Hubert LACÔTE
Guillaume STUBER

LO41

Projet : Système de gestion routier

Responsable de l'UV : Philippe DESCAMPS

Table des matières

1. Introduction

Dans ce rapport, nous allons détailler la conception de notre projet pour l'UV LO41 (Architecture et utilisation des systèmes d'exploitation). Notre projet concerne la création d'un système de gestion routier. Les principales contraintes du sujet sont que le projet doit fonctionner sous Solaris, qu'il doit être programmé en C système en utilisant les notions acquises ce semestre dans l'UV. (Création de processus, files de messages, signaux, mémoire partagée, sémaphores, ...)

Nous avons choisi de traiter le sujet du système de gestion routier car c'était le sujet qui nous paraissait le plus attractif et qui nous permettait de mettre en application un maximum les connaissances acquises ce semestre.

Pour traiter le sujet, nous avons respecté un certain planning : nous avons pris connaissance du sujet fin novembre, puis nous avons élaboré un cahier des charges pour mi-décembre qui présente le fonctionnement global du programme avec l'interaction des différents acteurs. A partir du cahier des charges, nous avons eu un mois pour réaliser le programme.

Nous détaillerons dans ce rapport le fonctionnement du programme, et pour ce faire, nous vous présenterons les parties importantes du code. Vous pourrez bien sûr vous référer au code source commenté en annexe si vous souhaitez approfondir le fonctionnement.

2. Cahier des charges

2.1. Énoncé du sujet

Le projet consiste à décliner le comportement d'un Système d'Aide à l'Exploitation – SAE. Le gestionnaire de ce système assure au quotidien la bonne marche d'une flotte de véhicules. L'exploitant doit répondre aux aléas de la circulation (accidents, travaux) et aux problèmes techniques (panne du véhicule, absence de conducteurs). Par le SAE, il est capable de prendre des décisions qui pourront affecter un ou plusieurs véhicules.

Dans ce contexte, nous devons être capable de simuler le comportement élémentaire d'un réseau de bus (circonscrit à deux lignes du réseau aller-retour et à une dizaine d'arrêts de bus) et de la gestion des incidents vers un poste de régulation.

2.2. Compréhension

La simulation sera donc composée de deux lignes de bus, comportant chacune 10 arrêts. Pour cela, chaque bus sera représenté par un processus, communiquant avec le gestionnaire ainsi qu'avec les arrêts de bus, eux-mêmes représentés par des processus distincts. Le gestionnaire du réseau de bus doit pouvoir agir directement (donner des ordres) lors des cas suivants :

- Accidents/ Travaux
- Panne de Bus

2.3. Objectifs fixés

a) Arrêt de bus

Chaque arrêt de bus possèdera un nombre de personnes attendant le bus. Ce nombre s'incrémentera au fur et à mesure du temps (boucle avec un random) qui pourra être interrompu par un signal, dont le traitant écrira dans une file de message le nombre de personnes en attente afin de le communiquer au bus lui ayant envoyé le signal.

b) Ligne de bus

Chaque ligne de bus comportera un certain nombre d'arrêt de bus, chacun représenté par une structure de donnée comportant par exemple le temps moyen de trajet et le temps moyen de l'itinéraire de secours pour aller à l'arrêt suivant, la présence d'un accident ou de travaux entre les deux arrêts. Cette ligne de bus devra être accessible par chacun des bus, ainsi elle sera certainement représentée par une mémoire partagée.

c) Bus

Chaque bus se verra attribué une ligne de bus qu'il pourra consulter dans la mémoire partagée. Il aura une certaine capacité N, qu'il ne pourra pas dépasser. A chaque arrêt, le nombre de passagers variera suivant le nombre de personnes présentes à l'arrêt, ainsi que du nombre de personne descendant (aléatoirement défini). Lorsqu'il arrive à un arrêt, le bus lui envoie un signal, il récupère dans une file de message le nombre de personnes en attente, et écrit dans la file de message le nombre de personnes qu'il prend effectivement(dans le cas où le bus est plein).

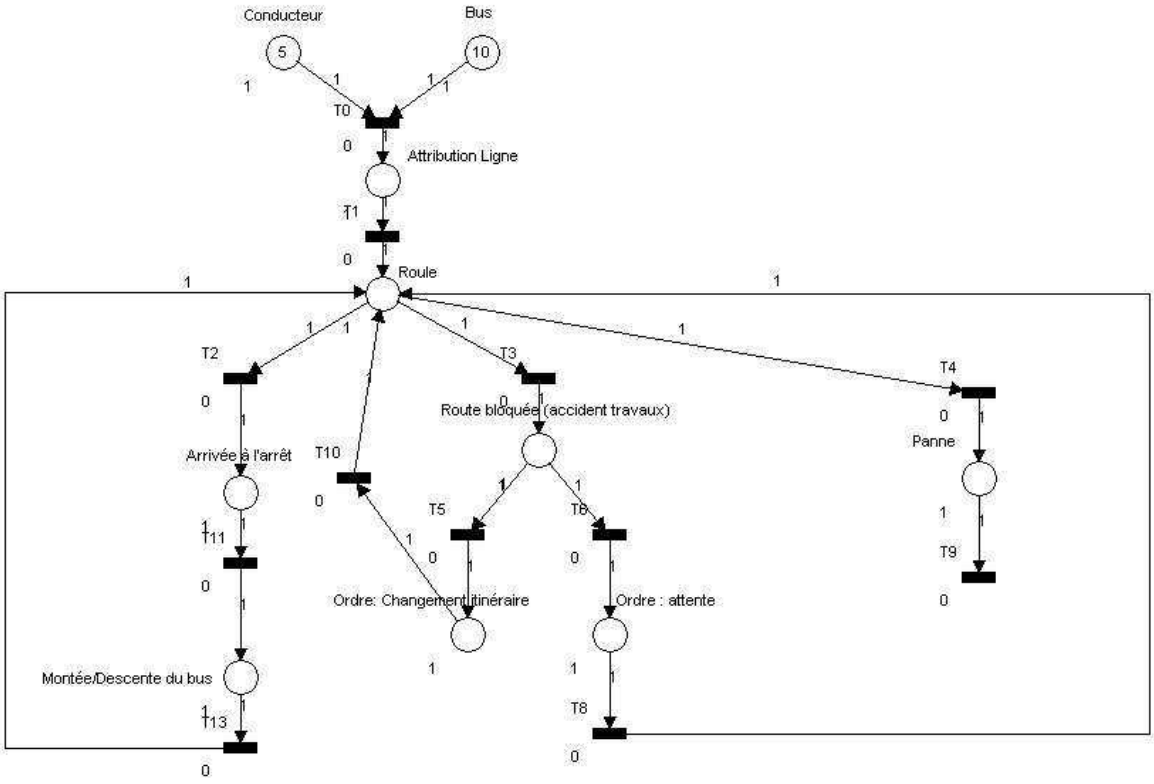
d) Accident/Travaux

Un processus se chargera de mettre des accidents/travaux sur les deux lignes de bus. Pour cela, il aura accès aux deux mémoires partagées Ligne de bus.

