

Curs 7

Taskuri

Future-Promise

Executie asincrona

Executori

Apeluri asincrone

Future

Promise

Istoric

- Termenul *promise* a fost propus de catre Daniel P. Friedman si David Wise in 1976;
- ~ aceeaasi perioada Peter Hibbard l-a denumit *eventual*;
- conceptul *future* a fost introdus in 1977 intr-un articol scris de catre Henry Baker si Carl Hewitt .
- *Future* si *promise* isi au originea in programarea functionala si paradigmele conexe (progr. logica)
- Scop: decuplarea unei valori (*a future*) de ceea ce o calculeaza
 - Permite calcul flexibil si paralelizabil
- Folosirea in progr. Paralela si distribuita a aparut ulterior mai intai pentru
 - reducerea latentei de comunicatie (*round trips*).apoi
 - in programele asincrone.

Promise pipelining

Barbara Liskov and Liuba Shrira in 1988

Mark S. Miller, Dean Tribble and Rob Jellinghaus 1989

```
t3 := ( x.a() ).c( y.b() )
```

Echivalent cu

```
t1 := x.a();
```

```
t2 := y.b();
```

```
t3 := t1.c(t2); //executie dupa ce t1 si t2 se termina
```

Daca x, y, t1, si t2 sunt localizate pe aceeasi masina *remote* atunci este nevoie de 3 round-trip.

Folosind futures

```
t3 := (x <- a()) <- c(y <- b())
```

Daca x, y, t1, si t2 sunt localizate pe aceeasi masina remote atunci se poate rezolva in 1 round-trip.

- O cerere trimisa si un raspuns necesar!

Evaluare

- *call by future*
 - *non-deterministic*: valoarea se va calcula candva intre momentul crearii variabilei future si momentul cand aceasta se va folosi
 - *eager evaluation*: imediat ce future a fost creata
 - *lazy evaluation*, doar atunci cand e folosita
 - Odata ce valoarea a fost atribuita nu se mai recalculeaza atunci cand se refoloseste.
- *lazy future* : calculul valorii incepe prima oara cand aceasta este ceruta (folosita)
 - in C++11
 - Politica de apel *std::launch::deferred* ->La apelul *std::async*.

Blocking vs non-blocking semantic

- Accesare asincrona->
 - De exemplu prin transmiterea unui mesaj (se asteapta pana la primirea mesajului)
- Accesare sincrona-> posibilitati
 - Accesul blocheaza threadul curent /procesul pana cand se calculeaza valoarea (eventual timeout).
 - Accesul sincronizat produce o eroare (aruncare exceptie)
 - Se poate obtine fie succes daca valoarea este deja calculata sau se transmite eroare daca nu este inca calculata -> poate introduce race conditions.
- in C++11, un thread care are nevoie de valoarea unei *future* se poate bloca pana cand se calculeaza (wait() ori get()). Eventual timeout.
 - Daca future a aparut prin apelul de tip `std::async` atunci un apel `wait` cu blocare poate produce invocare sincrona a functiei calculeaza rezultatul.

C++11

- future, promise, async, packaged_task

Future

- (1) future from a packaged_task*
- (2) future from an async()*
- (3) future from a promise*

packaged_task

- `std::packaged_task` object = wraps a callable object
 - *can be wrapped in a `std::function` object,*
 - *passed to a `std::thread` as the thread function,*
 - *passed to another function that requires a callable object,*
 - *invoked directly.*

async

- Depinde de implementare daca `std::async` porneste un nou thread sau daca taskul se va executa sincron atunci cand se cere valoarea pt future.
 - `std::launch::deferred` - se amana pana cand se apeleaza fie `wait()` fie `get()` si se va rula in threadul curent (care poate sa nu fie cel care a apelat `async`)
 - `std::launch::async` - se ruleaza in thread separat. (lazy evaluation)

async

- `async`
 - Executa o functie `f` asincron
 - posibil in alt thread si
 - returneaza un obiect `std::future` care va contine rezultatul

std::promise

- Furnizeaza un mecanism de a stoca o valoare sau o exceptie care va fi apoi obtinuta asincron via un obiect [std::future](#) care a fost creat prin obiectul [promise](#).
- Actiuni:
 - *make ready*: se stocheaza rezultatul in 'shared state'.
 - Deblocheaza threadurile care asteapta actualizarea unui obiect future asociat cu 'shared state'.
 - *release*: se elibereaza referinta la 'shared state'.
 - *abandon*: shared state = *ready* +
 - exception of type [std::future_error](#) with error code [std::future_errc::broken_promise](#)

std::promise

`promise` furnizeaza un obiect `future`.

- Se furnizeaza si un mecanism de transfer de informatie intre threaduri
 - T1-> wait()
 - T2-> set_value() => future ready.
- d.p.d.v al threadului care asteapta nu e important de unde a aparut informatia.

