Προγραμματισμός Υπολογιστών με C++

ΠΡΟΧΩΡΗΜΕΝΑ ΘΕΜΑΤΑ

Γεώργιος Παπαϊωάννου (2015-16)
gepap@aueb.gr

Περιεχόμενο Παρουσίασης

- Περιγραφή:
  - Shared Pointers
  - Νήματα
  - Αποκλειστική πρόσβαση σε πόρους (Mutexes)
  - Move assignment
- Τελευταία ενημέρωση: Δεκ 2017

Shared Pointers

Υλοποιούν τη λειτουργικότητα ενός pointer που κρατά reference counting και αυτόματα διαγράφει τον εαυτό του όταν:

- Μηδενιστούν οι αναφορές σε αυτόν
- Ρητά διαγραφεί

Εξ αρχής κατασκευή shared pointer

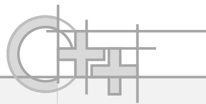
```
shared_ptr<MyClass> p = make_shared<MyClass>();
shared_ptr<MyClass> p_i =
    shared_ptr< MyClass >(new MyClass [100]);
```

“Copy” constructor από κανονικό pointer

Shared Pointers

- Οι shared pointers δουλεύουν όπως οι αναφορές στη Java
  - Μόνο που αντί τη διαγραφή τους να την αναλαμβάνει κάποιος garbage collector, εδώ το κάθε στιγμιότυπο αμέσως διαγράφει τη μνήμη του αν μηδενιστούν οι αναφορές σε αυτό (*)
  - Δε χρειάζονται delete
- Είναι πιο αργοί στη χρήση σε σχέση με τους συμβατικούς pointers

(*) Ανάλογα με τον allocator που χρησιμοποιείται. Στην ουσία, μπορούμε να υλοποιήσουμε ολόκληρο gc σύστημα από πίσω μόνοι μας…



Shared Pointers - Παράδειγμα

```
#include <memory>
#include <iostream>
#include <string>
using namespace std;

shared_ptr<string> CreatePointer()
{
    return make_shared<string>("foo");
}


void main()
{
    shared_ptr<string> p_s = CreatePointer();

    cout << *p_s << '\n';
    cout << "ref. count: " << p_s.use_count() << '\n';   // τυπώνει ref. count: 1

    shared_ptr<string> p2 = p_s;
    cout << "ref. count: " << p_s.use_count() << '\n';   // τυπώνει ref. count: 2


    getchar();
}
```

Lvalues και Rvalues

- Χρήσιμος σύνδεσμος:

http://www.cprogramming.com/c++11/rvalue-references-and-move-semantics-in-c++11.html

- Μια **Lvalue** είναι μια μη προσωρινή μεταβλητή στην οποία αναθέτουμε ένα προσωρινό αποτέλεσμα, το **Rvalue**
- Τη διεύθυνση μιας Lvalue μπορούμε να τη διαβάσουμε και να τη χρησιμοποιήσουμε, ενώ μιας Rvalue χάνεται
- Μπορούμε να το σκεφτούμε ως το αριστερό και δεξιό τελεστέο μιας εξίσωσης:

  a = b

  a: Lvalue, b: Rvalue

Rvalues και Αναθέσεις

- Πολύ συχνά, κατά τις αναθέσεις, τα Rvalues είναι αναφορές σε προσωρινά αντικείμενα
  - Τα αντικείμενα αυτά χάνονται μετά την «κατανάλωσή» τους

```
string getName (){
    return "Alex";
}
…
string name = getName();
```

Επιστροφή προσωρινού αντικειμένου: Rvalue reference

Αντιγραφή αντικειμένου (εδώ με ctor)

Rvalue References

- Κανονικά δε μπορούμε να αναθέσουμε προσωρινά δεδομένα σε μια αναφορά
- string & name = getName(); // Σφάλμα!
- Στη C++ 11, ορίζεται ένας νέος τύπος αναφοράς, η Rvalue reference που μας επιτρέπει να το κάνουμε
- string && name = getName();
- Μας βοηθάει να προσδιορίζουμε ρητά (κυρίως σε ορίσματα συναρτήσεων) ότι ένα δεδομένο είναι προσωρινό

