urgent wait

```
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.

- Works for both SC and SW.
- SW is FIFO, but not SC.
- With SW, `while` can be replaced by `if`

```
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.

Differences between `P` and `V` and `wait` and `signal`

- `wait` and `P`:
  - `wait` *always* delays until a `signal`
  - `P` delays only if semaphore is zero
- `signal` and `V`:
  - `signal` has no effect on an empty queue
  - `V` either awakens a process or increments the semaphore

Book assumes Signal and Continue

- SC used in Unix, Java, and Pthreads.
- Makes `signal_all` well-defined.
- It is compatible with priority-based scheduling.
- It has simpler formal semantics.
- Historically, SW was first proposed for use in monitors.

```
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.

- `wait` enclosed in `while`

- `signal` is merely a hint that condition *might* be met

  —could be sent at any time

```
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
  }
}
```

- Monitor invariant: $(nr = 0 \vee nw = 0) \wedge nw \leq 1$

Figure 5.5 Readers/writers solution using monitors.

- `signal` can be given at any time

```
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);
  }

}
```

- *SJN:* `turn` ordered by time $\wedge$ (`free` $\Rightarrow$ `turn` is empty)

Figure 5.6 Shortest-job-next allocation with monitors.

- `wait` not in `while`, why?

```
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.

- Uses a covering condition for all waiting processes.

```
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.

- Three ways to synchronize when conditions depend on local variables:

  1. Priority wait.

  —best

  2. Covering condition.

  —compact, correct solutions, possibly inefficient

  3. Record private condition variables for each waiting process.

  —more complex

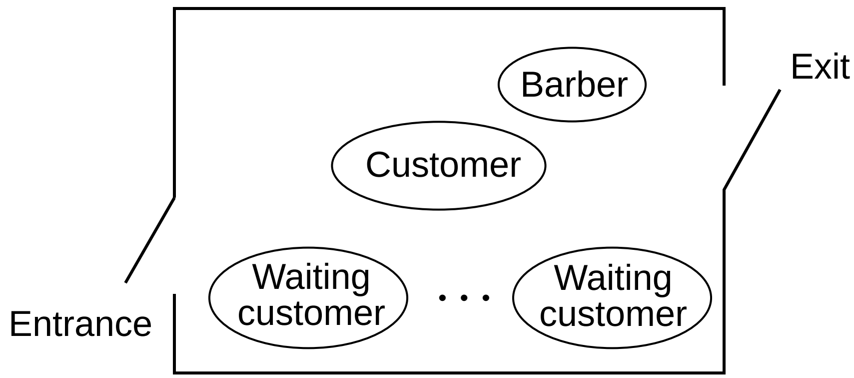


Figure 5.9 The sleeping barber problem.

- $C1$: `cinchair >= cleave` $\wedge$ `bavail >= bbusy >= bdone`
- $C2$: `cinchair <= bavail` $\wedge$ `bbusy <= cinchair`
- $C3$: `cleave <= bdone`

BARBER: $C1 \wedge C2 \wedge C3$

- `barber == (bavail - cinchair)` $\in \{0, 1\}$
- `chair == (cinchair - bbusy)` $\in \{0, 1\}$
- `open == (bdone - cleave)` $\in \{0, 1\}$

```
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.

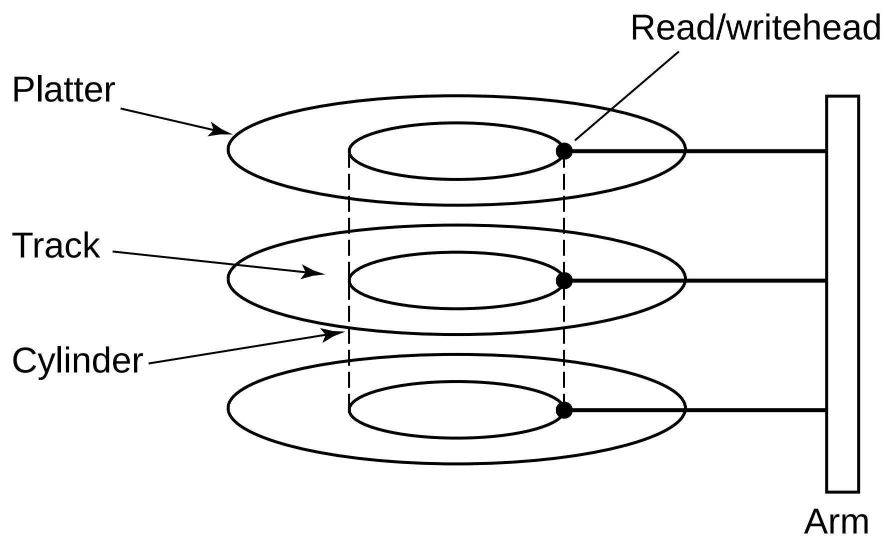


Figure 5.11 A moving-head disk.