Example

//(1) future from a packaged_task

```
std::packaged_task<int> task( []() { return 7; } ); // wrap the function
```

```
std::future<int> f1 = task.get_future(); // get a future
```

```
std::thread(std::move(task)).detach(); // launch on a thread
```

//(2) future from an async()

```
std::future<int> f2 = std::async(std::launch::async, [](){ return 8; });
```

// (3) future from a promise

```
std::promise<int> p;
```

```
std::future<int> f3 = p.get_future();
```

```
std::thread( [&p]{ p.set_value_at_thread_exit(9); }).detach();
```

```
f1.wait();
```

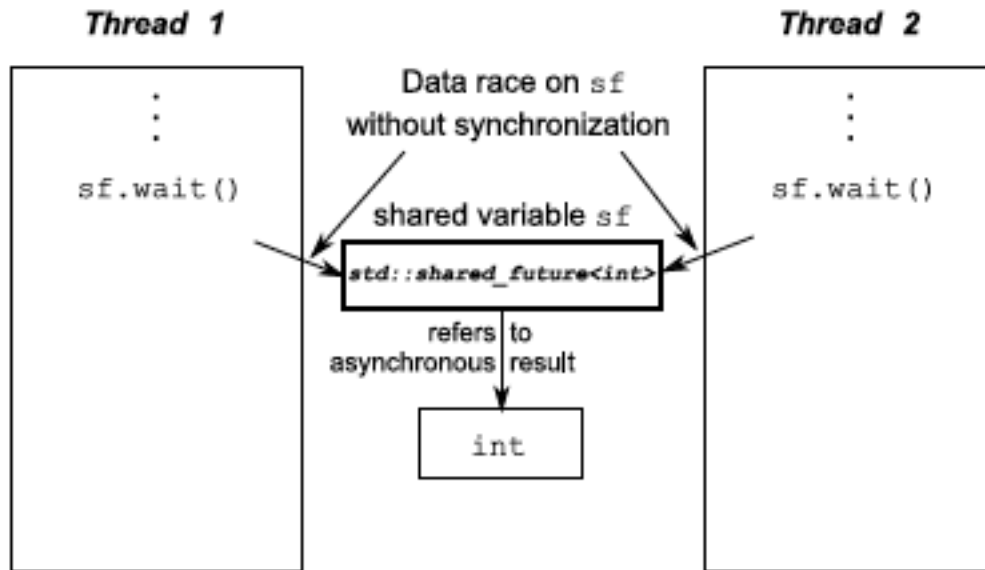
```
f2.wait();
```

```
f3.wait();
```

std::shared_future

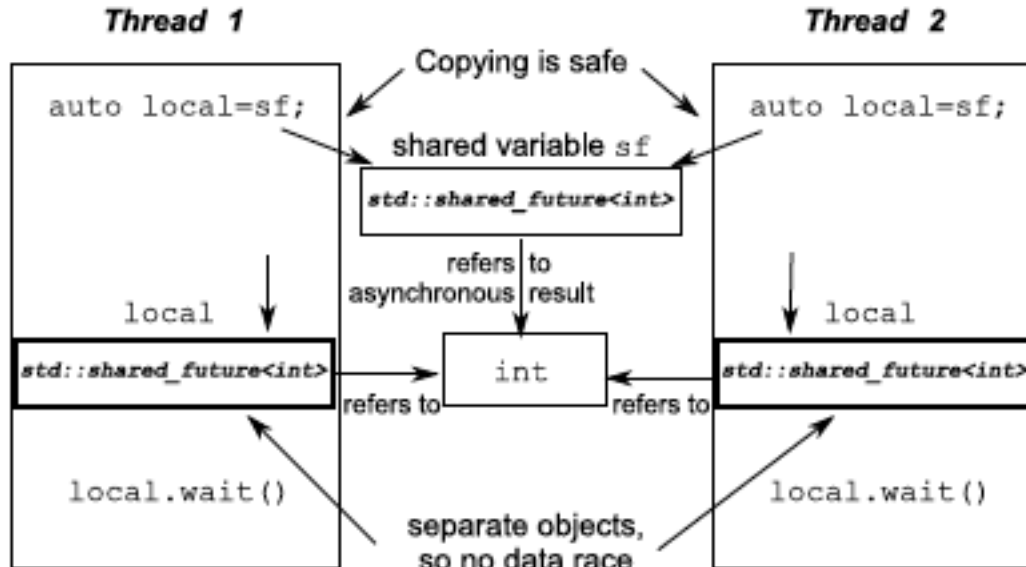
PROBLEMA: daca se acceseaza un obiect `std::future` din mai multe threaduri fara sincronizare aditionala => ***data race***.

