

Sujet de TD n°07

séances n° 14

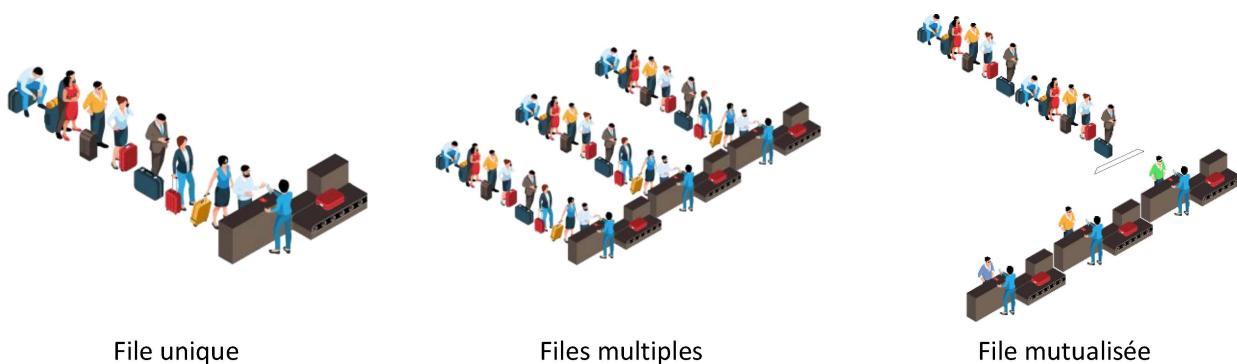
Notions abordées

- ✓ Structures de données
- ✓ Manipulation de file

File d'attente

L'objectif de ce TD est de développer un système permettant de simuler différents types de files d'attente. Ce système doit permettre de simuler différentes stratégies de distribution afin de les comparer.

L'illustration ci-dessus montre dans le contexte d'un service de gestion de bagage d'un aéroport, trois stratégies de distribution possibles : une file unique, plusieurs files indépendantes et une file mutualisée.



Pour mettre en place ce système, nous allons utiliser une structure de données reposant sur trois éléments :

- 1- Un **Service** (l'enregistrement des bagages ici) : ce composant possède une étiquette (`label`), une liste de **Worker**, une file principale (`main_queue`) et une valeur de temps (`current_frame`).
- 2- Un **Worker** (chacun des guichets) : ce composant représente une unité de traitement et possède une étiquette (`label`) et une liste de **Job** qui lui est affecté.
- 3- Un **Job** (chacune des personnes de la file) : cet élément correspond à une tache à réaliser. Celle-ci possède une étiquette (`label`), une durée totale (`duration`) et la durée restante (`remaining_time`).

Après l'initialisation du service, nous allons procéder en trois temps :

- 1- Créer une liste de **Worker** et les ajouter au **Service** ;
- 2- Créer une liste de **Job** dans la file principale du **Service** ;
- 3- Paramétrier et lancer la simulation.

Dans les grandes lignes, le programme principal de l'application sera le suivant :

```
#include <iostream>
#include <sstream>
#include <random>

#include "tm.hpp"

int main() {
    TaskManager::Service service = TaskManager::createService("Passage en caisse");

    // Setup
    const std::size_t nb_workers = 5;
    const std::size_t nb_jobs = 50;
    TaskManager::Distribution distribution = TaskManager::Distribution::RANDOM;

    // Generate workers
    for(std::size_t n=0; n<nb_workers; ++n) {
        std::stringstream ss;
        ss << "caisse " << n+1;

        TaskManager::Worker worker = TaskManager::createWorker(ss.str());
        TaskManager::addWorker(service, worker);
    }

    // Generate jobs
    std::uniform_int_distribution<std::size_t> dist_duration(1, 5);
    for(std::size_t n=0; n<nb_jobs; ++n) {
        std::stringstream ss;
        ss << "client " << n+1;
        const std::size_t job_duration = dist_duration(TaskManager::random_engine);

        TaskManager::Job job = TaskManager::createJob(ss.str(), job_duration);
        TaskManager::addJob(service, job);
    }

    // Run simulation
    TaskManager::displayService(service);
    TaskManager::run(service, distribution);

    return 0;
}
```