Un accident/travaux ou une panne mettra le système en pause et demandera au gestionnaire quoi faire. Selon le cas, celui-ci pourra donner l'ordre d'attendre, de prendre l'itinéraire de secours, etc...

2.4. Fonctionnement simplifié avec un réseau de Petri

Ce réseau de pétri décrit le fonctionnement simplifié de notre programme en présentant la marche d'un bus et les différents états qu'il peut prendre.



3. Fonctionnement du programme

3.1. Présentation des différentes entités créées

Nous allons maintenant présenter les entités telles qu'elles ont été créées dans le projet, nous expliquerons brièvement leur fonctionnement et présenterons du pseudo-code permettant de comprendre rapidement le fonctionnement du programme. Les points nécessitant une explication seront repris dans la suite du rapport.

a) Description des principales caractéristiques des entités

Nous utilisons dans notre projet 5 types de processus différents :

- *Le programme principal*

Sa fonction est de créer les arrêts de bus, les bus, le générateur d'accident et le système d'aide à l'exploitation. Il se charge aussi de mettre en place la mémoire partagée et les mutex associés.

Programme principal *Principal.c*
DEBUT

Mise en place de la simulation :

Créer la mémoire partagée
Initialiser le tableau de sémaphores
Créer les arrêts de bus de la ligne 1
Créer les arrêts de bus de la ligne 2
Enlever un jeton à la sémaphore busCrees
Enlever un jeton à la sémaphore saeCree
Créer le SAE
Créer les bus de la ligne 1
Créer les bus de la ligne 2
Créer le générateur d'accident

Instaurer le traitant pour le signal SIGINT : le traitant exécutera detruireSimulation();

Attendre le signal SIGINT

FIN

detruireSimulation *Principal.c*
DEBUT

Détruire la mémoire partagée
Détruire le tableau de sémaphores
FIN

- *Les arrêts de bus*

Chaque arrêt de bus possède une file de messages pour communiquer avec les bus. A chaque arrêt de bus est associé de la mémoire partagée qui contient les informations de l'arrêt (l'identifiant de la file de messages, la présence d'un accident, et si il y a un accident, le choix du gestionnaire pour tous les bus).

```
creerArretBus                                                    ArretBus.c
DEBUT

        Création de la file de messages de l'arrêt de bus

        Initialiser la mémoire partagée de l'arrêt de bus :
                accident = 0
                l'identificateur de la file de messages de l'arrêt

        Instaurer le traitant pour le signal SIGINT : le traitant exécutera detruireArretBus();

        Lancer la boucle sans fin de l'arrêt de bus :
                nbPersonnesEnAttente = 0
                Tant que vrai
                        Répéter
                                pidBus = prendreBusEnAttente
                                Si pidBus > 0
                                        nbPersonnesEnAttente -= transaction de passagers avec le bus
                        Tant que il y a un bus à l'arrêt

                        incrémenter aléatoirement nbPersonnesEnAttente

                        attendre 1 seconde

                Fin tant que
FIN
```

```
detruireArretBus                                                 ArretBus.c
DEBUT

        Détruire la file de messages de l'arrêt de bus
FIN
```

- *Les bus*

A chaque bus est attribué un numéro de ligne. Les bus partent les uns après les autres séparés d'un certain temps. Chaque bus va ensuite démarrer si le conducteur est présent puis parcourra tout les arrêts de sa ligne en effectuant ces actions : arriver à un arrêt, faire descendre des passagers, prendre les passagers attendant à l'arrêt de bus puis se rendre à l'arrêt prochain en tombant éventuellement en panne et en gérant les accidents sur le parcours.

```
creerBus                                                    Bus.c
DEBUT

    Initialiser la mémoire partagée du bus :
        Le placer à l'arrêt 0 dans le sens 1 avec 0 passager.
        Vider son message.
        Aucune panne , sans conducteur.

    Ajouter un jeton permettant de signaler qu'un bus a été créé

    Attente que le SAE soit créé

    Attendre le temps avant le départ

    Si le conducteur est arrivé alors

        Lancer la boucle du bus :
            Tant que vrai

                Faire descendre les passagers
                Prendre les passagers à l'arrêt de bus
                Rouler vers le prochain arrêt

            Fin tant que

    Sinon

        Quitter le processus

    Fin si

FIN
```

- *Le générateur d'accidents*

La fonction du générateur d'accidents est de placer aléatoirement des accidents sur une des deux lignes de bus. Pour cela, il demandera au gestionnaire comment devront réagir les bus qui seront confrontés à l'accident. Une fois la réponse du gestionnaire reçue, il inscrira dans la mémoire partagée de l'arrêt de bus choisi l'accident et le choix du gestionnaire, ce qui permettra à tous les bus de connaître le choix du gestionnaire si ils sont confrontés à l'accident.

```
creerGenerateurAccident                                          Accident.c
DEBUT

        Tant que vrai

                Attendre 3 secondes

                Générer un nombre aléatoire

                Si nombreAleatoire = 0 Alors

                        Bloquer tous les processus

                        Choisir la ligne et l'arrêt de l'accident

                        Demander au gestionnaire son choix concernant l'accident

                        Écrire l'accident et le choix du gestionnaire dans la mémoire partagée

                        Débloquer tous les processus

                        Attendre la durée d'un accident

                        Enlever l'accident de la mémoire partagée

                Fin Si

        Fin tant que

FIN
```