- `std::future` modeleaza *unique ownership*
 - doar un thread poate sa preia valoarea
- `std::shared_future` permite accesarea din mai multe threaduri
- `std::future` este *moveable* (ownership can be transferred between instances)
- `std::shared_future` este *copyable* (mai multe obiecte pot referi aceeasi stare asociata).



`std::shared_future`, member functions on an individual object are still unsynchronized.

- To avoid data races when accessing a single object from multiple threads, you must protect accesses with a lock.
- The preferred way to use it would be to take a copy of the object instead and have each thread access its own copy.
- Accesses to the shared asynchronous state from multiple threads are safe if each thread accesses that state through its own `std::shared_future` object.



Java

Task

- Task = activitate independenta
 - Nu depinde de :
 - starea,
 - rezultatul, ori
 - ‘side effects’
- ale altor taskuri

=> Concurenta /Parallelism

Ref: Java Concurrency in Practice

Curs 7 - PPD

Exemplu 1

Aplicatii client server-> task = cerere client

```
class SingleThreadWebServer {  
    public static void main(String[] args) throws IOException {  
        ServerSocket socket = new ServerSocket(80);  
        while (true) {  
            Socket connection = socket.accept();  
            handleRequest(connection);  
        }  
    }  
}
```

Analiza

- procesare cerere =
 - socket I/O (read the request + write the response) -> se poate bloca
 - file I/O or make database requests-> se poate bloca
 - Procesare efectiva

Single-threaded => inefficient

- Timp mare de raspuns
- Utilizare ineficienta CPU

Exemplu 2

```
class ThreadPerTaskWebServer {  
    public static void main(String[] args) throws IOException {  
        ServerSocket socket = new ServerSocket(80);  
        while (true) {  
            final Socket connection = socket.accept();  
            Runnable task = new Runnable() {  
                public void run() {  
                    handleRequest(connection);  
                }  
            };  
            new Thread(task).start();  
        }  
    }  
}
```

Dezavantaje ale nelimitarii numarului de threaduri create

- **Thread lifecycle overhead**
 - Creare threaduri
- **Resource consumption**
 - Threadurile active consuma resursele sistemului (memorie)
 - Multe threaduri inactive blocheaza spatiu de memorie -> probleme – garbage collector
 - Multe threaduri => probleme cu CPU-> costuri de performanta
- **Stability**
 - exista o limita a nr de threaduri care se pot crea (depinde de platforma)
 - > OutOfMemoryError.

Executori

- Task = unitate logica
- Thread = un mecanism care poate executa taskurile asincron
- Interfata Executor
 - Mecanism de decuplare a submiterii unui task de executia lui
 - Suport pentru monitorizarea executiei
 - Se bazeaza pe sablonul producator-consumator

```
public interface Executor {  
    void execute(Runnable command);  
}
```

Exemplu 3

```
class TaskExecutionWebServer {  
    private static final int NTHREADS = 100;  
    private static final Executor exec= Executors.newFixedThreadPool(NTHREADS);  
    public static void main(String[] args) throws IOException {  
        ServerSocket socket = new ServerSocket(80);  
        while (true) {  
            final Socket connection = socket.accept();  
            Runnable task = new Runnable() {  
                public void run() {  
                    handleRequest(connection);  
                }  
            };  
            exec.execute(task);  
        }  
    }  
}
```


Adaptare – task per thread

```
public class ThreadPerTaskExecutor implements Executor {
    public void execute(Runnable r) {
        new Thread(r).start();
    };
}
```

```
public class WithinThreadExecutor implements Executor {
    public void execute(Runnable r) {
        r.run();
    };
}
```

Executor care executa taskurile direct in threadul apelant (synchronously).

Execution policy

“what, where, when, how” pentru executia taskurilor
= instrument de management al resurselor

- In ce thread se executa un anumit task?
- In ce ordine se aleg taskurile pentru executie (FIFO, LIFO, priority)?
- Cate taskuri se pot executa concurent?
- Cate taskuri se pot adauga in coada de executie?
- Daca sistemul este supracincarcat - *overloaded*, care task se va alege pentru anulare si cum se notifica aplicatia cre l-a trimis?
- Ce actiuni trebuie sa fie facuta inainte si dupa executia unui task

Thread pool

- Un executor care gestioneaza un set omogen de threaduri = worker threads.
- Se foloseste
 - *work queue*
pentru stocare task-uri
- Worker thread =>
 - cerere task din *work queue*,
 - Executie task
 - Intoarcere in starea de asteptare task.

Variante

- `newFixedThreadPool`.

A fixed-size thread pool creates threads as tasks are submitted, up to the maximum pool size, and then attempts to keep the pool size constant (adding new threads if a thread dies due to an unexpected Exception).