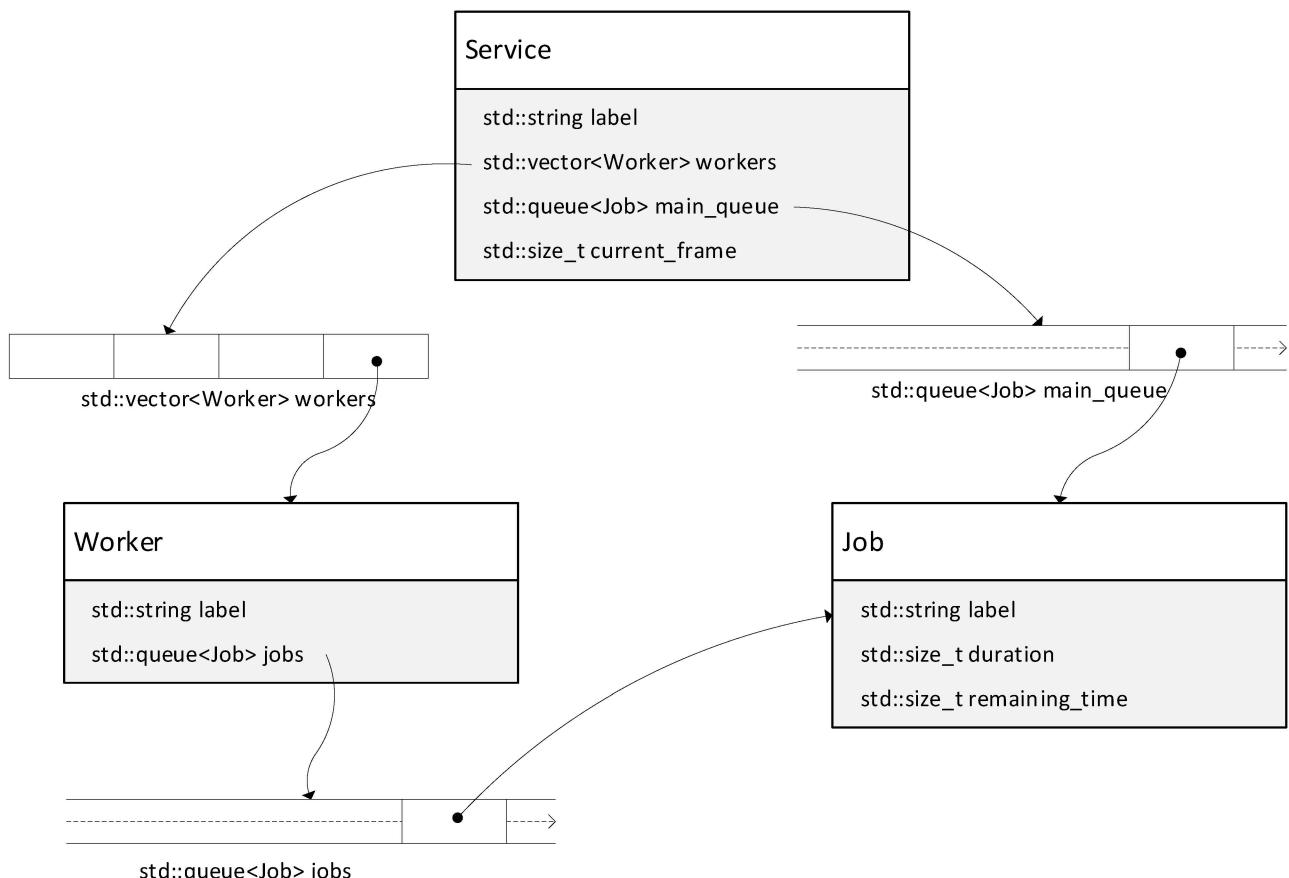
On retrouve ici chacune des étapes citées précédemment. Tachez de les identifier.

