```
/*-----------------------------------------------------
 * Uppgifts-scenario 1
 * http://www.ida.liu.se/~TDDB68/exam/exam-081018.pdf
 *-----------------------------------------------------

Once upon a time, a small cinema with a single ticket
counter used a seat reservation program for a (for
simplification) single movie presentation per day.

Initially, all seats are free. This is fixed by the
'initialize_seats' function, which is executed before
the cinema ticket counter is opened every day.

Before reserving a seat for a new visitor, the clerk at
the ticket counter queries the current status of free
and booked seats (i.e., the contents of seat_booked),
which is displayed graphically. The function
'show_seat_booking_status' fill in the occupied seats
in the graphical figure before every booking.

After asking the customer for his/her preferences, the
clerk chooses and books a seat that was displayed as
free, which is done by the function 'book_seat'.

As time goes by, the number of visitors grows and the
cinema moves to a building with a larger room (= larger
values for nrows and seats per row). To reduce queueing
at the ticket counter, a second ticket counter is
added, and even a web interface for selfbooking is
created. All these become additional clients of the
seat reservation program, which now will run on a
dual-core (!) server running a modern operating system
with preemptive thread scheduling that supports
semaphores and mutual exclusion locks. The revised seat
reservation program shall be multithreaded, such that
the above data structures are shared and several book
seat calls may be issued concurrently. It is your task
to make the seat reservation program thread-safe.

*/
```

```
const int nrows = 8;           // Number of rows of seats
const int seats_per_row = 12; // Number of seats per row

// 2D array of seats, seat_booked[i][j] is 1 iff seat in
// row i, column j is booked
boolean seat_booked[ nrows ][ seats_per_row ];

void initialize_seats ( )
{
  for (int i=0; i<nrows; i++)
    for (int j=0; j<seats_per_row; j++)
      seat_booked[i][j] = 0; // free
}

void show_seat_booking_status ( )
{
  for (int i=0; i<nrows; i++)
    for (int j=0; j<seats_per_row; j++)
      display_status( i, j, seat_booked[i][j] );
}

boolean book_seat ( int i, int j )
{
  if (i<0 || i>=nrows || j<0 || j>=seats_per_row) {
    error_message("no such seat"); return 0;
  }
  if (seat_booked[i][j]) {
    error_message("seat already booked"); return 0;
  }
  seat_booked[i][j] = 1;
  return 1; // successful - can now print the ticket.
}

//-------------------------------------------------

// koden motvarande detta körs i en atomisk instruktion
int test_and_set (int* mem)
{
  int test = *mem; // remember old value
  *mem = true;     // set to 1
  return test;     // return old value
}

// koden motvarande detta körs i en atomisk instruktion
void swap (int* a, int* b)
{
  int save = *a;
  *a = *b;
  *b = save;
}
```

```
/*-----------------------------------------------------
 * Uppgifts-scenario 2
 *-----------------------------------------------------

Bertram har implementerat en associativ container (map)
där man kan stoppa in en pekare till en godtycklig
informationsstruktur och få tillbaka en nyckel
(hitta_ledig_plats). Man kan även fråga efter vilken
informationsstruktur som är kopplad till en viss nyckel
(hämta_pekare) och ta bort pekaren som hör till en viss
nyckel (frigör_plats).

Ett multitrådat serverprogram använder Bertram's
container för att hålla reda på alla aktiva
sessioner. När en användare ansluter till servern läggs
information om användaren och användarens dator i en
informationsstruktur och sätts in i containern. Nyckeln
som då fås används som sessions-ID. Nästa gång
användaren komunicerar med servern skickas sessions-ID
(nyckeln) med och serverprogrammet kan anpassa sitt
svar till just den användaren (t.ex. hålla reda på
varor i en kundvagn som lagras i
informationsstrukturen). Varje begäran servern får
utförs i en egen tråd. För att undvika arbete med att
hela tiden skapa och ta bort trådar samlas alla trådar
som är klara i en pool och återanvänds när en ny
begäran kommer in.

Se till att containern blir trådsäker.
*/
```

```
typedef container
{
  void* plats[SIZE]; // alla index initieras till NULL
}

void* hämta_pekare(container* k, int nyckel)
{
  if ( nyckel >= 0 && nyckel < SIZE ) // giltig nyckel?
    return k->plats[nyckel];
  else
    return NULL; // värde för nyckeln fanns inte
}

void* frigör_plats(container* k, int nyckel)
{
  void* info;

  if ( nyckel >= 0 && nyckel < SIZE ) // giltig nyckel?
  {
    info = k->plats[nyckel];
    k->plats[nyckel] = NULL;
    return info; // returnera gamla värdet på platsen
  }
  return NULL; // värde för nyckeln fanns inte
}

int hitta_ledig_plats(container* k, void* info)
{
  int i;

  for (i = 0; i < SIZE; ++i)
  {
    if ( k->plats[i] == NULL ) // är det ledigt?
    {
      k->plats[i] = info;
      return i; // indexet blir nyckel
    }
  }
  return -1;
}

//-----------------------------------------------------

// koden motvarande detta körs i en atomisk instruktion
int compare_and_swap (int* mem, int oldval, int newval)
{
  int old_mem_val = *mem;
  if (old_mem_val == oldval)
    *mem = newval;
  return old_mem_val;
}
```