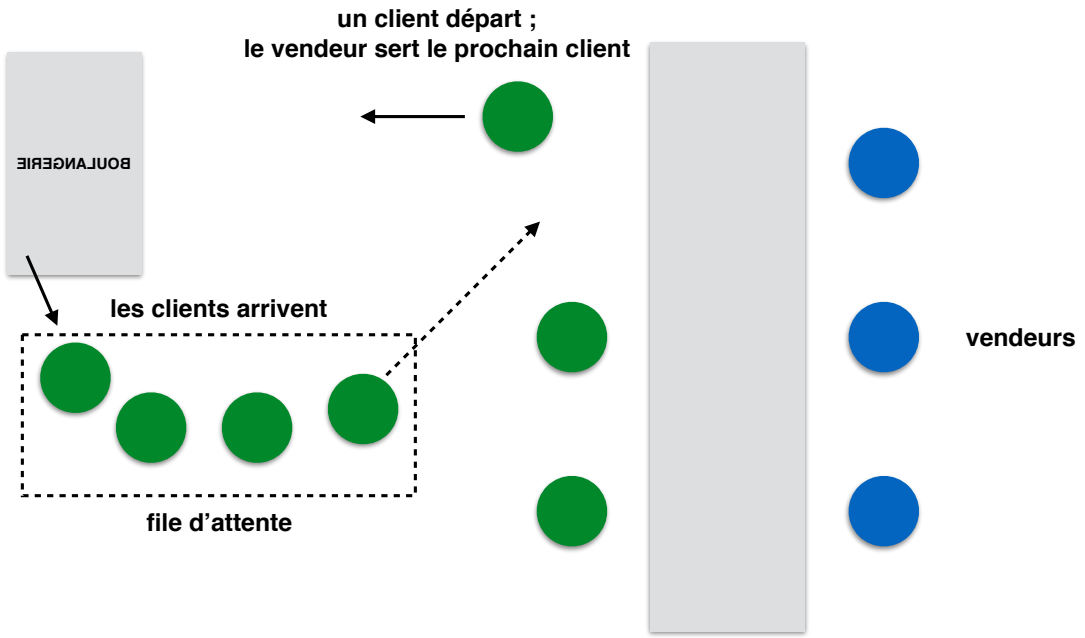
TP 5 et 6 – Simulation discrète d'une boulangerie

Objectif: Le but de ce TP est d'implémenter une simulation d'une file d'attente dans une boulangerie. Ce TP va vous occuper pendant deux séances. La première séance, vous disposez quelques structures de données déjà implémentées, une file d'attente et une file de priorité, pour implémenter la simulation. La deuxième séance, vous allez réimplémenter ces deux structures en utilisant des listes chaînées. Il n'y a qu'un seul rendu à déposer, après la deuxième séance (voir "4 Rendu" ci-dessous).

1 Description du système



Les clients (verts dans la figure) arrivent avec une certaine fréquence, un par minute, par exemple. S'il y a un vendeur (bleus dans la figure) qui est libre, alors le client est immédiatement servi. Sinon, le client se met à la fin de la file d'attente. Après un certain temps un vendeur termine avec un client et le client sort de la simulation (il sort de la boulangerie si vous voulez). Le vendeur inspecte la file d'attente. Si elle est vide, le vendeur devient libre. Sinon, le vendeur sert le client en tête de la file d'attente. À la fin de la journée, on affiche quelques statistiques : combien de clients ont été servis ? Quel était le *temps moyen de service* (le temps moyen qu'un client passe à l'intérieur de la boulangerie) ? Les paramètres de la simulation sont (1) le temps entre l'arrivée des clients, (2) le temps de service d'un client et (3) le nombre de vendeurs.

Le type de simulation qu'on va implémenter s'appelle une *simulation à événements discrets*. Cela veut dire qu'il n'y a pas d'horloge dans le système qui avance en attendant une action. À la place, l'heure saute chaque fois qu'un *événement* se passe. Un événement de notre système peut être soit l'arrivée soit le départ d'un client.