Move Assignment Και Construction

- Κατά την αντιγραφή μεγάλων αντικειμένων όπου απαιτείται deep copy, ο copy constructor και copy assignment operator πρέπει να:
  - Δεσμεύσουν μνήμη
  - Να αντιγράψουν δεδομένα
  - Να αποδεσμεύσουν παλιά μνήμη (copy assignment)
- Αν γνωρίζουμε ότι η Rvalue είναι προσωρινή, μπορούμε απλά να της «κλέψουμε» τους πόρους και να γλυτώσουμε δέσμευση μνήμης και αντιγραφή

Move Assignments και Constructors

- Επιπρόσθετα με τον Copy assignment operator και Copy constructor, μπορούμε να έχουμε και Move εκδόσεις τους
- Παίρνουν τα δεδομένα από το όρισμα αντί να τα αντιγράφουν
- Η επιλογή εξαρτάται από το αν το όρισμα είναι Rvalue reference ή όχι

Παράδειγμα Υλοποίησης Copy Assignment

```
class ArrayWrapper
{
private:
    int *_p_vals;
    int _size;
public:
    // copy constructor
    ArrayWrapper (const ArrayWrapper& other)
            : _p_vals( new int[ other._size  ] )
            , _size( other._size )
        {
            for ( int i = 0; i < _size; ++i )
            {
                _p_vals[ i ] = other._p_vals[ i ];
            }
        }
        ~ArrayWrapper ()
        {
            if (_p_vals)
                delete [] _p_vals;
        }
};
```

Δέσμευση μνήμης

Αντιγραφή δεδομένων

Παράδειγμα Υλοποίησης Move Assignment

```
class ArrayWrapper
{
private:
    int *_p_vals;
    int _size;
public:
     ...
    // move constructor
     ArrayWrapper (ArrayWrapper&& other)
        : _p_vals( other._p_vals  )
        , _size( other._size )
    {
        other._p_vals = NULL;
        other._size = 0;
    }
    ~ArrayWrapper ()
    {
        if (_p_vals)
          delete [] _p_vals;
    }
};
```

Συνυπάρχει με τον copy constructor και επιλέγεται όταν το όρισμα είναι προσωρινή μεταβλητή

Κλέβει τον pointer του ορίσματος

Ο μηδενισμός του Pointer του ορίσματος μας προφυλάσσει από εσφαλμένη διαγραφή του στο αντικείμενο other

Move Assignment Operator

- Αντί να αντιγράφει το δεξί μέλος στη μεταβλητή ανάθεσης, «κλέβει» τους πόρους από το δεξί μέλος

```
struct A {
   string s;
   A() : s("test") { }
   A& operator=(A&& other){
      s = move(other.s);
      return *this;
   }
};
```

Δήλωση MOVE assignment

Ρητή κλήση του MOVE operator, αν υπάρχει
Αναγκαστική μετατροπή (cast) σε Rvalue reference

