Benjamin Guillet - Ludovic Henry

Projet de LO41

Simulation de traffic ferroviaire

Rapport de projet

Automne 2011

Table des matières

1 Présentation du sujet

1.1 Généralités

Ce semestre dans le cadre de l'unité de valeur LO41 (architecture et utilisation des systèmes d'exploitation), nous sommes amenés à modéliser un système de gestion du traffic ferroviaire.

Le réseau ferroviaire est composé d'une gare, d'un ensemble de voies uni ou bidirectionnelles, de voies de stockage, d'un tunnel unidirectionnel etc. Le traffic, lui, est composé de différents trains (TGV, grandes lignes ou marchandises) qui peuvent emprunter certaines voies (mais pas toutes), possèdent une direction et une priorité plus ou moins importante (le TGV est le train le plus prioritaire, vient ensuite le train grande ligne, et enfin le train de marchandises).

Le but est que le traffic soit fluide, et qu'aucune collision entre les trains n'apparaisse.

Les trains sont des threads possédants une priorité plus ou moins importante, et qui demandent des voies (sémaphores). Ces voies sont regroupées dans des dictionnaires (clef, valeur) qui correspondent aux différents postes d'aiguillages. Ceux-ci sont ensuite envoyés à un «serveur central »qui accorde ou non les trains à passer. Si le serveur accepte, le train prends toutes les sémaphores et traversent les voies. Il rend ensuite les sémaphores (voies) inutilisées (toutes les précédentes sauf la courante), et demandent les voies suivantes. Dans le cas où un train ne peut pas prendre les voies, il doit attendre sur sa voie actuelle.

1.2 Les structures de données

Le programme utilise les principales structures suivantes :

```
typedef struct {
    dictionary* disponibilites;
    pthread_mutex_t mutex_modification;
} banquier;
```

La structure banquier, il s'agit de notre «serveur central ». Il possède un mutex lui permettant de modifier ou non les disponibilites des voies.

```
typedef struct ___dictionary_element{
    void* key;
    void* value;
    struct ___dictionary_element* next;
} dictionary_element;

typedef dictionary_element dictionary;
```

La structure dictionary, comme son nom l'indique il s'agit d'un dictionnaire associant les voies et leur capacité a accueillir 1 ou 2 trains.

```
typedef struct ___dictionary_char_element{
    char* key;
    void* value;
    struct ___dictionary_char_element* next;
} dictionary_char_element;

typedef dictionary_char_element dictionary_char;
```

La structure dictionary_char qui associe également un nom de voie à une capacité. Nous

verrons la différence entre ces deux structures plus tard.

```
typedef struct {
    char* nom;
    direction direction;
    dictionary_char* postes;
    banquier* banquier;
} train;
```

La structure train, qui possède un nom, une direction, une liste des postes d'aiguillage et un serveur réservé.

```
typedef enum {
    EO = 1,
    OE = 2
} direction;
```

Le type direction qui peut-être Est-Ouest ou Est-Ouest.

2 Les choix techniques

2.1 Le multi-threading

Nous avons fait le choix des threads pour modéliser les trains, car contrairement aux processus, ceux-ci partagent leurs variables locales, globales et leur descripteurs de fichiers. Ce qui facilitent énormément la programmation et la rapidité du programme.

Leur création n'impliquent donc pas la création d'un nouveau contexte, de même que leur ordonnancement (pas de context-switching), ce qui allège le système et le fonctionnement du programme.

C'est pourquoi nous avons préféré l'utilisation de threads décrits dans la norme POSIX, les pthreads.

Il faut cependant savoir que les threads n'offrent pas que des avantages. Le partage de données nécessite souvent d'utiliser des techniques permettant de protéger à un instant donné l'accès concurrent à une variable partagée. Nous avons pour cela utilisé des sémaphores et des mutex.

2.2 Mutex et sémaphores

Nous avons choisis des sémaphores pour les voies de trains, car leurs valeurs initiales associées permettant de configurer facilement combien de trains pourront avoir accès à une voie en particulier.