Quand un événement est créé, par exemple : "à l'heure 135 secondes après le début de la simulation, un client va arriver", alors cet événement est placé dans une *file de priorité d'événements* (*event queue* en anglais). Le temps donné à un événement s'appelle l'heure de début (event time) de l'événement. Le cœur de la simulation est la boucle principale : tant que la file de priorité d'événements n'est pas vide, alors on extrait l'événement avec le temps de début le plus proche et on le traite. Par exemple, s'il y a deux événements dans la file de priorité ayant l'heure de début 135 secondes et 87 secondes, alors la boucle va d'abord extraire l'événement avec un temps de début de 87 secondes et le traiter.

Le traitement d'un événement commence en avançant l'heure globale jusqu'à l'heure de début de l'événement. Le traitement peut créer d'autres événements qui sont insérés dans la file de priorité d'événements. Par exemple, s'il y a un événement qui annonce l'arrivée d'un client à l'heure 135 et si à ce moment il y a un vendeur qui est libre, alors le traitement de l'événement crée un nouvel événement de type : "le vendeur va terminer la service du client à l'heure 285".

La simulation d'une boulangerie comporte deux types d'événements : un événement d'arrivée (d'un client) et un événement de départ (d'un client). Ci-dessous, le traitement de ces deux types d'événements est décrit. Dans les exercices, vous trouvez plus de détails pour l'implémentation.

Traitement d'un événement d'arrivée :

1. Créer un nouveau client.
2. S'il y a un vendeur v libre, alors
   — le vendeur v commence à servir le client ;
   — un événement de départ pour le client à l'heure + X secondes est placé dans la file de priorité d'événements.
3. Sinon,
   — le client se place dans la file d'attente.
4. Un événement d'arrivée à l'heure + Y secondes est placé dans la file de priorité d'événements.

Traitement d'un événement de départ :

1. Le vendeur v a fini de servir son client.
2. Supprimer le client du système.
3. Si la file d'attente est vide, alors
   — le vendeur v devient libre.
4. Sinon,
   — le client à la tête de la file d'attente est extrait et le vendeur v commence à le servir ;
   — un événement de départ pour le client à l'heure + X secondes est placé dans la file de priorité d'événements.

2 File d'attente à la boulangerie

▸ Exercice 1 : Votre premier client

Commencer par récupérer l'archive `bakery.zip` et extraire les fichiers dans votre répertoire de travail. L'archive contient les fichiers suivants :

- `customer.h/.c` : la structure qui modélise un client ;
- `event.h/.c` : la structure qui modélise un événement ;
- `queue.h/.o` : la structure de données qui implémente la file d'attente des clients ;
- `prioqueue.h/.o` : la structure de données qui implémente la file de priorité des événements ;
- `bakery.c` : le fichier principal qui contient la fonction `main` ;
- `Makefile` : le makefile.

Familiarisez-vous avec le contenu de ces fichiers. Pendant cette séance vous allez uniquement rédiger le fichier `bakery.c`. Si vous avez des questions, n'hésitez pas à les poser à votre chargé de TP.

Dans le fichier `bakery.c`, la constante `N_VENDORS` indique le nombre de vendeurs dans la simulation. Trois variables globales sont déclarées qui représentent les composantes principales de la simulation :

- `prioqueue *event_queue` – un pointeur sur la file de priorité d'événements ;
- `queue *customer_queue` – un pointeur sur la file d'attente des clients ;
- `customer *vendor[N_VENDORS]` – un tableau qui pour chaque vendeur contient un pointeur sur le client que le vendeur est en train de servir, ou `NULL` si le vendeur est libre.