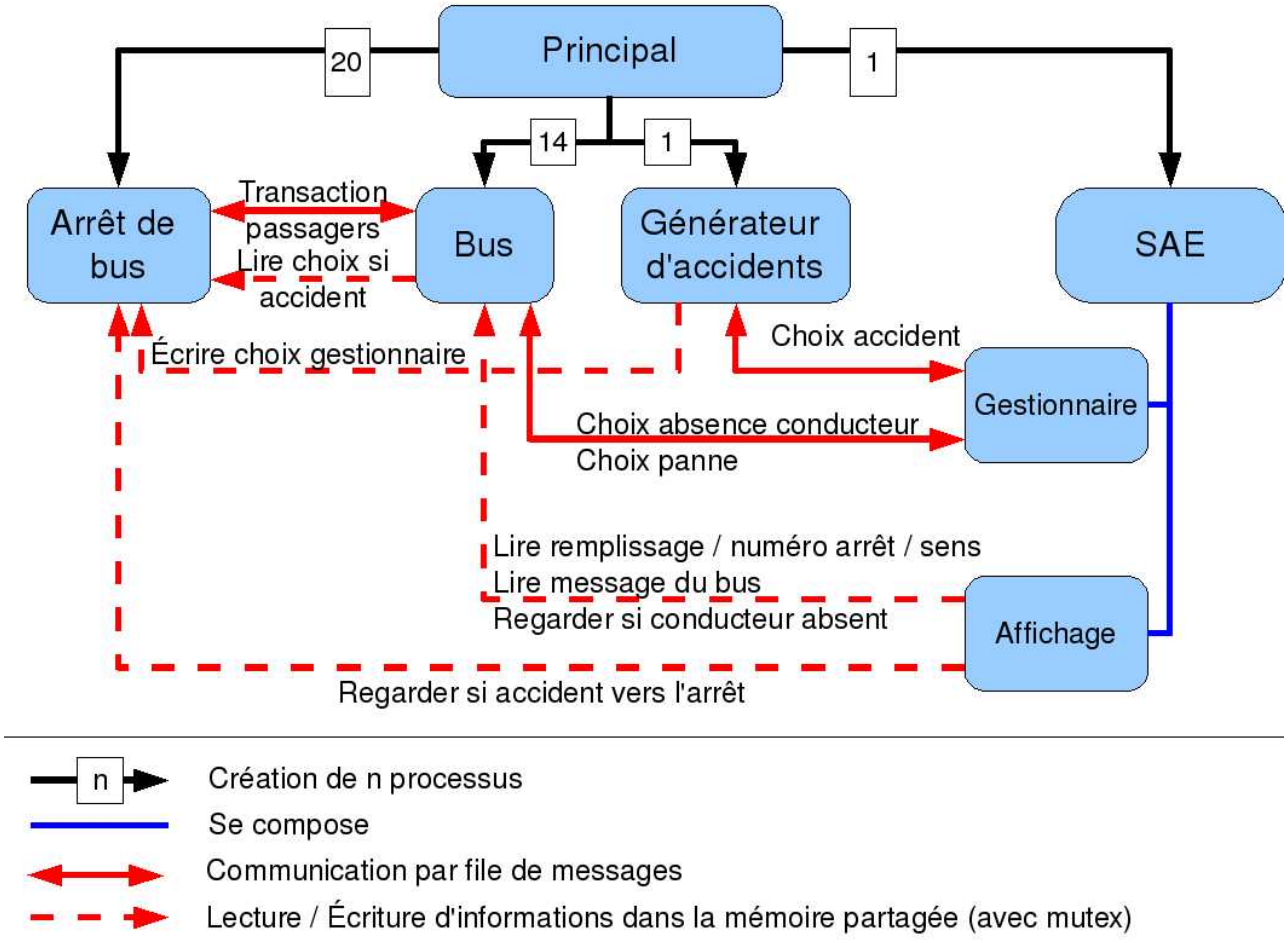
- *Le Système d'Aide à l'Exploitation (SAE)*

Il se compose de l'affichage qui lira dans la mémoire partagée les différentes informations à afficher (à l'aide de mutex) et du gestionnaire qui traite les problèmes reçus dans sa file de messages en demandant à l'utilisateur de faire des choix.

```
creerSAE                                                                    SAE.c
DEBUT

        Créer la file de messages du gestionnaire

        Écrire dans la mémoire partagée l'identificateur de la file de messages

        Instaurer le traitant pour le signal SIGINT : le traitant exécutera detruireSAE();

        Attendre que les bus soient tous créés

        Informer que le SAE est créé

        Lancer la boucle du SAE :

                Afficher la ligne de bus 1

                Afficher la ligne de bus 2

                Traiter les problèmes :

                        Si Accident Alors
                                Afficher le menu accident
                        Sinon
                                Si Panne Alors
                                        Afficher le menu Panne
                                Sinon
                                        Si Absence de conducteur Alors
                                                Afficher le menu absence
                                        Fin Si
                                Fin Si
                        Fin Si

                Attendre 1 seconde
FIN
```

```
detruireSAE                                                                 SAE.c
DEBUT
        Mettre fin à l'affichage
        Détruire la file de messages du SAE
FIN
```

b) Schéma présentant les relations entre les entités



3.2. Présentation de la communication entre les entités

a) Communication par file de messages

Comme nous pouvons le voir sur le schéma présentant les relations entre les entités, trois communication par file de messages sont mises en place dans le projet. Nous avons choisi d'utiliser les files de messages pour créer un certain réalisme (une communication est réellement mise en place). Nous avons privilégié les files de messages par rapport à d'autres méthodes (utilisation des sockets, ...) puisque c'est la méthode que nous avons le plus travaillé pendant le semestre. Elle permet aussi de créer facilement une certaine synchronisation.

- *La communication entre bus et arrêts de bus*

Utilisation des fichiers : ArretBus.c Bus.c FileMessageArretBus.c.

Une file de messages est créée par chaque arrêt de bus pour assurer la communication avec les bus. Les fonctions utilisées par les bus et par les arrêts de bus sont présentes dans FileMessageArretBus.c.

Communication simplifiée du bus	Communication simplifiée de l'arrêt	Type	Message
Ecrire le pid du bus en attente		1	Pid du bus
	Lire pid du bus (non bloquant)	1	Pid du bus
	Ecrire le nombre de personnes à l'arrêt	Pid bus	Nombre p.
Lire le nombre de personnes à l'arrêt		Pid bus	Nombre p.
Ecrire le nombre de personnes prises		2	Nombre p.
	Lire le nombre de personnes prises	2	Nombre p.

Nous pouvons noter que l'arrêt regarde en permanence si un bus est arrivé à son arrêt à l'aide d'une lecture non bloquante dans sa file de messages. De plus l'utilisation d'une file de messages assure la synchronisation des bus. En effet l'arrêt de bus traitera chaque bus l'un après l'autre, et les bus seront en attente à l'arrêt jusqu'à ce qu'ils soient sélectionnés par l'arrêt de bus pour prendre des passagers.

- La communication avec le gestionnaire en cas de problèmes

Utilisation des fichiers : SAE.c Accident.c Bus.c FileMessageGestionnaire.c.

Une file de messages est créée par le gestionnaire pour assurer la communication avec :

- les bus en cas d'absence de conducteurs ou de panne
- le générateur d'accident si un accident a été introduit

Les fonctions utilisées par les bus et par le générateur d'accident pour communiquer avec le gestionnaire sont présentes dans FileMessageGestionnaire.c.

Voici les trois différentes communication avec le gestionnaire :

- Cas d'une absence de conducteur

Lors de la création du bus, il est aléatoirement déterminé si le conducteur est absent. Si il est absent, une communication avec le gestionnaire est alors engagée :

Communication du bus	Communication du SAE	Type	Message
Écrire une absence en attente		3	Pid du bus Num. ligne / bus
	Lire pid du bus Non bloquant	3	Pid du bus Num. ligne / bus
	Écrire temps choisi par gestionnaire > 0 : attendre autre conducteurs -1 : ne pas faire rouler le bus	Pid bus	Temps
Lire temps choisi par gestionnaire		Pid bus	Temps

- Cas d'une panne du bus

Lorsque le bus se rend à l'arrêt suivant, il peut aléatoirement tomber en panne. Si c'est le cas, il engage alors une communication avec le gestionnaire :

Communication du bus	Communication du SAE	Type	Message
Écrire une panne en attente		2	Pid du bus Num. ligne / bus
	Lire pid du bus Non bloquant	2	Pid du bus Num. ligne / bus
	Écrire temps choisi par gestionnaire Variant suivant choix (réparateur ou autre bus)	Pid bus	Temps
Lire temps choisi par gestionnaire		Pid bus	Temps

- Cas d'un accident

Lorsque le générateur d'accident a sélectionné une ligne et un arrêt après lequel il y aura un accident, il engage alors une communication avec le gestionnaire pour savoir ce que devra faire chaque bus qui sera confronté à l'accident :