Nous avons cependant utilisé un mutex pour la libération et la prise de voies par le serveur central (le banquier), car il s'agissait seulement d'autoriser ou non l'entrée dans la section critique. Une sémaphore binaire aurait également marché, mais ce n'est rien d'autre qu'un mutex finalement.

3 L'implémentation

3.1 Le réseau ferroviaire

Le réseau ferroviaire est constitué de différentes voies et postes d'aiguillages, ainsi que de lieux clés (gare, tunnel, etc.).
Le programme, avant toute chose, commence donc par créer des voies.

3.1.1 Les voies

Les voies sont de simples sémaphores :

```
sem_t *voie_A = (sem_t*) malloc(sizeof (sem_t)),
      *voie_B = (sem_t*) malloc(sizeof (sem_t)),
      *voie_C = (sem_t*) malloc(sizeof (sem_t)),
      *voie_D = (sem_t*) malloc(sizeof (sem_t)),
      *ligne_M_EO = (sem_t*) malloc(sizeof (sem_t)),
      *ligne_M_OE = (sem_t*) malloc(sizeof (sem_t)),
      *aiguillage_1 = (sem_t*) malloc(sizeof (sem_t)),
      *aiguillage_2 = (sem_t*) malloc(sizeof (sem_t)),
      *ligne_TGV = (sem_t*) malloc(sizeof (sem_t)),
      *ligne_GL = (sem_t*) malloc(sizeof (sem_t)),
      *tunnel = (sem_t*) malloc(sizeof (sem_t)),
      *ligne = (sem_t*) malloc(sizeof (sem_t));
```

```
sem_init(voie_A, 0, 2);
sem_init(voie_B, 0, 2);
sem_init(voie_C, 0, 2);
sem_init(voie_D, 0, 2);
sem_init(ligne_M_EO, 0, 2);
sem_init(ligne_M_OE, 0, 2);

sem_init(aiguillage_1, 0, 1);
sem_init(aiguillage_2, 0, 1);
sem_init(ligne_TGV, 0, 1);
sem_init(ligne_GL, 0, 1);
sem_init(tunnel, 0, 1);
sem_init(ligne, 0, 1);
```

Celles-ci sont initialisées selon le nombre de trains quelles peuvent accueillir (deux trains allant dans le même sens dans le cas de la gare par exemple).

```
Gare = dictionary_char_new("Voie C", (void*) voie_C);
dictionary_char_add(Gare, "Voie D", (void*) voie_D);

P0 = dictionary_char_new("Voie C", (void*) voie_C);
dictionary_char_add(P0, "Voie D", (void*) voie_D);
dictionary_char_add(P0, "Aiguillage 2", (void*) aiguillage_2);

P1 = dictionary_char_new("Aiguillage 1", (void*) aiguillage_1);
dictionary_char_add(P1, "Aiguillage 2", (void*) aiguillage_2);
dictionary_char_add(P1, "Ligne TGV", (void*) ligne_TGV);
dictionary_char_add(P1, "Ligne GL", (void*) ligne_GL);
dictionary_char_add(P1, "Ligne M - EO", (void*) ligne_M_EO);
dictionary_char_add(P1, "Ligne M - OE", (void*) ligne_M_OE);
dictionary_char_add(P1, "Voie A", (void*) voie_A);
dictionary_char_add(P1, "Voie B", (void*) voie_B);

P2 = dictionary_char_new("Ligne TGV", (void*) ligne_TGV);
dictionary_char_add(P2, "Ligne GL", (void*) ligne_GL);
dictionary_char_add(P2, "Ligne M - EO", (void*) ligne_M_EO);
dictionary_char_add(P2, "Ligne M - OE", (void*) ligne_M_OE);
dictionary_char_add(P2, "Tunnel", (void*) tunnel);

P3 = dictionary_char_new("Tunnel", (void*) tunnel);
dictionary_char_add(P3, "Ligne", (void*) ligne);
```

Elles sont ensuites regroupées par postes d'aiguillages (la gare étant considérée comme un poste). Les sémaphores sont castées en void* car nous avons codé le dictionnaire pour accepter tout types de données. Il peut ainsi être réutilisé dans un autre cadre.