1. Au début de votre fonction `main`, ajoutez des instructions pour initialiser les trois variables globales décrites ci-dessus avec une file de priorité vide, une file d'attente vide et des pointeurs `NULL` pour chaque vendeur.
2. Ajoutez les instructions nécessaires pour créer un client (avec un temp d'arrivée de 10 secondes, par exemple) et ajoutez-le à la file d'attente. Ensuite, extraire le client à la tête de la file d'attente et afficher son temps d'arrivée. Quand ca fonctionne, vous pouvez supprimer ces lignes.

▸ Exercice 2 : La boucle d'événements

Vous allez maintenant ajouter la boucle d'événements.

1. La boucle d'événement se comporte comme suit :
   - Tant que la file de priorité d'événements n'est pas vide, alors extraire le prochain événement de la file et traitez-le.
   - Le traitement d'un événement dépend de son type. Appeller la fonction `process_arrival` ou la fonction `process_departure` d'après le type de l'événement. (Pour l'instant, ces fonctions ne font que libérer la mémoire associée à l'événement.)

   Écrire la boucle d'événements dans votre fonction `main` ou dans une fonction séparée qui sera appelée par la fonction `main`.
2. Ajoutez une instruction à la boucle d'événement qui affiche le temps de début de l'événement juste avant de le traiter.
3. Dans votre fonction `main`, créez un événement d'arrivée avant d'entrer dans la boucle d'événements. Lancez votre programme pour vérifier que la boucle d'événements traite correctement cet événement. Après avoir affiché le temps de début de l'événement et appelé la fonction `process_arrival`, votre boucle d'événements devrait terminer (comme la file de priorité d'événements est devenue vide).

▸ **Exercice 3 : L'arrivée d'un client**

Dans cet exercice, vous allez implémenter le traitement d'un événement d'arrivée. À l'occurrence d'un tel événement, le programme crée un nouveau client et soit l'affecte à un vendeur libre, soit l'insère à la fin de la file d'attente.

1. Ajouter une variable globale `int current_time`, le temps courant, qui représente le nombre de secondes passé depuis le début de la simulation. Au début du programme, initialiser cette variable à 0. Dans la boucle d'événements, lui affecter le temps de début de l'événement extrait de la file de priorité.
2. Écrire une fonction `void add_customer(customer *c)`. Cette fonction parcourt les vendeurs en cherchant un qui est libre. Le premier vendeur libre trouvé va servir le client `c`. Cela se fait en affectant `vendor[vid]=c` où `vid` est le numéro du vendeur. S'il n'y a pas de vendeur libre, alors le client sera ajouté à la file d'attente. Un client est donc toujours dans *exactement un seul endroit du système* : soit il est dans la file d'attente, soit il est en train d'être servi par un vendeur.
3. Réécrire la fonction `process_arrival` pour effectuer le suivant :
   1) Créer un nouveau client avec un temps d'arrivée = temps courant du système ;
   2) Appeler la fonction `add_customer` avec le client ;
   3) Libérer la mémoire associée à l'événement ;
   4) Créer un nouvel événement d'arrivée avec un temps de début = temps courant + 60
   5) Insérer le nouvel événement dans la file de priorité d'événements.
4. Créez une fonction pour visualiser l'état courant du système et appelez-le à la fin de la boucle d'événement. Exemple d'affichage :

```
 60 | X__ |
120 | XX_ |
180 | XXX |
240 | XXX | X
300 | XXX | XX
360 | XXX | XXX
```

   La première colonne montre le temps courant. La deuxième colonne montre les vendeurs, `X` signifie que le vendeur est en train de servir un client, `_` signifie que le vendeur est libre. La troisième colonne montre la file d'attente. Chaque `X` représente un client en attente.

   Dans la visualisation ci-dessus, on voit que :
   — Le premier client est arrivé à l'heure 60. Il est immédiatement servi par un vendeur.
   — Les deuxième et troisième clients arrivent respectivement à l'heure 120 et 180 et sont également immédiatement servis.
   — Le quatrième client qui arrive à l'heure 240 est placé dans la file d'attente.