Communication du générateur d'accident	Communication du SAE	Type	Message
Écrire un accident en attente		1	Pid du générateur Num. ligne / arrêt
	Lire pid du générateur Non bloquant	1	Pid du générateur Num. ligne / arrêt
	Écrire temps choisi par gestionnaire > 0 : Prendre détour (temps) -1 : attendre fin de l'accident	Pid gé	Temps
Lire temps choisi par gestionnaire		Pid gé	Temps

Nous pouvons noter que le SAE ne possède qu'une seule file de messages dans laquelle trois types de communication transitent. L'utilisation de cette file de messages permet comme pour la transaction entre bus et arrêt de bus une certaine synchronisation. En effet, si plusieurs pannes surviennent en même temps, le SAE traitera les pannes les unes après les autres. De même, le SAE traitera d'abord les accidents, ensuite les pannes et en dernier les absences. Cela permet de n'avoir qu'une seule question à la fois qui apparaît pour le gestionnaire.

b) Communication à l'aide de la mémoire partagée

Comme nous pouvons le voir sur le schéma présentant les relations entre les entités, certaines entités communiquent à l'aide de lecture et d'écriture dans la mémoire partagée. En effet, nous pouvons voir que l'affichage communique avec le reste du programme uniquement grâce à la mémoire partagée. De même, le générateur d'accident va écrire le choix du gestionnaire concernant les accidents dans la mémoire partagée de l'arrêt de bus concerné et chaque bus lira dans cette mémoire pour savoir comment il doit se comporter.

- *Présentation de la mémoire partagée*

Chaque arrêt de bus, chaque bus ainsi que le gestionnaire possèdent un espace de mémoire partagée qui leur est propre. L'ensemble des structures concernant la mémoire partagée ainsi que les accès à la mémoire partagée se trouve dans le fichier MemoirePartagee.c. Nous avons déclaré pour l'ensemble de la mémoire partagée des méthodes permettant de lire et d'écrire dans la mémoire partagée en utilisant des mutex. Voici les champs que possèdent chaque entité avec les prototypes des méthodes qui permettent d'y accéder :

Arrêt de bus *Nom de la structure pour stocker ces champs :* ***ArretLigne***	void mem_arretBus_msgId_ecrire(int numLigne, int numArret, int msgId) int mem_arretBus_msgId_lire(int numLigne, int numArret) void mem_arretBus_accident_ecrire(int numLigne, int numArret, int accident) int mem_arretBus_accident_lire(int numLigne, int numArret) void mem_arretBus_tempsSecours_ecrire(int numLigne, int numArret, int tempsSecours) int mem_arretBus_tempsSecours_lire(int numLigne, int numArret)
int msgId int accident int tempsSecours	
Bus *Nom de la structure pour stocker ces champs :* ***InfoBus***	void mem_bus_numArret_ecrire(int numLigne, int numBus, int numArret) int mem_bus_numArret_lire(int numLigne, int numBus) void mem_bus_sens_ecrire(int numLigne, int numBus, int sens) int mem_bus_sens_lire(int numLigne, int numBus) void mem_bus_nbPassagers_ecrire(int numLigne, int numBus, int nbPassagers) int mem_bus_nbPassagers_lire(int numLigne, int numBus) void mem_bus_absenceConducteur_ecrire(int numLigne, int numBus, int absence) int mem_bus_absenceConducteur_lire(int numLigne, int numBus) void mem_bus_msg_ecrire(int numLigne, int numBus, char msg[CONST_TAILLE_MSG]) void mem_bus_msg_lire(int numLigne, int numBus, char msg[CONST_TAILLE_MSG])
int numArret int sens int nbPassagers int absenceConducteur char msg[22]	
SAE *Pas de structure pour stocker ce champs :*	void mem_gestionnaire_msgId_ecrire(int msgId) int mem_gestionnaire_msgId_lire()
int gestionnaireMsgId	

- *Utilisation de mutex pour l'accès à la mémoire partagée*

Afin d'éviter des problèmes de lecture ou d'écriture lors de l'accès aux ressources partagées, nous avons introduit l'utilisation de mutex. Ainsi, chacune des méthodes ecrire() et lire() présentée ci-dessus utilise un mutex qui permet de bloquer l'accès à la mémoire pour les autres processus pendant la lecture / l'écriture.

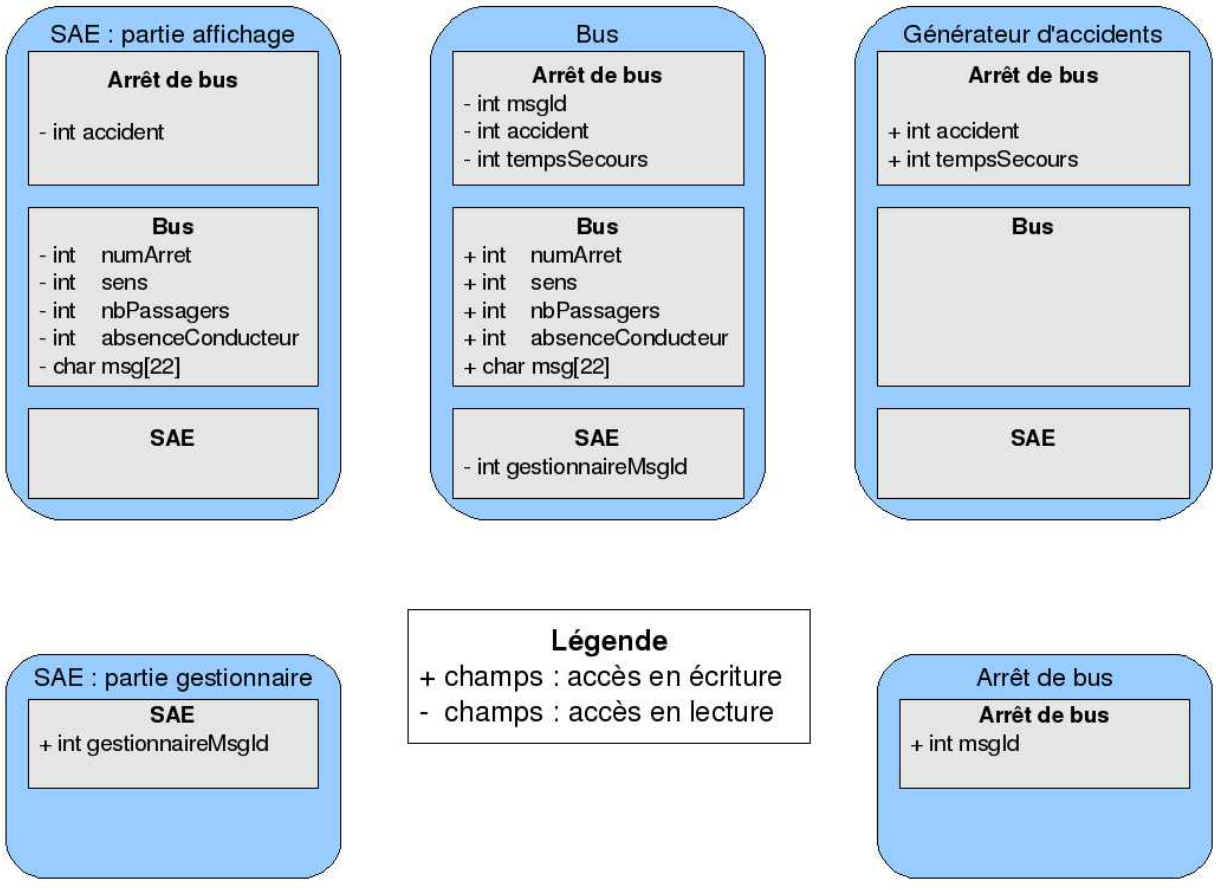
Nous pouvons noter que les mutex utilisés par chacune de ces méthodes sont réalisés à l'aide de sémaphores. La gestion des sémaphores s'effectue à l'aide du fichier Semaphore.c. Chacune des méthodes ecrire() et lire() bloque la ressource en début d'accès et la débloque à la fin à l'aide des fonctions mem_***_acceder_***() et mem_***_liberer_***().

