Synchronization

Exercises on semaphores and mutexes

Stefano Quer
Dipartimento di Automatica e Informatica
Politecnico di Torino

License Information

This work is licensed under the license

CC BY-NC-ND 4.0

Attribution-NonCommercial-NoDerivatives 4.0 International

This license requires that reusers give credit to the creator. It allows reusers to copy and distribute the material in any medium or format in unadapted form and for noncommercial purposes only.

BY: Credit must be given to you, the creator.

NC: Only noncommercial use of your work is permitted.
Noncommercial means not primarily intended for or directed towards commercial advantage or monetary compensation.

ND: No derivatives or adaptations of your work are permitted.

To view a copy of the license, visit:
https://creativecommons.org/licenses/by-nc-nd/4.0/?ref=chooser-v1

Exercise 01

- Implement C++ program that
  - Runs 1 thread TA and 1 thread TB
  - TA and TB include an infinite cycle in which they display one single character, 'A' or 'B', respectively
  - Synchronize threads such that for each set of 3 characters, there is 1 character A and 2 characters B in any position
  - Execution example

```
pgrm
    ABB
    BBA
    BAB
    etc.
```

Solution

```
#include <iostream>
#include <semaphore>
#include <thread>
#include <unistd.h>

using std::cout;
using std::endl;

std::counting_semaphore sa{1}, sb{2}, me{1};
int n;

static void TA (int);
static void TB (int);
```

To "sleep" for a random time

Mutexes cannot be used because they must be locked and unlocked by the same thread

Counter

2 Threads
2 semaphores
1 mutex (semaphore)

Solution

```
int main (int argc, char **argv) {
  int n1, n2;

  if (argc != 2) {
    fprintf (stderr, "Syntax: %s num_threads\n", argv[0]);
    return (1);
  }
  n1 = atoi(argv[1]);
  n2 = 2 * n1;
  n = 0;

  std::thread ta (TA, n1);
  std::thread tb (TB, n2);

  ta.join();
  tb.join();

  return (0);
}
```

To avoid running forever
TA iterates n1 times
TB iterates n2 times

Solution

```
static void TA (int nc) {
  for (int i=0; i<nc; i++) {
    sleep (rand()%2);
    sa.acquire();
    me.acquire();
    cout << "A";
    n++;
    if (n>=3) {
      cout << endl;
      n = 0; sa.release(); sb.release(); sb.release();
    }
    me.release();
  }
  return;
}
```

Wait for a random time

If TA starts

It must not start with TB

The last thread wakes-up one A and two B threads

Solution

```
static void TB (int nc) {
  for (int i=0; i<nc; i++) {
    sleep (rand()%2);
    sb.acquire();
    me.acquire();
    cout << "B";
    n++;
    if (n>=3) {
      cout << endl;
      n = 0;
      sa.release(); sb.release(); sb.release();
    }
    me.release();
  }
  return;
}
```

Wait for a random time

If TB starts

It must not start with TA

The last thread wakes-up one A and two B threads

Exercise 02

Exam of September 08, 2023

- A C program can execute four different threads
  - TP (thread plus), TM (thread minus), TS (thread star), and TNL (thread newline)
- Each thread is organized through an infinite cycle containing synchronization instructions but a single IO instruction
  - Thread TP displays a "+"
  - Thread TM displays a "-"
  - Thread TS displays a "*"
  - Thread TNL displays a "\n" (endl)

Exercise 02

- Synchronize the four threads to print the following sequence of lines

```
++++++++++
----------
**********
++++++++++
----------
**********
etc.
```

  - Where the number of characters on each row is given as a parameter to the main program (e.g., 10)

Solution

```
#include <iostream>
#include <semaphore>
#include <thread>
#include <unistd.h>

using std::cout;
using std::endl;

std::counting_semaphore sp{1}, sm{0}, ss{0}, snl{0};

static void TP (int);
static void TM (int);
static void TS (int);
static void TNL ();
```

4 Threads
4 Semaphores
SP (+) is the one to start

Solution

```
int main (int argc, char **argv) {
  int n;
  if (argc != 2) {
    ... error ...
  }
  n = atoi(argv[1]);
  std::thread tp (TP, n);
  std::thread tm (TM, n);
  std::thread ts (TS, n);
  std::thread tnl (TNL);
  tp.join();
  tm.join();
  ts.join();
  tnl.join();
  return (0);
}
```

Threads never stop; but if we do not wait, we return and we stop all threads (there is no pthread_exit)