- `newCachedThreadPool`.

A cached thread pool has more flexibility to reap idle threads when the current size of the pool exceeds the demand for processing, and to add new threads when demand increases, but places no bounds on the size of the pool.

- `newSingleThreadExecutor`.

A single-threaded executor creates a single worker thread to process tasks, replacing it if it dies unexpectedly. Tasks are guaranteed to be processed sequentially according to the order imposed by the task queue (FIFO, LIFO, priority order).

- `newScheduledThreadPool`.

A fixed-size thread pool that supports delayed and periodic task execution, similar to Timer.

Runnable vs. Callable

- abstract computational tasks:
 - Runnable
 - Callable
 - Return a value
- Task
 - Start
 - [eventually] terminates
- Task Lifecycle:
 - created
 - submitted
 - started
 - completed
- Anulare (cancel)
 - Taskurile submise dar nepornite se pot anula
 - Taskurile pornite se pot anulare doar daca raspund la intrerureri
 - Taskurile terminate nu sunt influentate de 'cancel'.

Interfetele Callable si Future

```
public interface Callable<V> {  
    V call() throws Exception;  
}  
  
public interface Future<V> {  
    boolean cancel(boolean mayInterruptIfRunning);  
  
    boolean isCancelled();  
  
    boolean isDone();  
  
    V get() throws InterruptedException, ExecutionException,  
        CancellationException;  
  
    V get(long timeout, TimeUnit unit) throws InterruptedException,  
        ExecutionException, CancellationException, TimeoutException;  
}
```

Java

Future

FutureTask -> A cancellable asynchronous computation.

CompletableFuture

apel direct fara executor

```
public class Test {  
    public static class AfisareMesaj implements Callable<String>{  
        private String msg;  
        public AfisareMesaj(String m){  
            msg = m;  
        }  
        public String call(){  
            String threadName = Thread.currentThread().getName();  
            System.out.println(msg + " " + threadName);  
            return msg + " " + threadName;  
        }  
    }  
}  
  
public static void main(String a[]){  
    FutureTask<String> fs = new FutureTask<String>(new AfisareMesaj("TEST"));  
    fs.run();  
    try {  
        System.out.println(fs.get());  
    } catch (InterruptedException | ExecutionException e2) {  
        e2.printStackTrace();  
    }  
}
```


Suma numere consecutive – afisare rezultate

```
public class MyRunnable implements Runnable {  
    private final long countUntil;  
  
    MyRunnable(long countUntil) {  
        this.countUntil = countUntil;  
    }  
  
    @Override  
    public void run() {  
        long sum = 0;  
        for (long i = 1; i < countUntil; i++) {  
            sum += i;  
        }  
        System.out.println(sum);  
    }  
}
```

```
import java.util.concurrent.ExecutorService;
import java.util.concurrent.Executors;

public class Main {
    private static final int NTHREDS = 10;

    public static void main(String[] args) {
        ExecutorService executor = Executors.newFixedThreadPool(NTHREDS);
        for (int i = 0; i < 500; i++) {
            Runnable worker = new MyRunnable(100000000L + i);
            executor.execute(worker);
        }
        // This will make the executor accept no new threads
        // and finish all existing threads in the queue
        executor.shutdown();
        // Wait until all threads are finish
        executor.awaitTermination();
        System.out.println("Finished all threads");
    }
}
```

Exemplu: Futures & Callable

Suma de numere consecutive – acumulare

```
import java.util.concurrent.Callable;

public class MyCallable implements Callable<Long> {
    @Override
    public Long call() throws Exception {
        long sum = 0;
        for (long i = 0; i <= 100; i++) {
            sum += i;
        }
        return sum;
    }
}
```

```

import java.util.ArrayList;
import java.util.List;
import java.util.concurrent.Callable;
import java.util.concurrent.ExecutionException;
import java.util.concurrent.ExecutorService;
import java.util.concurrent.Executors;
import java.util.concurrent.Future;

public class CallableFutures {
    private static final int NTHREDS = 10;
    private static final int MAX = 20000;

    public static void main(String[] args) {
        ExecutorService executor =
            Executors.newFixedThreadPool(NTHREDS);
        List<Future<Long>> list =
            new ArrayList<Future<Long>>();
        for (int i = 0; i < MAX; i++) {
            Callable<Long> worker = new MyCallable();
            Future<Long> submit =
                executor.submit(worker);
            list.add(submit);
        }
    }

```

```

long sum = 0;
    System.out.println(list.size());
    // now retrieve the result
    for (Future<Long> future : list) {
        try {
            sum += future.get();
        } catch (InterruptedException e) {
            e.printStackTrace();
        } catch (ExecutionException e) {
            e.printStackTrace();
        }
    }
    System.out.println(sum);
    executor.shutdown();
}

```