Pour éviter une redondance de sémaphores, les méthodes qui bloquent et débloquent l'accès à la mémoire partagée, bloque la structure de données entière où se trouve le champs désiré. Par exemple, pour accéder au champs sens d'un bus, on va bloquer l'accès à toute la structure contenant les ressources du bus : InfoBus (structure qui contient par exemple numArret, nbPassagers, ...).

Nous avons fait le choix d'utiliser des mutex sur toutes les mémoires partagées dans un soucis d'évolutivité du programme. Ainsi, même si certaines ressources ne nécessitent pas forcément d'être bloquées (les identificateurs des files de messages des arrêts de bus ou du gestionnaire), nous utilisons des mutex pour que l'on puisse ajouter des accès sans craindre l'apparition de problèmes indésirables.

- *Utilisation de cette mémoire partagée*

Voici tout d'abord un schéma résumant les différentes communications instaurées à l'aide de la mémoire partagée. Les champs qui sont accédés en lecture par les processus sont précédés d'un « - », ceux qui sont accédés en écriture sont précédés d'un « + ».



Pour expliquer ce schéma, nous allons mentionner les différents cas où cette mémoire partagée est utilisée (voir schéma présentant les relations entre les entités) :

- Pour l'affichage des informations

La mémoire partagée à laquelle va tenter d'accéder le processus SAE se chargeant de l'affichage est la mémoire partagée des arrêts de bus et celles des bus.

La partie affichage du processus SAE va accéder au champs accident de chaque arrêt de bus pour afficher deux croix « xx » à l'endroit où il y a un accident. La partie affichage va aussi accéder aux champs numArret, nbPassagers, sens, absenceConducteur et msg pour afficher l'état du bus. Notons que le champs msg permet au bus d'afficher son état pour plus de clarté. C'est bien entendu le bus qui se chargera d'écrire son état dans tout ces champs lors de son parcours.

- Pour la transmission du choix de l'accident à tous les bus

Le processus générateur accident après avoir créé un accident va demander au gestionnaire son choix concernant le comportement des bus lorsqu'ils rencontreront l'accident. Pour que son choix soit transmis à tous les bus qui rencontreront l'accident, nous avons choisi d'utiliser la mémoire partagée. Ainsi, le générateur d'accident, après avoir reçu la réponse du gestionnaire concernant l'accident va inscrire dans la structure ArretLigne de l'arrêt concerné la présence d'un accident (accident = 1) et le temps choisi par le gestionnaire (si temps > 0, les bus prendront le détour qui durera temps, si temps = -1, les bus attendront la fin de l'accident).

Une fois l'accident et le temps inscrit dans la structure ArretLigne de l'arrêt concerné, les bus pourront accéder au champs accident, si le champs accident = 1, les bus sauront qu'il y a un accident, ils regarderont ainsi le champs temps qui leur permettra de suivre la décision du gestionnaire.

- Pour la lecture des identificateurs des files de messages.

Afin que les processus qui communiquent par le biais d'une file de messages avec un processus qui en détient une connaissent l'identificateur de la file de messages, nous avons décidé de mettre en place un champs dans la mémoire partagée des arrêts de bus et un champs dans la mémoire partagée du gestionnaire où seront stockés ces identificateurs de files de messages. Ainsi, le gestionnaire et chaque arrêt de bus écriront lors de leur création leurs identifiants de files de messages dans leurs champs msgId. Ainsi, les bus, avant la mise en place de la communication par files de messages avec le SAE et avec les arrêts de bus pourront lire l'identifiant de la file de messages désirée.

3.3. Les points critiques rencontrés

a) Instauration d'un traitant pour le signal SIGINT

Afin que l'on puisse arrêter la simulation simplement, nous avons décider d'instaurer un traitant pour le signal SIGINT. Il suffit donc d'appuyer sur CTRL-C pour arrêter la simulation.

Afin que toute la mémoire utilisée, les files de messages et le tableau de sémaphore soient supprimés correctement, nous avons instauré un traitant sur chaque processus nécessitant lors de son arrêt une suppression quelconque.

Pour ce faire, et comme nous avons dû le mettre en place pour la plupart des processus, nous avons décidé de créer un fichier Signaux.c qui permet rapidement de mettre en place un traitant :

Signaux.c

```
void (*fonctionTraitant)();

void instaurerTraitantSIGINT(void (*_fonctionTraitant)()){
        fonctionTraitant = _fonctionTraitant;
        signal(SIGINT, traitantSIGINT);
}

void traitantSIGINT(int sig)
{
        if (sig == SIGINT){
                (*fonctionTraitant)();
                exit(0);
        }
}
```

Ce code peut être simplement utilisé par les processus en créant une fonction de destruction et en appelant à la création du processus la fonction instaurerTraitantSIGINT. Par exemple, le processus Arrêt de bus a seulement besoin du code suivant :

ArretBus.c

```
instaurerTraitantSIGINT(detruireArretBus);
void detruireArretBus(){
        destructionFileMessageArretBus(msgId);
}
```

Lors de l'appui sur CTRL-C, le signal sera transmis au programme principal ainsi qu'à tous ses fils ce qui assure la destruction de toutes les ressources. Toujours dans un souci d'évolutivité, nous avons choisi de traiter ce signal pour tous les processus ce qui permet lors de l'ajout de nouvelles ressources de n'avoir à rajouter qu'une ligne pour supprimer les nouvelles ressources.

b) Synchronisation entre les processus

Nous avons été confrontés à un problème à la fin de notre projet qui nous a obligé à synchroniser les processus en créant un point de rendez-vous.

Le problème se situait entre les processus bus et SAE. En effet, le programme principal créait d'abord la flotte de bus et ensuite le SAE. Il pouvait dès lors survenir des problèmes dans les deux processus :

- La partie affichage du SAE pouvait tenter d'accéder aux informations du bus présentent dans la mémoire partagée alors qu'elles n'étaient pas correctement initialisées (le message, le sens, le numéro de l'arrêt ne sont initialisés qu'à la création du bus).
- Les bus, si leur conducteur était absents tentait d'accéder dans la mémoire partagée à l'identificateur de file de messages du gestionnaire alors que celle-ci n'avait pas encore été créée par la partie gestionnaire du SAE.