   Comme les clients ne sortent pour l'instant jamais du système, la file d'attente va continuer à augmenter indéfiniment.
5. Définir une constante `CLOSING_TIME` qui représente le temps de fermeture de la boulangerie. Dans la boucle d'événement, ajouter une condition d'arrêt : si le temps courant dépasse `CLOSING_TIME`, alors la boucle termine.

▸ Exercice 4 : Le départ d'un client

Dans cet exercice, vous allez implémenter le traitement d'un événement de départ. À l'occurrence d'un tel événement, le programme supprime le client concerné du système. De plus, le vendeur qui est désormais devenu libre va chercher un autre client dans la file d'attente (s'il y en a un) et commencer à le servir.

1. Écrire la fonction `void remove_customer(int vid)`. Cette fonction libère la mémoire associée au client de l'adresse `vendor[vid]` et met `vendor[vid]` à `NULL`. Si la file d'attente n'est pas vide, alors
   — la fonction extrait le premier client `c` de la file et met `vendor[vid]=c`. Ensuite, elle crée un nouvel événement de départ avec un temps de début = le temps courant + 150. Cet événement représente le temps auquel le vendeur terminera de servir le client `c`. Finalement, la fonction insère l'événement dans la file de priorité d'événements.
2. Réécrire la fonction `process_departure` pour qu'elle appelle la fonction `remove_customer` avec le *vid* du vendeur de l'événement avant de libérer la mémoire associée à l'événement.
3. Dans la fonction `add_customer`, après avoir affecté un client à un vendeur, créer un nouvel événement de départ de la même manière qu'en (1).
4. Votre programme devrait maintenant produire un affichage qui ressemble au suivant.

```
  60 | X__ |
 120 | XX_ |
 180 | XXX |
 210 | _XX |
 240 | XXX |
 270 | X_X |
 300 | XXX |
 330 | XX_ |
 360 | XXX |
```

   Remarquer qu'aux temps 210, 270 et 330, des clients sont *supprimés* de la simulation.

▸ Exercice 5 : Arrivée et départ aléatoire

Les clients arrivent pour l'instant avec une ponctualité surhumaine et sont tous servis dans un temps précis de 150 secondes. Vous allez maintenant remplacer cette exactitude par un processus aléatoire.

1. Ajouter la fonction suivante au fichier `bakery.c`. Assurez-vous que les librairies `<math.h>` (pour la fonction `log`) et `<stdlib.h>` (pour la fonction `rand`) sont incluses et changez la ligne `LDFLAGS=` à `LDFLAGS=-lm` dans le makefile.

   /ens/IR/IR1/2016-2017/Algo/src/tp05/normal_delay.c

```
double normal_delay(double mean) {
    return -mean*log(1-((double)rand()/RAND_MAX));
}
```

2. L'arrivée des clients sera modélisée par un *processus de Poisson*[1]. Cela veut dire que les clients arrivent indépendamment aux moments aléatoires avec une intensité moyenne fixe, par exemple un client toutes les 60 secondes.

1. https://fr.wikipedia.org/wiki/Processus_de_Poisson

Pour l'implémenter, définir une constante `#define ARRIVAL_RATE (1.0/60)` et remplacer la constante 60 utilisée dans la fonction `process_arrival` par l'appel : `normal_delay(1.0/ARRIVAL_RATE)`. Le temps moyen entre l'arrivée des clients reste à 60 secondes. Vous allez voir ce processus et d'autres dans le cours "Probabilités, statistiques" au troisième semestre.

3. Le temps pour servir un client est pour l'instant exactement 150 secondes. On va le modéliser par une variable aléatoire suivant une *loi normale* de moyenne `MEAN_SERVICE_TIME`[2].

   Définir une constante `#define MEAN_SERVICE_TIME 150` et remplacer 150 (deux occurrences) par l'appel : `normal_delay(MEAN_SERVICE_TIME)`.