3.1.2 Les postes d'aiguillages

Les postes d'aiguillages sont des dictionnaires associant une clé texte à une sémaphore. Ils utilisent la structure dictionary_char vue précédemment.

Ces postes d'aiguillages sont ensuite regroupés dans un dictionnaire de dictionnaires, nommé postes d'aiguillages :

```
postes_aiguillages = dictionary_char_new("Gare", (void*) Gare);
dictionary_char_add(postes_aiguillages, "P0", (void*) P0);
dictionary_char_add(postes_aiguillages, "P1", (void*) P1);
dictionary_char_add(postes_aiguillages, "P2", (void*) P2);
dictionary_char_add(postes_aiguillages, "P3", (void*) P3);
```

3.2 L'aiguillage : le banquier

Tout l'aiguillage se fait par l'intermédiaire d'un «serveur central » qui accorde ou non le passage du train sur les voies demandées en donnant ou non des sémaphores (voies).

On commence donc par créer un ensemble de ressources disponibles (nos voies) :

```
    dictionary *p;

    int *voie_A = (int*) malloc(sizeof (int)),
        *voie_B = (int*) malloc(sizeof (int)),
        *voie_C = (int*) malloc(sizeof (int)),
        *voie_D = (int*) malloc(sizeof (int)),
        *ligne_M_EO = (int*) malloc(sizeof (int)),
        *ligne_M_OE = (int*) malloc(sizeof (int)),
        *aiguillage_1 = (int*) malloc(sizeof (int)),
        *aiguillage_2 = (int*) malloc(sizeof (int)),
        *ligne_TGV = (int*) malloc(sizeof (int)),
        *ligne_GL = (int*) malloc(sizeof (int)),
        *tunnel = (int*) malloc(sizeof (int)),
        *ligne = (int*) malloc(sizeof (int));

    *voie_A = *voie_B = *voie_C = *voie_D = *ligne_M_EO = *ligne_M_OE = 2;
    *aiguillage_1 = *aiguillage_2 = *ligne_TGV = *ligne_GL = *tunnel = *ligne = 1;

    p = dictionary_new(dictionary_char_2d_get(postes, "P1", "Voie A"), voie_A);
    dictionary_add(p, dictionary_char_2d_get(postes, "P1", "Voie B"), voie_B);
    dictionary_add(p, dictionary_char_2d_get(postes, "Gare", "Voie C"), voie_C);
    dictionary_add(p, dictionary_char_2d_get(postes, "Gare", "Voie D"), voie_D);
    dictionary_add(p, dictionary_char_2d_get(postes, "P1", "Aiguillage 1"), aiguillage_1);
    dictionary_add(p, dictionary_char_2d_get(postes, "P1", "Aiguillage 2"), aiguillage_2);
    dictionary_add(p, dictionary_char_2d_get(postes, "P1", "Ligne TGV"), ligne_TGV);
    dictionary_add(p, dictionary_char_2d_get(postes, "P1", "Ligne M - EO"), ligne_M_EO);
    dictionary_add(p, dictionary_char_2d_get(postes, "P1", "Ligne M - OE"), ligne_M_OE);
    dictionary_add(p, dictionary_char_2d_get(postes, "P1", "Ligne GL"), ligne_GL);
    dictionary_add(p, dictionary_char_2d_get(postes, "P2", "Tunnel"), tunnel);
    dictionary_add(p, dictionary_char_2d_get(postes, "P3", "Ligne"), ligne);
```

Nos ressources disponibles sont un dictionnaire à clé void*, qui associe une sémaphore à une valeure entière. On aurait pu faire plus simple directement, mais il s'agit ici de simuler une demande aux différents postes d'aiguillage des voies libres.