Lorsque l'on exécute ce programme, après 17 itérations, on obtient ce qui suit sur la sortie standard. Après avoir été dispatché dans les **Worker**, les **Job** sont consommés au fur et à mesure jusqu'à épuisement complet. L'objectif est d'améliorer la stratégie utilisée (cf. Annexe, fonction run) afin de réduire autant que possible le nombre de frames nécessaire sans pour autant augmenter le nombre de **worker**.

```
Services:::Passage en caisse
main queue: empty

frame:17
caisse 1: client 15[75%] client 24[0%] client 35[0%] client 37[0%] client 46[0%]
caisse 2: client 33[40%] client 39[0%] client 44[0%] client 48[0%]
caisse 3: client 41[50%] client 42[0%] client 45[0%]
caisse 4: client 25[0%] client 29[0%] client 30[0%] client 32[0%] client 36[0%]
client 38[0%] client 40[0%] client 47[0%] client 49[0%]
caisse 5: client 31[50%] client 43[0%] client 50[0%]
```

- 1) Ecrire la déclaration de la structure de données correspondant à la description donnée précédemment et représentée par le schéma ci-dessous. Ecrivez également la définition de la fonction **createService** utilisée dans le programme principal et qui permet de créer un nouveau service.



- 2) En utilisant le contenu du fichier d'entête qui vous est donné en annexe et la structure de donnée de la question 1, écrivez la définition de la fonction **addWorker** appelée dans le programme principal qui permet d'ajouter un nouveau **Worker** dans un service existant.
- 3) Ecrivez la définition de la fonction **addJob** qui permet d'ajouter un nouveau **Job** dans la file principale d'un service existant et qui est utilisé dans le programme principal. Attention, pour écrire cette fonction, il peut être utile d'en écrire d'autres.
- 4) Ecrivez la définition de la fonction **dispatchNextJob** (cf. annexe) qui permet d'affecter la prochaine tâche de la file principale d'un service à un **Worker**.

ANNEXE

tm.hpp

```
#ifndef __TM_HPP__
#define __TM_HPP__

#include <vector>
#include <queue>
#include <string>
#include <random>

namespace TaskManager {
    enum Distribution { RANDOM, SEQUENTIAL };

    // Data structures
    // question 1

    // Tools
    extern std::mt19937 random_engine;

    // Interfaces
    // Job
    void displayJob(const Job& job);
    void displayJobAchievement(const Job& job);

    // Worker
    void displayWorker(const Worker& worker);

    // Services
    Service createService(const std::string& label);
    Worker createWorker(const std::string& label);
    void addWorker(Service& service, const Worker& worker);
    Job createJob(const std::string& label, std::size_t duration);
    void addJob(Service& service, const Job& job);
    void displayService(const Service& service);
    bool idle(const Service& service);
    void run(Service& service, const Distribution& dist = Distribution::RANDOM);
} // namespace taskmanager

#endif
```

tm.cpp (part 1/4)

```
#include "tm.hpp"
#include <vector>
#include <queue>
#include <string>
#include <stdexcept>
#include <iostream>
#include <chrono>
#include <thread>

namespace TaskManager {
    // --- Tools -----
    // -----
    std::mt19937 random_engine(std::random_device {}());
```