4. Votre programme devrait maintenant produire un affichage qui ressemble au suivant.

```
 44 | X__ |
 62 | XX_ |
 75 | XXX |
118 | XXX | X
132 | XXX |
163 | XXX | X
169 | XXX | XX
176 | XXX | X
243 | XXX |
270 | XXX | X
286 | XXX | XX
```

5. Essayer des valeurs différentes pour les paramètres `N_VENDORS`, `ARRIVAL_RATE` et `MEAN_SERVICE_TIME` et observer les effets. Essayer de trouver une condition sur ces trois paramètres qui assure que la longueur de la queue reste raisonnable.

▸ Exercice 6 : Statistiques et nettoyage

1. Ajouter un compter pour le nombre de clients servis et un compteur pour le temps total que les clients ont passé dans le système. Le temps total est égal à la somme pour chaque client de son temps de départ moins son temps d'arrivée. À la fin de la simulation, afficher le nombre de clients servis ainsi que le temps moyen de service, le temps total que les clients ont passé dans le système divisé par le nombre de clients servis. Le temps moyen de service est le temps moyen pour un client entre son arrivée et son départ.
2. Votre programme termine probablement sans libérer toute la mémoire allouée. Les files peuvent contenir des clients et des événements qui ne sont pas utilisés après la fermeture de la boulangerie et les vendeurs peuvent aussi rester avec des clients. Les files elles-mêmes doivent également être libérées à la fin de l'exécution. Réfléchir, pour chaque client et chaque événement, où ils sont créés et où ils sont libérés.
3. La boulangerie ferme pour l'instant assez brusquement. Si vous voulez, vous pouvez changer ce comportement. Par exemple, les vendeurs peuvent continuer à servir les clients qui sont à l'intérieur de la boulangerie à l'heure de fermeture mais pas autoriser des nouveaux clients à se mettre dans la file d'attente.
4. Essayer des valeurs différentes pour les paramètres `N_VENDORS`, `ARRIVAL_RATE` et `MEAN_SERVICE_TIME` et observer les effets sur le temps moyen de service.

2. https://fr.wikipedia.org/wiki/Loi_normale

▸ Exercice 7 : Théorie des files d'attente (FACULTATIF)

Votre programme implémente un simple système qui s'appelle une file M/M/c[3] dans la *théorie des files d'attente*[4].

1. Utiliser la fonction suivante pour calculer la valeur théorique du temps moyen de service pour un tel système[5] :

/ens/IR/IR1/2016-2017/Algo/src/tp05/response_time.c

```
double response_time(double arrival_rate,
                     double mean_service_time,
                     int n_vendors) {
    double C;
    int c = n_vendors;
    double lambda = arrival_rate;
    double mu = 1.0/mean_service_time;
    double lm = lambda/mu;
    double rho = lm/c;
    double sum = 0.0;
    int j,k;
    for (k = 0; k < c; k++) {
        double term = 1.0;
        for (j = k+1; j <= c; j++) {
            term *= j/(c*rho);
        }
        sum += term;
    }
    C = 1.0/(1+(1-rho)*sum);
    return C/(c*mu-lambda)+1/mu;
}
```

2. La valeur obtenue dans votre simulation, correspond-elle à la moyenne théorique ? Sinon, pouvez-vous imaginer des raisons à cela ? Est-il possible d'ajuster votre programme pour avoir une meilleure correspondance ?
3. Modifier votre système pour limiter la capacité de la file d'attente. C'est-à-dire, si un client arrive à une file d'attente avec K client ou plus dedans, alors le client n'entre pas dans la file d'attente mais cherche une autre boulangerie.

3. https://en.wikipedia.org/wiki/M/M/c_queue
4. https://fr.wikipedia.org/wiki/Th%C3%A9orie_des_files_d'attente
5. https://en.wikipedia.org/wiki/M/M/c_queue#Response_time

3 Listes chaînées

▸ **Exercice 8 : `queue.c`**

Dans cet exercice, vous allez implémenter l'interface dans le fichier `queue.h` en utilisant une liste simplement chaînée. Pendant l'implémentation, vous pouvez faire des tests dans une fonction `main` que vous placez dans le fichier `queue.c` et compilez avec la commande `gcc queue.c`.

