ArtWorks

Advanced C++

Multithreading

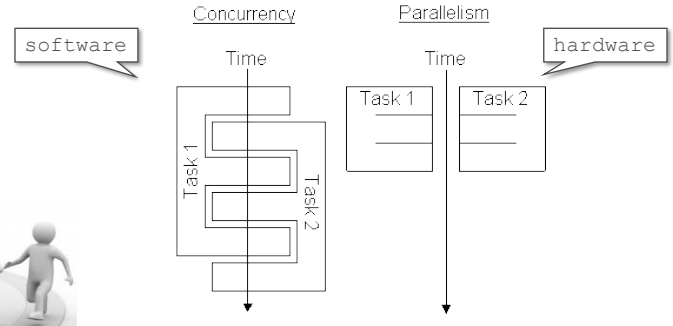
Microsoft Certified Professional Solution Developer Trainer

CompTIA. Certified Technical Trainer+

Michel André
michel@artworks.fr

Concurrence

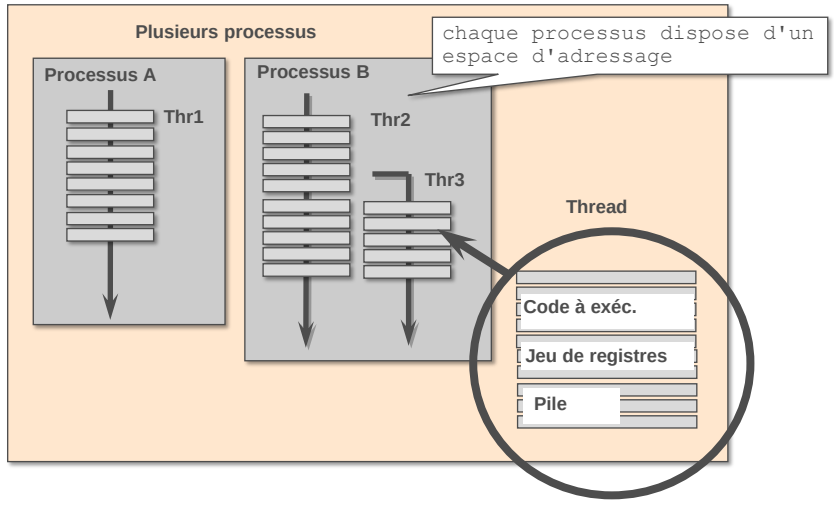
- Bloquer un thread est préjudiciable pour l'évolutivité:
  - côté client: les applications doivent être réactives
    - ne jamais bloquer une IHM
      - *toute opération de durée > 50 ms est asynchrone dans WinRT*
  - côté serveur: il faut traiter un maximum de requêtes / seconde
    - créer / détruire des threads est coûteux
    - les opérations d'I/O ne doivent pas être bloquantes



2

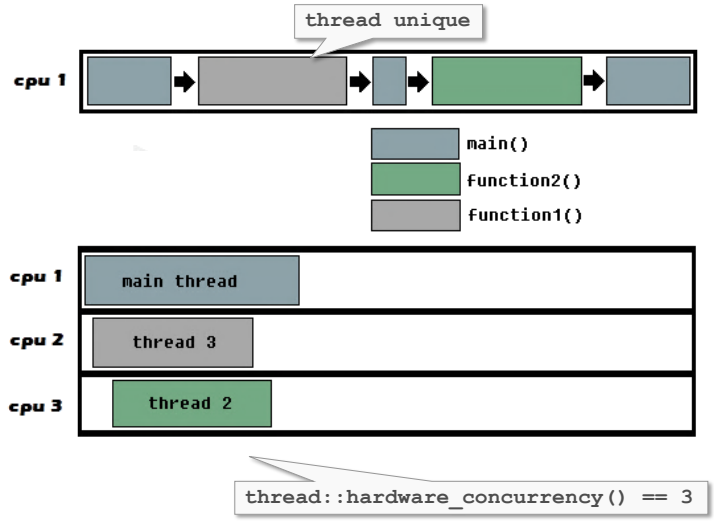
Processus et threads

Machine virtuelle du système



3

Exécution en parallèle



4

Démarrage d'un nouveau thread

```
struct task1{
  void operator() (const char* msg) const {
    this_thread::sleep_for(chrono::seconds(1));
    cout << "task1 says: " << msg << endl;
  }
};

void task2(const char* msg1, unsigned nbFois){
  while (nbFois--){
    cout << "   task2 says: " << msg1 << endl;
    this_thread::sleep_for(chrono::milliseconds(600));
  }
}
```

foncteur

```
int main() {
  thread t1{task1(), "Bonjour"};
  thread t2{task2, "Hello", 5};
  t1.join(); t2.join();
}
```

démarrage immédiat

attente de la terminaison de t1 et de t2

5

Transmission de paramètre (s)

- Un thread peut exécuter une fonction standard, un foncteur, une function<…> ou une lambda-expression
- On peut transmettre autant de paramètres que désiré
  - usage de ref / cref
- L'éventuel retour est ignoré
  - si nécessaire, transmettre une adresse/référence

```
struct functor {
  void operator()(ostream& o) const {o << "A\n";}
};
void f(ostream& o){ o << "B\n" << endl;}
```

```
thread {functor{}, ref(cout) }.join();
thread {f, ref(cout)}.join();
thread{[]{cout << "lambda\n"; }}.join();
```


6

Détachement d'un thread

- Si une instance de thread devient hors de portée (destruction par exemple), l'exécution de sa tâche s'effectue de façon détachée.
  - detach permet aussi ce détachement
    - join() devient alors inopérant
- detach (ou join) doit être appelé sur chaque thread avant la fin du programme afin d'éviter un crash
  - le destructeur de thread ne déclenche ni join, ni detach, pas de RAII

```
int main(){
  thread t {[]{
    this_thread::sleep_for(seconds(2));
    cout << "exit\n";
  }};

//  t.detach();
//  t.join();
}
```