Une fois que notre dictionnaire de ressources disponibles est prêt, on crée le serveur central banquier, avec ce dictionnaire initial.

```
banquier_new(p)
```

```
banquier* banquier_new(dictionary *ressources_disponibles) {
    banquier *b = (banquier*) malloc(sizeof (banquier));

    b->disponibilites = ressources_disponibles;

    pthread_mutex_init(&b->mutex_modification, NULL);

    return b;
}
```

Banquier est une structure possédant deux champs, comme on l'a vu plus haut. On assigne au banquier les ressources disponibles, et on initialise son mutex.

3.3 Les trains

Tout est prêt, on peut maintenant créer des trains.

Les trains sont des threads crées selon un nombre passé en paramètre du programme, ou définit dans le code source, on peut monter jusqu'à 10 000 sans soucis (nous n'avons pas testé avec plus).

Le type de train est généré de manière aléatoire, ainsi que sa direction.

Par exemple pour un TGV :

```
switch (rand() % 3) {
        case 0: /* TGV */
            nom = malloc(sizeof (char) * 8);
            sprintf(nom, "TGV %d", i);

            trains[i].nom = nom;
            trains[i].postes = postes_aiguillages;
            trains[i].banquier = banquier;

            switch (rand() % 2) {
                case 0:
                    trains[i].direction = OE;
                    break;
                case 1:
                    trains[i].direction = EO;
                    break;
            }

            pthread_create(&(trains_threads[i]), NULL, TGV_thread_fn, &(trains[i]));
            pthread_setschedprio(trains_threads[i], 16);

            break;
        /* ... */
}
```

On fournit au train son nom, les postes d'aiguillages du réseau (qui eux même contiennent les voies), le serveur central banquier ainsi que sa direction.

La priorité du thread train dépend du type de train, elle est de 16 pour un TGV, 4 pour un train grande ligne et 1 pour un train de marchandise. L'ordonnanceur de l'ordinateur donnera ainsi la priorité au TGVs pour accéder au serveur central banquier (on le verra pas la suite), et donc la priorité pour réserver les voies et avancer. On crée le thread avec la routine correspondante, et le train en argument.

Cette routine va créer une liste de voies désirées (exemple d'un TGV Ouest-Est) :

```
l = liste_new(dictionary_char_2d_get(TGV->postes, "Gare", "Voie C"));
        liste_add(l, dictionary_char_2d_get(TGV->postes, "P0", "Aiguillage 2"));
        liste_add(l, dictionary_char_2d_get(TGV->postes, "P1", "Ligne TGV"));
```

Et demande au banquier de bloquer ces voies pour lui :

```
banquier_lock(TGV->banquier, l);
```

```
do {
        pthread_mutex_lock(&self->mutex_modification);

        d = self->disponibilites;
        is_dispo = 1;

        /* On vérifie si toutes les ressources (voies) demandées sont dispos */
        while (d != NULL && is_dispo == 1) {
            l = semaphores;

            while (l != NULL && is_dispo == 1) {
                if (l->value == d->key && *(int*) (d->value) == 0) {
                    is_dispo = 0 && is_dispo;
                }

                l = l->next;
            }

            d = d->next;
        }

        /* Si jamais toutes les ressources (voies) demandées sont dispos, on les prends */
        if (is_dispo == 1) {
            d = self->disponibilites;

            while (d != NULL) {
                l = semaphores;

                while (l != NULL) {
                    if (l->value == d->key)
                        *(int*) (d->value) -= 1;

                    l = l->next;
                }

                d = d->next;
            }

            l = semaphores;

            while (l != NULL) {
                sem_wait((sem_t*) (l->value));

                l = l->next;
            }
        }

        /* On libère la ressource dans tous les cas */
        pthread_mutex_unlock(&self->mutex_modification);
    } while (is_dispo == 0);
}
```