Comment est-ce possible ? Cela vient tout simplement de l'utilisation de la primitive fork qui va créer très rapidement tous les processus alors que l'on pourrait avoir l'impression avec le pseudo-code que les processus sont créés les uns après les autres.

Le problème étant le même si le SAE était créé avant la flotte de bus, nous avons donc été contraints d'instaurer un système de point de rendez-vous à l'aide de sémaphores. L'implantation de ces sémaphores a été présentée brièvement dans le pseudo-code présent au début du rapport.

Rappel du code :

Programme Principal	**BUS**	**SAE**
...		
Créer les arrêts de bus de la ligne 1	Initialiser la mémoire partagée du bus	Créer la file de messages du gestionnaire
Créer les arrêts de bus de la ligne 2		
Enlever un jeton à busCrees		
Enlever un jeton à saeCree	Ajouter un jeton permettant de signaler qu'un bus a été créé	Écrire dans la mémoire partagée l'identificateur de la file de messages
Créer le SAE		
...	Attente que le SAE soit créé	
		Attendre que les bus soient tous créés
	...	
		Informer que le SAE est créé
		...

Nous pouvons donc voir avec ce pseudo-code que le programme principal enlève un jeton à busCrees et saeCree car ils sont initialisés à 1 (Note : on pourrait directement les initialisés à 0).

Le bus, dès sa création va initialiser sa mémoire partagée et va informer qu'il a été créé en ajoutant un jeton à la sémaphore busCrees. Il va ensuite attendre que la file de messages du SAE soit créée pour continuer.

Le SAE va lui créer sa file de messages et va écrire dans la mémoire partagée son identificateur, il va ensuite attendre que tous les bus soient créés et va informer qu'il est créé.

Nous pouvons noter que l'attente et l'information que les processus sont créés se font à l'aide des primitives P et V des sémaphores :

```
BUS
Ajouter un jeton permettant de signaler qu'un bus a été créé :
        V_busCrees();
Attente que le SAE soit créé :
        P_SAECree();
```

```
SAE
Attendre que les bus soient tous créés :
        Pour i de 1 à nombre total de bus Faire
                P_busCrees();
        Fin Pour
Informer que le SAE est créé
        Pour i de 1 à nombre total de bus Faire
                V_SAECree();
        Fin Pour
```

4. Présentation du programme réalisé

4.1. Utilisation du programme

a) Makefile

Nous avons choisi de créer un makefile pour simplifier la compilation du programme. Ainsi, il suffit d'entrer la commande make et le projet se compilera.

b) Bibliothèque ncurses

Nous avons choisi d'utiliser la bibliothèque ncurses qui est une bibliothèque graphique libre fournissant une API pour concevoir des programmes avec une interface utilisateur textuelle, de manière indépendante du terminal.

Cette bibliothèque est à la fois compatible avec Linux et avec Solaris et est très simple à mettre en place. Nous n'en utilisons qu'une infime partie, mais cela nous permet d'avoir une interface lisible.

Sous linux, nous avons été obligé d'installé le paquet libncurses5-dev alors que sous solaris la bibliothèque était déjà installée.

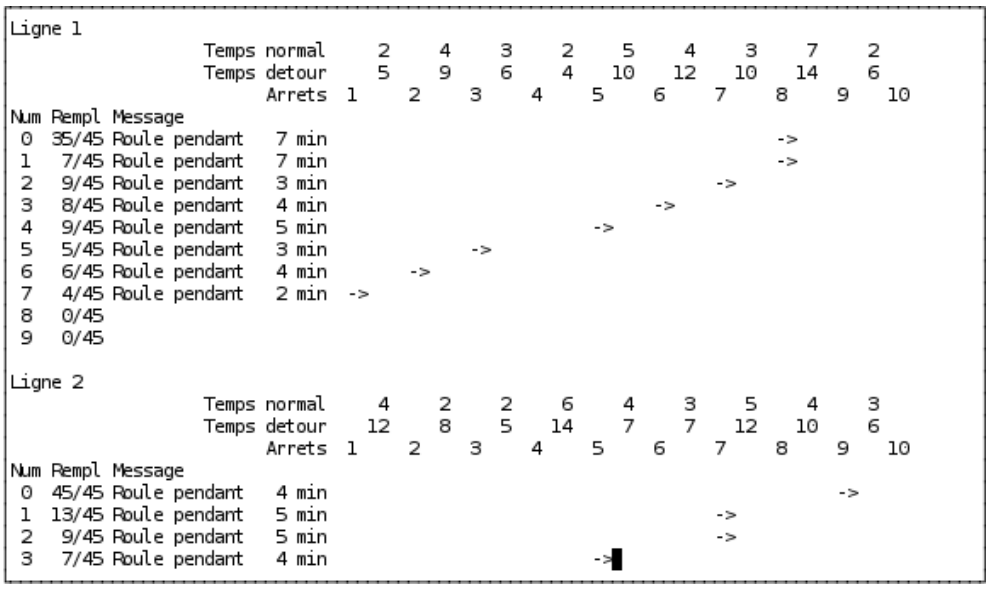
4.2. Présentation

a) Configuration du programme

Il est bien entendu possible de configurer notre programme en modifiant les fichiers Constantes.c et Constantes.h. Cela va permettre de modifier le nombre de bus et d'arrêts de chaque ligne, de modifier la fréquence de génération des différents problèmes (absences, pannes, accidents), de modifier le nombre maximum de passagers que peut prendre un bus. Il sera aussi possible de modifier le temps que doit attendre un bus quand il arrive en bout de ligne, les temps d'attente entre chaque arrêt (temps normal et temps de secours).

b) Aperçu du programme et description des principales composantes

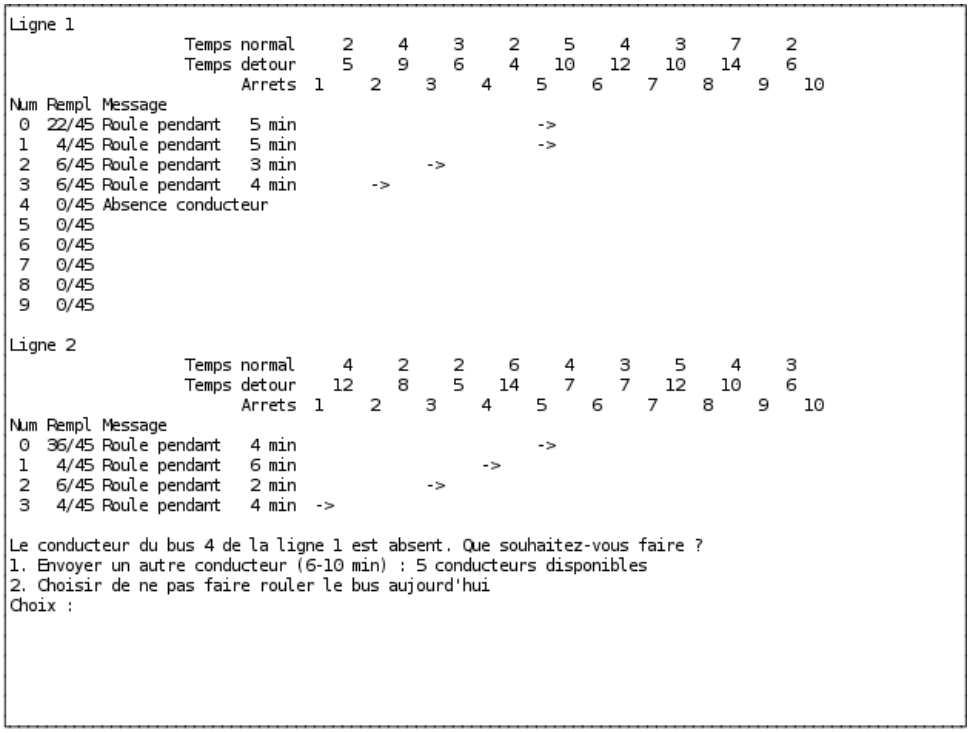
Voici une capture d'écran du programme en fonctionnement :