Move Assignment Operator

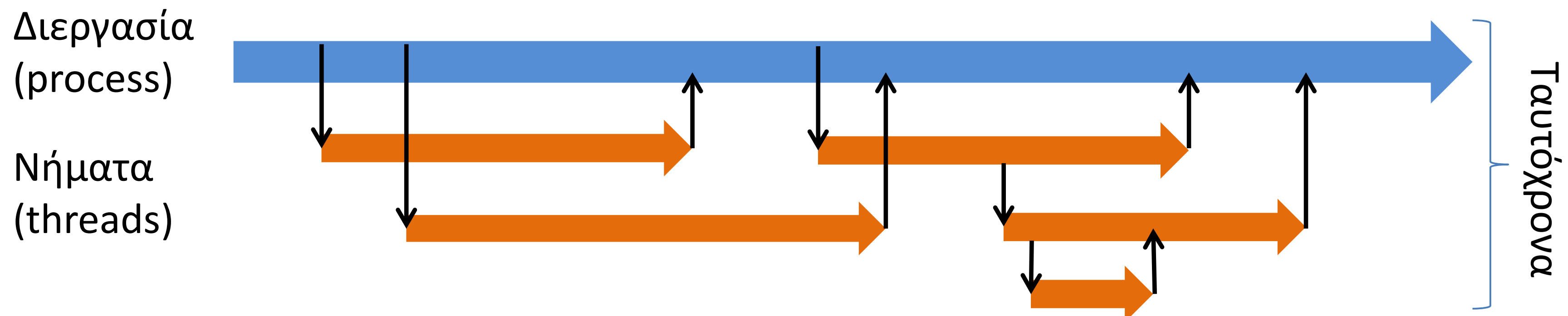
```
struct A {
    string s;
    A() : s("test") { }
    A& operator=(A&& other){
        s = move(other.s);
        return *this;
    }
};
void main() {
    A f;
    A g;
    f = A();
    f = g;    }
```

Επιλογή του MOVE = operator από τον compiler. Το A() είναι προσωρινό αντικείμενο.

Επιλογή του COPY = operator από τον compiler. Το g έχει λόγο να συνεχίσει να υπάρχει.

Νήματα (Threads)

- Τα νήματα αποτελούν μια εύκολη και γρήγορη μορφή εκτέλεσης κώδικα **παράλληλα**, μέσα στην ίδια τη διαδικασία της εφαρμογής
  - Ξεκινάνε γρήγορα
  - Χρησιμοποιούν την κοινή περιοχή μνήμης της διεργασίας που τρέχει η εφαρμογή
  - Παρέχουν εύκολους μηχανισμούς ελέγχου και συγχρονισμού



Νήματα στη C++ (C++11)

- Υλοποιούνται με την κλάση **thread** (header: <thread>)
- Ένα thread υποχρεωτικά το αρχικοποιούμε με μια συνάρτηση την οποία θα εκτελέσει
  - Η συνάρτηση αυτή καθορίζει τη «ζωή» του thread
  - Όταν τελειώσει, «κλείνει» το thread

Δημιουργία Νημάτων στη C++ (C++11)

```
void Foo(int a) {
    long b = a;
    for (int i = 0; i < 100; i++)
        cout << (b*=a);
}


void main() {
    std::thread th = std::thread(Foo, 2);
}
```

Κατασκευαστής

Όνομα συνάρτησης που θα τρέξει

Ορίσματα της συνάρτησης που θα τρέξει

Δημιουργία Νημάτων στη C++ - Τρόπος 2

```
void Foo(int a) {
    long b = a;
    for (int i = 0; i < 100; i++)
        cout << (b*=a);
}


void main() {
    thread th;
    th = thread(Foo, 2);
}
```

Προκαθορισμένος κατασκευαστής

“Move” assignment:
Το προσωρινό αντικείμενο (δεξιά του =), ακυρώνεται μετά την ανάθεση

Οποιαδήποτε άλλη ανάθεση δε θα είχε νόημα. Διαφορετικά τι θα παρίστανε το προσωρινό αντικείμενο τύπου thread??!

Νήματα – Η μέθοδος join

- Είδαμε πώς να ξεκινάμε νήματα. Πώς τα συγχρονίζουμε όμως;
  - Πρέπει να έχουν ένα συγκεκριμένο σημείο συνάντησης, όπου η συνάρτηση που ξεκινάει ένα thread, περιμένει το (τα) thread(s) να ολοκληρώσει τη δουλειά του
- Αυτός είναι ο ρόλος της μεθόδου join