Si aucun autre thread ne tente d'accéder au banquier (mutex disponible), le banquier commence par vérifier si toutes les ressources demandées sont disponibles, si elles ne le sont pas, le thread train est mis en attente et le banquier est disponible pour un autre thread (on rend le mutex). Si toutes les ressources (voies) demandées sont disponibles, on les réserve toutes, et on libère le banquier pour un autre thread train. Des trains avec des parcours différents peuvent ainsi circuler simultanément.

```
            liste_del(l);

            l = liste_new(dictionary_char_2d_get(TGV->postes, "Gare", "Voie C"));
            liste_add(l, dictionary_char_2d_get(TGV->postes, "P0", "Aiguillage 2"));

            printf("%p - %s - OE - releasing : Voie C, Aiguillage 2\n", (void*) (pthread_self()), TGV->nom);

            banquier_unlock(TGV->banquier, l);
```

Le parcours du train n'est pas fini, et celui-ci libère les voies qu'ils n'utilisent plus (toutes les précédentes, sauf la courante), en faisant appel à banquier_unlock.

```
void banquier_unlock(banquier* self, list* l) {
    list_element *e;
    dictionary_element* d;

    if (l == NULL || self->disponibilites == NULL)
        return;

    pthread_mutex_lock(&self->mutex_modification);

    d = self->disponibilites;

    while (d != NULL) {
        e = l;

        while (e != NULL) {
            if (e->value == d->key) {
                *(int*) (d->value) += 1;
            }

            e = e->next;
        }

        d = d->next;
    }

    e = l;

    while (e != NULL) {
        sem_post((sem_t*) (e->value));

        e = e->next;
    }

    pthread_mutex_unlock(&self->mutex_modification);
}
```

Cette fonction est très similaire à banquier_lock. On utilise ici le même mutex (mutex_modification), cela permet de ne pas effectuer de banquier_unlock et de banquier_lock en même temps.

On peut constater que la fonction banquier_lock effectue une attente semi-active via la boucle do...while (heureusement la fonction reste appelable par tous les threads, ce n'est donc pas gênant au final). Il serait cependant plus approprié d'utiliser une attente passive. C'est ce que nous avons commencé à faire : nos threads demandant des voies non disponibles attendaient via un wait, et lorsque l'ordonnanceur passait au thread qui utilisait les ressources, et que celui-ci les libérait, il faisait un broadcast (et un mutex_unlock). Seulement le temps que l'ordonnanceur réélise le premier thread qui était bloqué et attendait les ressources, le broadcast était déjà passé et le thread attendait un signal qui

était déjà passé. Le programme continuant, il pouvait reprendre au broadcast suivant, seulement il y avait un problème quand tout à la fin, toute la mémoire était libéré, les broadcasts ne pouvaient plus se faire, et les quelques threads restants attendaient à l'infini.

Les trains continuent leur avancée comme ça, demandant morceaux par morceaux leur chemin, le prenant quand ils peuvent, et attendant quand ce n'est pas possible. Les priorités liés aux types de threads (TGV, GL ou M) permettent à ces threads d'être plus ou moins souvent élus par l'ordonnanceur et donc d'appeler plus ou moins souvent le banquier.

4 Conclusion

Ce projet a été particulièrement intéressant puisqu'il nous a permis d'approfondir des aspects qui ont été rapidement abordés dans le cadre de l'UV, à savoir les threads et l'algorithme du banquier.

Ces aspects du C système sont d'autant plus intéressant lorsque l'on prend conscience qu'un système UNIX possède de nombreux logiciels basés sur ces techniques.

Ce projet s'inscrit donc dans la continuation du programme de TP à la différence près que nous n'étions pas guidé et que nous devions de ce fait effectuer des choix divers et variés quand à l'implémentation de celui-ci.