- Closest job next
- SCAN, LOOK (elevator algorithm)
- CSCAN or CLOOK (circular algorithm)

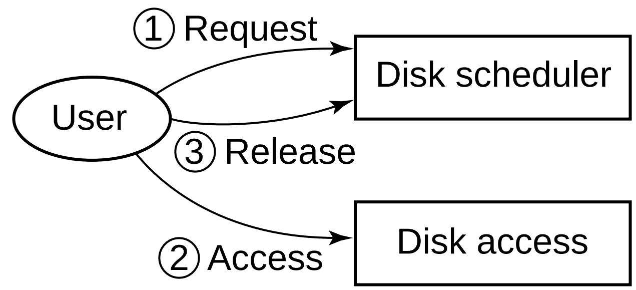


Figure 5.12 Disk scheduler as separate monitor.

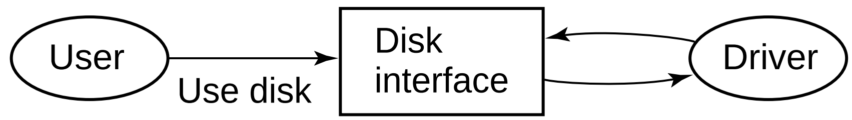


Figure 5.14 Disk scheduler as intermediary.

Figure 5.17 Disk access using nested monitors.

- *Open* and *Closed* monitors
- When a procedure in one monitor calls a procedure in a different monitor:

  If it releases the original monitor it is an *open* call.

  If it retains the lock on the original monitor it is a *closed* call.

Java: synchronized methods

```
class Counter {
  private int c = 0;
  public void increment() {
    c++;
  }
  public int value() {
    return c;
  }
}

class SynchronizedCounter {
  private int c = 0;
  public synchronized void increment() {
    c++;
  }
  public synchronized int value() {
    return c;
  }
}
```

Java: `wait` and `notify`

- Methods of the class `Object`
- Can only be used in `synchronized` methods
- Only a single delay queue per object
- There are no condition variables
  - exactly one (implied) condition variable per object
- FIFO not guaranteed
- `notifyAll`
- Nested calls are closed calls

```java
// 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;
  }
  public void run() {
    for (int i = 0; i < rounds; i++) {
      RW.write();
    }
  }
}
class Main {
  static RWbasic RW = new RWbasic();
  public static void main(String[] args) {
    int rounds = Integer.parseInt(args[0],10);
    new Reader(rounds, RW).start();
    new Writer(rounds, RW).start();
  }
}
```

Does not actually create parallel readers and writers. (See java programs in repo.)

```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.

```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.

Pthreads: locks and condition variables

Lock

```
...
pthread_mutex_t mutex;
...
pthread_mutex_init(&mutex, NULL);
...
```

Condition variable

```
...
pthread_cond_t cv;
...
pthread_cond_init(&cv, NULL);
...
```

Thread 0

```
...
pthread_mutex_lock(&mutex);
...
pthread_cond_wait(&cv, &mutex);
...
pthread_mutex_unlock(&mutex);
```

Thread 1

```
...
pthread_mutex_lock(&mutex);
...
pthread_cond_signal(&cv);
...
pthread_mutex_unlock(&mutex);
```

```c
#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;
  pthread_t workerid[MAXWORKERS];

  /* set global thread attributes */
  pthread_attr_init(&attr);
  pthread_attr_setscope(&attr, PTHREAD_SCOPE_SYSTEM);

  /* initialize mutex and condition variable */
  pthread_mutex_init(&barrier, NULL);
  pthread_cond_init(&go, NULL);

  /* read command line */
  size = atoi(argv[1]);
  numWorkers = atoi(argv[2]);
  stripSize = size/numWorkers;

  /* initialize the matrix */
  for (i = 0; i < size; i++)
    for (j = 0; j < size; j++)
      matrix[i][j] = 1;

  /* create the workers, then exit main thread */
  for (i = 0; i < numWorkers; i++)
    pthread_create(&workerid[i], &attr,
                   Worker, (void *) i);
```

```
    pthread_exit(NULL);
  }

  /* Each worker sums the values in one strip.
     After a barrier, worker(0) prints the total */
  void *Worker(void *arg) {
    int myid = (int) arg;
    int total, i, j, first, last;

    /* determine first and last rows of my strip */
    first = myid*stripSize;
    last = first + stripSize - 1;

    /* sum values in my strip */
    total = 0;
    for (i = first; i <= last; i++)
      for (j = 0; j < size; j++)
        total += matrix[i][j];
    sums[myid] = total;
    Barrier();
    if (myid == 0) {  /* worker 0 computes the total */
      total = 0;
      for (i = 0; i < numWorkers; i++)
        total += sums[i];
      printf("the total is %d\n", total);
    }
  }
```

Figure 5.18 Parallel matrix summation using Pthreads.