tm.cpp (part 2/4)

```

void clear() {
    std::cout<< "\x1B[2J\x1B[1;1H";
}

// --- Job -----
// -----
// Create a new job
Job createJob(const std::string& label, std::size_t duration) {
    // question 3
}

// Display job informations
void displayJob(const Job& job) {
    std::cout<<"\x1B[38;5;4m"<<job.label<<"\x1B[0m[\x1B[38;5;204m"<<job.duration<<"\x1B
[0m] ";
}

// Display job informations and achievement
void displayJobAchievement(const Job& job) {
    std::cout<<"\x1B[38;5;4m"<<job.label<<"\x1B[0m[\x1B[38;5;204m"<<100.0*(job.duration
- job.remaining_time)/(double)job.duration<<%"\x1B[0m] ";
}

// --- Worker -----
// -----
// create a new worker
Worker createWorker(const std::string& label) {
    // question 2
}

// Display worker information
void displayWorker(const Worker& worker) {
    std::cout << "\x1B[38;5;47m" << worker.label << "\x1B[0m (working "
        << worker.working_frames<<"/ idle "
        << worker.idle_frames<<") : ";

    if (worker.jobs.empty()) {
        std::cout << std::endl;
        return;
    }

    std::queue<Job> jobs_copy = worker.jobs;
    while (!jobs_copy.empty()) {
        displayJobAchievement(jobs_copy.front());
        jobs_copy.pop();
    }
    std::cout << std::endl;
}

// Add a job to a worker
void addJob(Worker& worker, const Job& job) {
    // question 4
}

```

tm.cpp (part 3/4)

```

// --- Service -----
// Create a new services with a given label
Service createService(const std::string& label) {
    // question 1
}

// Add a worker to a given services
void addWorker(Service& service, const Worker& worker) {
    // question 2
}

// Add a job to services's main queue
void addJob(Service& service, const Job& job) {
    // question 3
}

// Dispatch next job to a given worker id
void dispatchNextJob(Service& service, std::size_t worker_id) {
    // question 4
}

// Display service informations
void displayService(const Service& service) {
    clear();
    std::cout << "Services::" << service.label << std::endl;
    std::cout << "main queue: ";
    if (service.main_queue.empty()) {
        std::cout << "empty" << std::endl;
    } else {
        std::queue<Job> jobs_copy = service.main_queue;
        while (!jobs_copy.empty()) {
            displayJob(jobs_copy.front());
            jobs_copy.pop();
        }
    }
    std::cout << std::endl;
    std::cout << "frame:" << service.current_frame << std::endl;
    for (const Worker& worker : service.workers) {
        displayWorker(worker);
    }

    std::this_thread::sleep_for(std::chrono::milliseconds(125));
    std::cout << std::endl;
}

// true if all workers are idle
bool idle(const Service& service) {
    bool is_empty = true;
    for (std::vector<Worker>::const_iterator it = service.workers.begin(); it != service.workers.end() && is_empty; ++it) {
        is_empty &= (*it).jobs.empty();
    }
    return is_empty;
}

```

tm.cpp (part 4/4)

```

// Dispatch jobs to worker using random distribution
void dispatchAllJobsWithRandomDistribution(Service& service) {
    std::uniform_int_distribution<std::size_t> dist_workers(0, service.workers.size() - 1);
    while (!service.main_queue.empty()) {
        const std::size_t worker_id = dist_workers(random_engine);
        TaskManager::dispatchNextJob(service, worker_id);
    }
}

// Dispatch jobs to worker using sequential distribution
void dispatchAllJobsWithSequentialDistribution(Service& service) {
    std::size_t worker_id = 0;
    while (!service.main_queue.empty()) {
        TaskManager::dispatchNextJob(service, worker_id);
        worker_id = (worker_id + 1) % service.workers.size();
    }
}

// Start simulation
void run(Service& service, const Distribution& dist) {
    std::this_thread::sleep_for(std::chrono::milliseconds(3000));

    // Select distribution
    switch (dist) {
        case Distribution::SEQUENTIAL:
            dispatchAllJobsWithSequentialDistribution(service);
            break;
        case Distribution::RANDOM:
            dispatchAllJobsWithRandomDistribution(service);
        default:
            break;
    }

    // run simulation
    while (!idle(service)) {
        for(Worker& worker : service.workers) {
            if (!worker.jobs.empty()) {
                worker.working_frames++;
                if (worker.jobs.front().remaining_time <= 1)
                    worker.jobs.pop();
                else
                    worker.jobs.front().remaining_time--;
            } else {
                worker.idle_frames++;
            }
        }
        service.current_frame++;
        displayService(service);
    }
}
} // namespace taskmanager

```