Νήματα – Η μέθοδος join

```
void EstimatePI(double * res, long iterations);

void main() {
    double result[2];
    thread t = thread(EstimatePI, result, 1000);
    EstimatePI(result + 1, 1000);

    t.join();
    double final = (result[0] + result[1]) / 2.0;
}
```

Ξεκινάμε ένα thread να τρέξει την EstimatePI

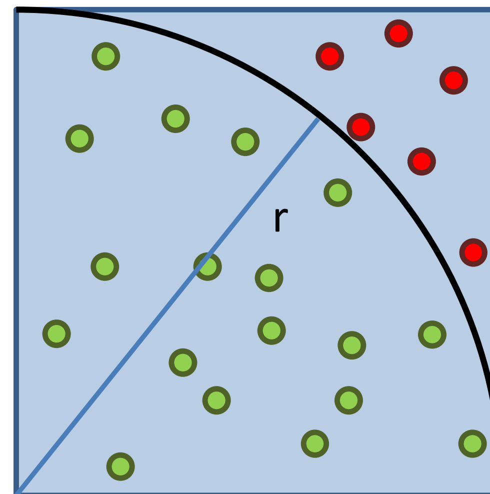
Τρέχουμε την EstimatePI στο υπάρχον thread

Δεν ξέρουμε τι θα τερματίσει πρώτο. Γι αυτό πρέπει να βάλουμε ένα σημείο συνάντησης εδώ πριν χρησιμοποιήσουμε τα αποτελέσματα και από τους 2 υπολογισμούς

Εφαρμογή: Στατιστικός υπολογισμός του π

- Ένας τρόπος να υπολογίσουμε την τιμή του π είναι με τη μέθοδο του dart throwing:
  - Παίρνουμε τυχαία και ομοιόμορφα δείγματα σε ένα τετράγωνο πλευράς 1
  - Η καταμέτρηση όσων πέφτουν μέσα στην ακτίνα (1) του κύκλου μας δίνει το ¼ του ολοκληρώματος του εμβαδού A και επομένως του $\pi = 4A/r^2$

A = hit / total



Βλ. Σχετικό Παράδειγμα Κώδικα

Νήματα και Μέθοδοι

- Μπορώ να καλέσω μια μέθοδο ως καλούμενη συνάρτηση ενός thread;
  - Κανονικά όχι, εκτός και αν η μέθοδος είναι στατική
- Η γλώσσα όμως (C++11), μας επιτρέπει να δούμε μια μέθοδο ενός υπαρκτού στιγμιότυπου ως συνάρτηση με τη συνάρτηση μετατροπής **bind**

Μέθοδοι ως Ελεύθερες Συναρτήσεις

```
#include <iostream>
#include <functional>
struct int_holder {
    int value;
    int triple() { return value*3;}
};

int main () {
    int_holder five {5};
    std::function<int()> triple =
        std::bind (&int_holder::triple, five);
    cout << triple();
    thread th = thread (triple);
}
```

Αντικείμενο – περίβλημα (wrapper) για συναρτήσεις. Οτιδήποτε μπορεί κάποια στιγμή να κληθεί, μπορεί να μετατραπεί σε αυτόν τον τύπο.

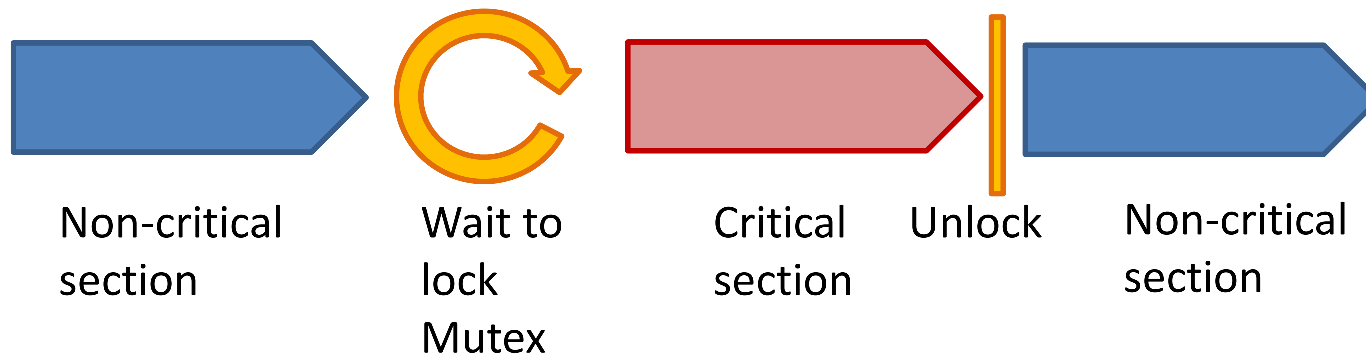
Διεύθυνση στην οποία βρίσκεται η αρχή της υλοποίησης της μεθόδου