Nous allons vous présenter les principales fonctionnalités du programme :

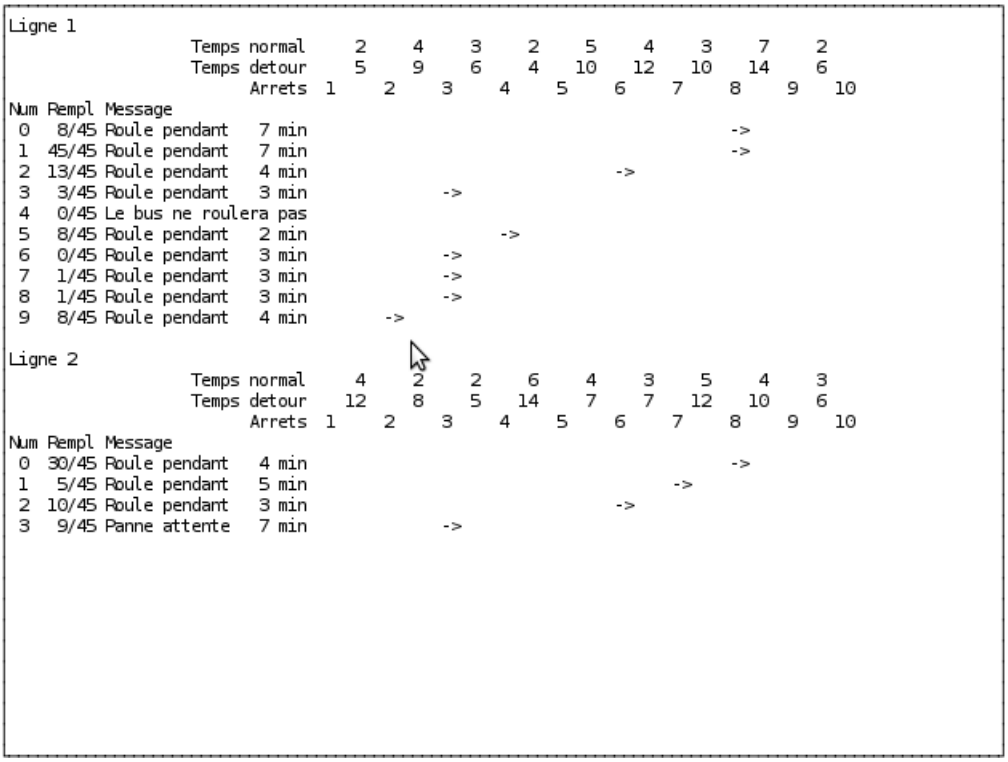
- *Absence d'un conducteur*

Lorsque les bus démarrent, il se peut qu'un conducteur soit absent, on demande alors au gestionnaire son choix :



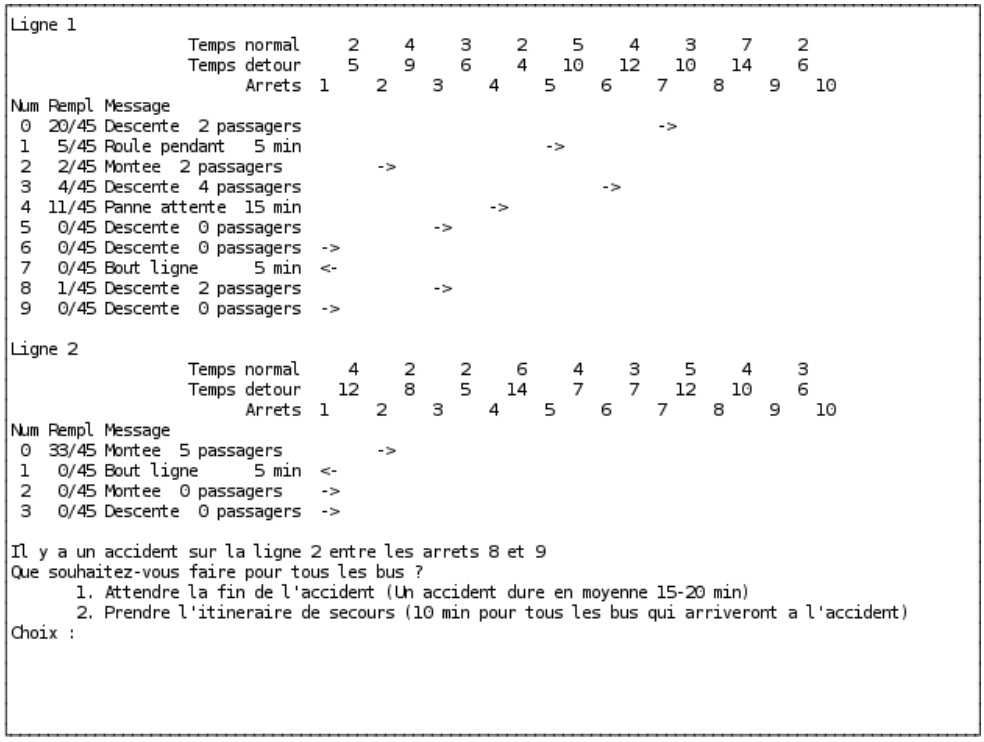
Si il y a suffisamment de conducteurs, il peut envoyer un autre conducteur.