Réseau de pétri

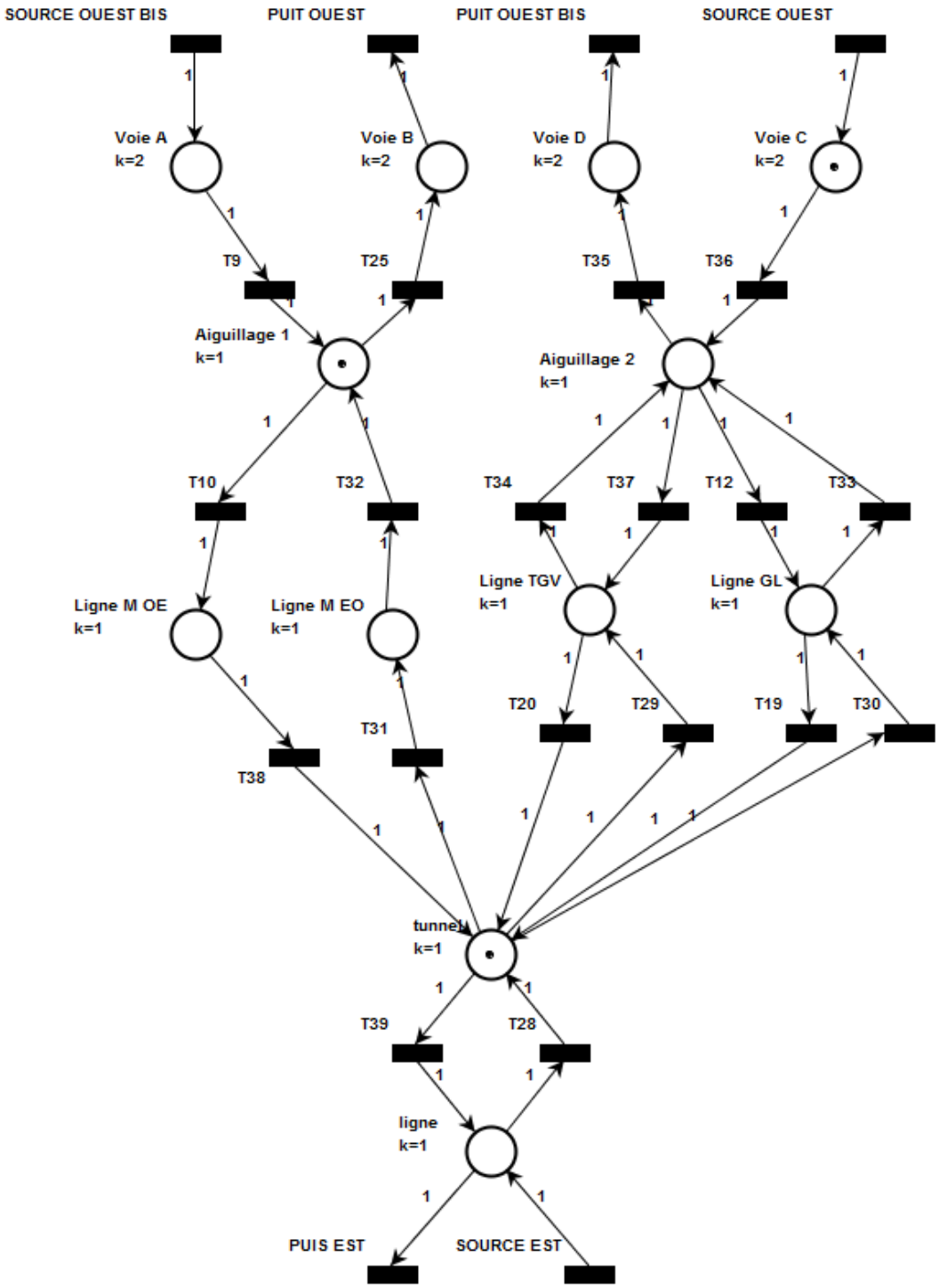

Figure 1 – Réseau de pétri du traffic ferroviaire