Νήματα και Αποκλειστική Πρόσβαση σε Πόρους

- Τα νήματα έχουν πρόσβαση σε κοινούς πόρους, όπως αρχεία, μνήμη κλπ
- Αν τα νήματα κάνουν υπολογισμούς πάνω σε δεδομένα που τα άλλα νήματα τροποποιούν, θα προκύψουν λάθη
- Πολλές φορές χρειάζονται επομένως αποκλειστική πρόσβαση σε πόρους

Mutex (mutual exclusion)

- Είναι ένας μηχανισμός που επιτρέπει σε ένα νήμα να μπει σε ένα τμήμα κώδικα που εγκυμονεί προβλήματα (critical section) και να εξασφαλίσει την αποκλειστική χρήση των πόρων
- Τα υπόλοιπα νήματα (με δική μας εντολή):
  - Είτε περιμένουν να πάρουν σειρά
  - Είτε εγκαταλείπουν την προσπάθεια (return)



Χρήση Mutex

```
#include <thread>
#include <mutex>

using namespace std;

mutex m;

void ThreadFunction()
{
   // Non-critical section
   ...

   // Critical section
   m.lock();
   ...
   m.unlock();
   // Exit critical section

   // Non-critical section
   ...
}

void main()
{
   thread t1 = thread(ThreadFunction);
   thread t2 = thread(ThreadFunction);
   t1.join();
   t2.join();
}
```

Βλ. Σχετικό Παράδειγμα Κώδικα

Ασύγχρονη Εκτέλεση

- Η C++ μας δίνει τη δυνατότητα να κρύψουμε την έννοια του νήματος
- Με τη συνάρτηση std::async μπορούμε να ζητήσουμε την ασύγχρονη (και πιθανά παράλληλη) εκτέλεση μίας ή περισσοτέρων διαδικασιών
- Το αποτέλεσμα μιας ασύγχρονης εκτέλεσης μιας διαδικασίας (task) είναι ένα future
- Future: μελλοντικό αποτέλεσμα, κάτι που «θα» είναι έτοιμο σε κάποια μεταγενέστερη στιγμή

Futures

- Ένα future ενσωματώνει 2 πράγματα σε ένα περίβλημα:
  - Την τιμή ενός αποτελέσματος όταν αυτό γίνει διαθέσιμο
  - Το μηχανισμό συγχρονισμού για να περιμένει το αποτέλεσμα (ένα join() πρακτικά)

Παράδειγμα Ασύγχρονης Εκτέλεσης

```
template <typename T, typename TIter>
int parallel_sum(TIter beg, TIter end)
{
   auto len = end - beg;
   if (len < 1000)
      return std::accumulate(beg, end, 0);

   TIter mid = beg + len / 2;
   std::future<T> handle =
      std::async(std::launch::async,  parallel_sum<T, TIter>, mid, end);
   int sum = parallel_sum<T>(beg, mid);
   return sum + handle.get();
}

int main()
{
   std::vector<int> v(10000, 1);
   std::cout << "Sum = " << parallel_sum<int>(v.begin(), v.end()) << '\n';
}
```