Solution

```
static void TP (int n) {
  int np = 0;
  while (1) {
    sp.acquire();
    cout << "+";
    np++;
    if (np<n) {
      sp.release();
    } else {
      np = 0;
      snl.release();
    }
  }
  return;
}
```

Re-wake up TP

Reset the number of calls for TP and call TNL

Solution

```
static void TM (int n) {
  int nm = 0;
  while (1) {
    sm.acquire();
    cout << "-";
    nm++;
    if (nm<n) {
      sm.release();
    } else {
      nm = 0;
      snl.release();
    }
  }
  return;
}
```

Re-wake up TM

Reset the number of calls for TM and call TNL

Solution

```
static void TS (int n) {
  int ns = 0;
   while (1) {
    ss.acquire();
    cout << "*";
    ns++;
    if (ns<n) {
      ss.release();
    } else {
      ns = 0;
      snl.release();
    }
  }
  return;
}
```

Re-wake up TS

Reset the number of calls for TS and call TNL

Solution

```
static void TNL () {
  int nnl = 0;
  while (1) {
    snl.acquire(); nnl++; cout << endl;
    sleep (rand()%2);
    if (nnl==1) {
      sm.release();
    } else {
      if (nnl==2) {
        ss.release();
      } else {
        sp.release(); nnl = 0;
      }
    }
  }
  return;
}
```

POSIX
(we can use C++ to sleep)

Wake up TM

Wake up TS

Wake up TP
and restart

Exercise 03

- Fairness consideration on synchronization primitives
  - C++ synchronization primitives are unfair
    - Some threads can lock a mutex more often than others
      - A simple experiment on Linux shows that if threads repeatedly try to lock the same mutex, some threads lock the mutex 1.13x more often than others
    - Some threads can lock a semaphore or a spinlock 3.91x more often than others

Exercise 03

- Implement a **priority semaphore**, i.e., a semaphore in which
  - Each thread has an intrinsic priority
    - The priority is an integer value
    - The **higher** priority corresponds to the **lower** value
  - Unlocking is done **in order** following the thread priority

Solution

- Core idea
  - The semaphore must have a **priority queue** associated with it, where threads await to be signalled
  - When a call to the signal function wakes-up a thread, threads must be woken-up following their priority
    - We have to awake the threads with the higher priority among the ones waiting on that semaphore

In C++ lock and unlock must be called by the same thread.
We should use C++ semaphores, but semaphores are not copyable

Solution

```
#include <iostream>
#include <algorithm>
#include <vector>
#include <map>
#include <thread>
#include <semaphore>

using std::cout;
using std::endl;
...

const int TIME = 3;

map<int,std::unique_ptr<std::binary_semaphore>> my_sem;
std::mutex m;
```

C++20 semaphores are neither copyable nor movable. We need to use dynamic memory allocation carefully

Solution

Worker running threads

```
static void worker (int i, int priority) {
  m.lock();
  cout << "Locking thread " << i <<
          " with priority " << priority << endl;
  m.unlock();
  my_sem.insert
    ({priority,std::make_unique<std::binary_semaphore>(0)});
  (*my_sem[priority]).acquire();
  m.lock();
  cout << "          Unlocked thread " << i <<
          " with priority " << priority << endl;
  m.unlock();
  return;
}
```

Insert the worker in a (sorted) map

Send the worker to sleep

When the worker runs it does what it has to do

Solution

Main: Part 1

```
int main (int argc, char *argv[]) {
  int i, priority;
  if (argc != 2) {
    cout << "Syntax: " << argv[0] << " num_threads\n";
    return (1);
  }
  int n = atoi (argv[1]);
  vector<thread> pool;
  for (i=0; i<n; i++) {
    priority = (i+1) * 10;
    pool.emplace_back([i, priority] { worker (i, priority); });
  }
  std::this_thread::sleep_for
    (std::chrono::seconds(rand()%TIME));
```

Running workers

From POSIX sleep to C++

Put the thread in a sleep status for rand()%TIME seconds

Solution

Main: Part 2

```
  i = 0;
  for (const auto &t : my_sem) {
    m.lock();
    cout << "     Unlocking thread " << i++ <<
            " with priority " << t.first << endl;
    m.unlock();
    (*(t.second)).release();
  }
  for (i=0; i<n; i++) {
    pool[i].join();
  }
  cout << "Main exits." << endl;
  return (1);
}
```