1. Télécharger le fichier `queue.c` :

/ens/IR/IR1/2016-2017/Algo/src/tp05/queue.c

```
#include "queue.h"
#include <stdlib.h>

typedef struct _link {
    customer*       c;
    struct _link*   next;
} link;

struct _queue {
    link*   first;
    link*   last;
    int     size;
};

queue *create_q() {
    queue *q = (queue*)malloc(sizeof(queue));
    q->first = NULL;
    q->last = NULL;
    q->size = 0;
    return q;
}

void free_q(queue *q) {
    free(q);
}

int size_q(queue *q) {
    return q->size;
}
```

2. Écrire la fonction `void enqueue_q(queue *q, customer *c)` dans le fichier `queue.c`. La fonction alloue la mémoire d'une cellule (type `link`) et y ajoute le client `c`. Ensuite, la fonction insère la nouvelle cellule à la fin de la liste d'attente.
3. Écrire la fonction `customer *dequeue_q(queue *q)` dans le fichier `queue.c`. La fonction extrait la cellule en tête de la liste, libère la mémoire associée et renvoie le client qui était stocké dans la cellule.

4. Vérifier que la simulation du fichier `baker.c` fonctionne en tapant `make` et `./baker`
   Cela remplace le fichier `queue.o` par votre propre file d'attente.

▸ **Exercice 9 : `prioqueue.c`**

Dans cette exercice, vous allez implémenter l'interface dans le fichier `prioqueue.h` en utilisant une liste simplement chaînée triée sur les temps de début des événements.

1. Créer le fichier `prioqueue.c` et ajouter les définitions :

```
typedef struct _link {
    event*          e;
    struct _link*   next;
} link;

struct _prioqueue {
    link*   first;
    int     size;
};
```

2. Écrire les fonctions suivantes :

```
prioqueue *create_pq()
void free_pq(prioqueue *q)
int size_pq(prioqueue *q)
```

3. Écrire la fonction `void insert_pq(prioqueue *q, event *e)` qui alloue la mémoire d'une cellule (type `link`) et y ajoute l'événement `e`. Ensuite, la fonction insère la nouvelle cellule à sa place dans la liste afin de garder la liste triée sur les temps de début des événements.
4. Écrire la fonction `event *remove_min_pq(prioqueue *q)` qui extrait la cellule en tête de la liste (qui a le plus petit temps de début), libère la mémoire associée et renvoie l'événement qui était stocké dans la cellule.
5. Vérifier que la simulation du fichier `baker.c` fonctionne en tapant `make` et `./baker`
   Cela remplace le fichier `queue.o` par votre propre file de priorité.

4 Rendu

Le rendu se fait sur le site https://elearning.u-pem.fr/mod/assign/view.php?id=41199 avant la date limite indiquée.

- Vous rendez une archive `.zip` avec tous vos fichiers, le `Makefile` inclus.
- Vous rendez également un fichier `reponses.txt` avec des réponses courtes aux questions posées dans les exercices 5.5 (la condition) et 7 (si vous l'avez fait).

Un rendu minimal consiste en une simulation qui fonctionne avec des arrivées et départs aléatoires (jusqu'à l'exercice 5 inclus) ainsi qu'au moins une des deux structures de la deuxième partie implémentée : `queue` ou `prioqueue` (exercices 8 et 9).

Un rendu minimal sera noté par au plus 12.

- Tout retard sera sanctionné.
- Un rendu avec une archive `.rar` sera sanctionné.
- Un TP non rendu sera noté 0.
- Un TP qui ne compile pas en tapant `make` sera noté 0.
- Un TP qui sort 10 avertissements à la compilation et ensuite termine avec une erreur de segmentation sera noté 0 – testez vos programmes sur plusieurs machines.