Thread et gestion des exceptions

```
try {thread {[]{ throw exception {"ouppss"}; }}.join();}
catch (const exception& e){cout << e.what() << endl;}
```

crash

```
mutex g_mutex;
vector<exception_ptr>  g_exceptions;

void func() {
  try{throw exception{"oupss"};}
  catch (...){
    lock_guard<mutex> lock{g_mutex};
    g_exceptions.push_back(current_exception());
  }
}
```

```
int main(){
  thread t1{func}; thread t2{func};
  t1.join(); t2.join();
  for (auto& e : g_exceptions){
    try{ if (e){rethrow_exception(e);} }
    catch (const exception& e){
      cout << e.what() << endl;
    }
  }
}
```

L'espace de noms this_thread

- get_id: retourne l'identifiant natif du thread courant
  - utile pour recourir à l'API de la plateforme courante
    - ex: priorités sous Windows
- yield: indique au scheduler qu'il peut reprendre la main pour donner le contrôle à un autre thread
  - collaborations
- sleep_for: suspend l'exécution du thread courant pendant la durée transmise
- sleep_until: suspend l'exécution pendant jusqu'au time_point transmis en paramètre

Threads versus futures

- Les threads présentent quelques défauts:
  - l'appelant ne peut connaitre le retour
  - une exception levée est fatale
    - appel à terminate, sauf si une gestion lourde est mise en place
  - le détachement est à charge de l'appelant
  - terminaison brutale si oversubscription
- il y a plus simple:
  - les tâches dont la gestion incombe à l'implémenteur de la librairie standard

```
#include <future>

auto func { []{
  for (int i{}; i < 10; ++i)
    cout << i << endl;
  return "bonjour";
}};

future f {async(func)};
// …

cout << f.get() << endl;
```

Démarrage différé

- 2 politiques de lancement sont fixées par le standard:
  - launch::async
    - autorise le déclanchement asynchrone
      - *dans un autre thread*
      - *parallélisation les traitements*
  - launch::deferred
    - autorise l'évaluation "lazy"
      - *dans le même thread que l'appelant*
      - *sérialisation des traitements*

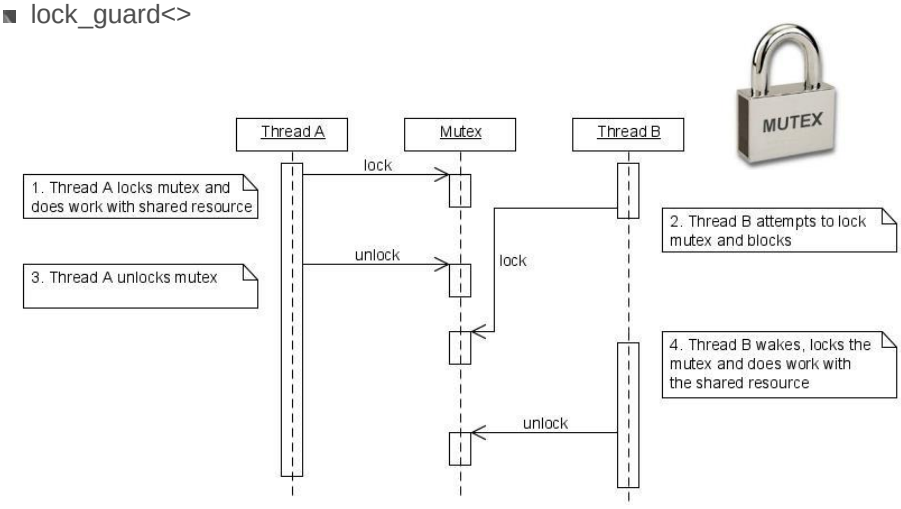
```
future f {async(launch::deferred, func)};
this_thread::sleep_for(chrono::seconds(6));

cout << f.get() << endl;
```

démarrage de func

11

Gestion de la concurrence

- mutex, recursive_mutex, timed_mutex, recursive_timed_mutex
- lock_guard<>



12

Partage de ressource

```
struct functor {
  void operator()(const char* const indent, int nbTimes) const{
    static int shared_counter{};
    static mutex shared_mutex;
    while (nbTimes-- > 0) {
      this_thread::sleep_for(chrono::milliseconds(300));
      {
        lock_guard <mutex> loc{shared_mutex};
        cout << indent << shared_counter << endl;
        ++shared_counter;
      }
    }
  }
};
```

ressource partagée

```
thread t {functor{}, "        ", 5};
thread t2 {functor{}, "", 12};
t.join();
t2.join();
```

```
        0
1
        2
3
4
        5
6
        7
        8
9
10
11
12
13
14
15
16
```


std::call_once

```
template< class Callable, class... Args >
void call_once(once_flag& flag, Callable&& f, Args&&... args);
```

- exécute le callable une seule fois dans le thread de l'appelant

```
once_flag flag;
void doIt1() {call_once(flag, []{cout << "done ... 1\n";});}
void doIt2() {call_once(flag, []{cout << "done ... 2\n"; });}
```

```
thread th1{doIt1};
thread th2{doIt2};
th1.join();
th2.join();
```

```
done ... 1
```

- si une exception est levée dans la fonction sélectionnée, elle est propagée à l'appelant
  - un autre callable est sélectionné et exécuté

La librairie atomic

```
#include <atomic>

class AtomicCounter {
  atomic<int> value;
 public:
  void increment(){++value;}
  void decrement(){--value;}
  int get() const {return value.load();}
};
```