Sinon, il est obligé de ne pas faire rouler le bus :

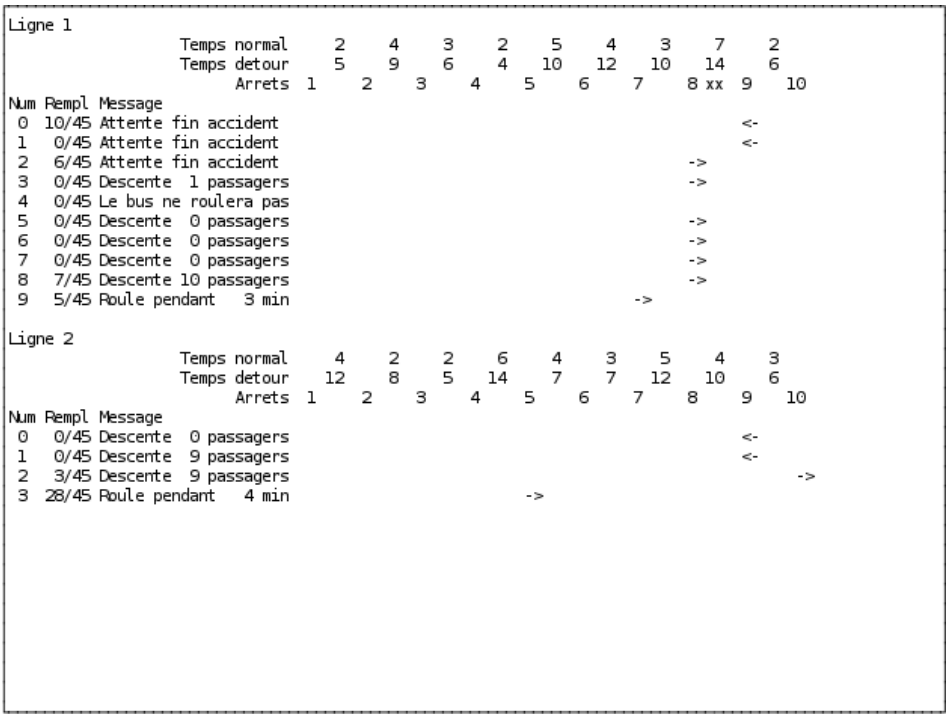


- *Présence d'un accident sur la ligne*

Si il y a un accident sur la ligne, le gestionnaire est averti et il doit prendre une décision :



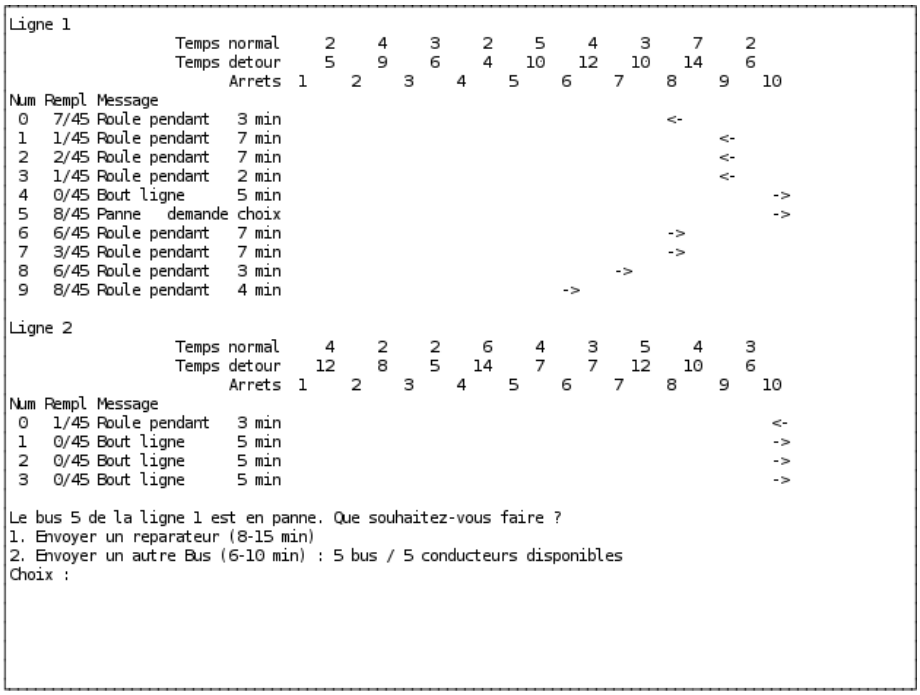
Le gestionnaire va pouvoir commander tous les bus. Si il choisit que les bus doivent attendre la fin de l'accident pour continuer, on aura par exemple (pour la ligne 1 au lieu de la 2) :



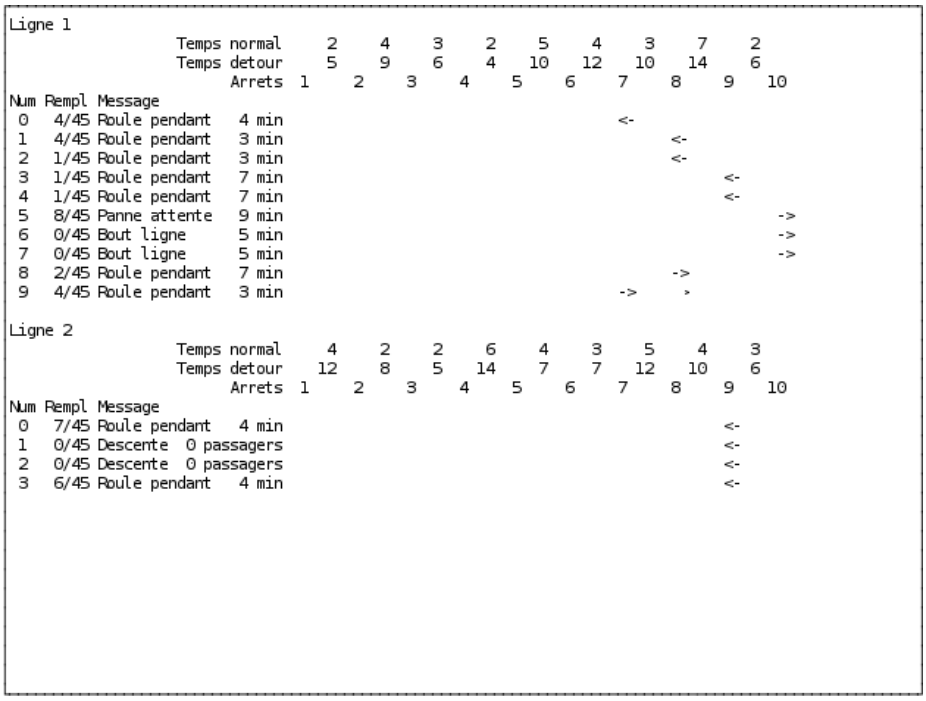
Sinon, les bus prendront le détour si ils sont confrontés à l'accident.

- *Panne de bus*

Nos bus peuvent bien sûr tomber en panne :



Le bus qui est en panne attendra suivant le choix du gestionnaire :



Si il n'y a plus assez de bus ou de conducteurs, le gestionnaire pourra seulement envoyer un réparateur :

```
Le bus 4 de la ligne 1 est en panne. Que souhaitez-vous faire ?
1. Envoyer un reparateur (8-15 min)
.. Plus assez de bus/conducteurs pour envoyer un autre bus : 0 bus / 0 conducteurs disponibles
Choix :
```

5. Conclusion

Pour réaliser ce projet, nous avons choisi avant tout de privilégier la mise en place des connaissances que nous avons acquises ce semestre. Nous n'avons donc pas chercher à créer un réseau de bus qui soit réaliste mais nous avons simplement implantées les bases d'un réseau de bus afin qu'il soit possible d'ajouter des fonctionnalités le rendant réaliste.

6. Sources du programme