Wake-up workers following the priority order

Release worker

Wait workers

Solution

```
Locking thread 0 with priority 10
Locking thread 6 with priority 70
Locking thread 2 with priority 30
Locking thread 1 with priority 20
Locking thread 9 with priority 100
...
     Unlocking thread 0 with priority 10
     Unlocking thread 1 with priority 20
     ...
          Unlocked thread 0 with priority 10
          Unlocked thread 1 with priority 20
     Unlocking thread 5 with priority 60
     ...
          Unlocked thread 5 with priority 60
          Unlocked thread 9 with priority 100
          Unlocked thread 4 with priority 50
          Unlocked thread 3 with priority 40
          ...
Main exits.
```

Output

Locking the threads

Unlocking them ...

... which then start

Exercise 04

- Write a program to implement an election algorithm that elects a leader thread
  - The system runs N threads
  - Each thread has its
    - Thread identifier
    - Rank, i.e., an integer value randomly generated
  - To elect the leader, each thread must
    - Compare its rank value with the current value in **best_rank** to decide if it is the leader or not
      - To do that, it synchronizes with all the other threads
      - It re-starts when the election process is completed (i.e., all other threads have updated the value of **best_rank**)

Exercise 04

- When all threads have done their job, each thread displays
  - Its identifier and its rank value
  - The leader thread identifier and its rank value
- Restriction
  - Threads cannot access the rank value of other threads, only the current best thread rank value is available in a global variable **best_rank** together with the corresponding thread identifier
  - Hint: Referring to a voting algorithm, use a global variable to count the number of threads that completed their voting process

Solution

```
#include <iostream>
...
struct Best {
  int rank;
  std::thread::id id;
  int num_votes;
  std::mutex mtx;
};
Best* best;
std::counting_semaphore<0> barrier(0);

int max_random(int max) {
  static std::random_device rd;
  static std::mt19937 gen(rd());
  return std::uniform_int_distribution<int>(0, max - 1)(gen);
}
```

Thread structure

Semaphore to make threads wait

Solution

```
int main() {
  best = new Best();
  best->rank = 0;
  best->num_votes = 0;
  auto seed =
    std::chrono::high_resolution_clock::now().
                                        time_since_epoch().count();
  std::mt19937 gen(seed);
  std::vector<std::thread> threads;
  for (int i=0; i<10; i++) {
    int rank = i+1; threads.emplace_back (process, rank);
  }
  for (auto& t : threads) t.join();
  delete best;
  return 0;
}
```

Running 10 threads

Worker thread

Solution

Worker thread

```
void process(int rank) {
  std::thread::id id = std::this_thread::get_id();

  // Lock the mutex to ensure exclusive access
  best->mtx.lock();
  if (rank > best->rank) {
    best->rank = rank;
    best->id = id;
  }
  best->num_votes++;
```

Update the leader rank

This thread is the new leader

Solution

```
 if (best->num_votes < 10) {
     // Wait for all threads to vote
     std::cout << "Thread WAITING id = " ...
     best->mtx.unlock();
     barrier.acquire();
     std::cout << "    Thread RELEASED id = " ...
} else {
     std::cout << "  Last Thread id = " .,.
     best->mtx.unlock();
     // Release all waiting threads
     for (int i = 0; i < 9; i++) {
       barrier.release();
     }
   }

}
```

Semaphore to make threads wait

The last thread wakes up all threads

Solution

Output

```
> ./imp1
WAITING  id = 443164591808 rank =  2 -- leader id = 443164591808 rank =  2
WAITING  id = 443156199104 rank =  3 -- leader id = 443156199104 rank =  3
WAITING  id = 443172984512 rank =  1 -- leader id = 443156199104 rank =  3
WAITING  id = 443147806400 rank =  4 -- leader id = 443147806400 rank =  4
WAITING  id = 443139413696 rank =  5 -- leader id = 443139413696 rank =  5
WAITING  id = 443014657728 rank =  6 -- leader id = 443014657728 rank =  6
...
Last     id = 443105842880 rank = 10 -- leader id = 443105842880 rank = 10
RELEASED id = 443164591808 rank =  2 -- leader id = 443105842880 rank = 10
RELEASED id = 443147806400 rank =  4 -- leader id = 443105842880 rank = 10
RELEASED id = 443131020992 rank =  7 -- leader id = 443105842880 rank = 10
RELEASED id = 443156199104 rank =  3 -- leader id = 443105842880 rank = 10
...
```