Mohamed FISSEL LARAKI
Sofiane TAÎBI

RAPPORT SE

Le projet de système d'exploitation avait pour but de nous familiariser avec les appels Unix/Linux concernant la communication et la synchronisation des processus et des threads. Pour ce projet, il s'agissait de simuler un circuit d'un système de transport en commun, composé d'un autobus, d'un métro et de 3 taxis.

Description : Circuit d´autobus et de métro

L'autobus roule dans une seule direction et parcours 5 stations, depuis la station numéro 0 jusqu'à la station numéro 4. Le métro quant à lui, peut servir seulement 3 stations mais dans les 2 sens, depuis la station 5 jusqu'à la station 7 et ensuite depuis la 7 jusqu'à la 5.

Les stations sont représentées par des files, une file par station. On a donc au total 11 stations (5 pour l'autobus et 6 pour le métro). L'implémentation de la structure file est évidente et on ne va donc pas l'expliquer. On a créé un tableau de 11 files qu'on a appelé **files** et on en a fait une variable globale car celle-ci devrait être accessible depuis tous les threads (Autobus, Métro et Vérificateur). Ainsi l'autobus servira les stations dont les files correspondantes sont **files[0] … files[4]** et le métro servira les stations **files[5] … files[10]**.

Pour mieux comprendre notre représentation des stations :

Stations :

Stations	0	1	2	3	4	5	6	7	7	6	5
Files correspondantes	0	1	2	3	4	5	6	7	10	9	8

Description : Taxis

Les taxis sont beaucoup plus faciles à comprendre puisqu'il s'agit seulement d'un processus fils contenant 3 threads, un thread correspond à un unique taxi. Chaque taxi a pour rôle de récupérer un passager depuis une certaine station et l'emmener à sa destination.

Implémentation

Ce système de transport peut être représenté grâce à des processus et des threads :

Tout d'abord, nous programme contiendra 2 processus, le processus principal et son fils qu'on appellera **taxis**. Ensuite chacun des 2 processus possède 3 threads, le père contient le thread

Mohamed FISSEL LARAKI
Sofiane TAÎBI

autobus, métro et **vérificateur**. Tandis que le processus fils **taxis** contient 3 thread pour le *taxi#1*… *taxi#3*.

I Processus principal

Dans ce processus se situeront nos thread **autobus, métro** et **vérificateur**. Les threads autobus et métro sont très semblables car tous les deux effectuent les mêmes actions, c'est à dire faire descendre les passagers, faire monter les passagers, changer de station et enfin passer le tour au thread vérificateur. Par exemple voici notre fonction du thread autobus :

```
void* autobus_ligne(void* i)
{
  uint32_t station = 0;
  while(!files_vides() || !file_vide(passagers_bus) || !file_vide(passagers_metro))
  {
    sem_wait(&sBus);
    puts("BUS EN MARCHE");
    descendre(passagers_bus, station%(NB_STATIONS_BUS));
    monter_bus(passagers_bus, station%(NB_STATIONS_BUS));

    station++;
    sem_post(&sVerifBus);
  }
  puts("out BUS");
  sem_post(&sVerifBus);
  pthread_exit(NULL);
}
```

Pour le moment ne tenons pas compte des sem_wait() et sem_post() on y viendra plus tard. On voit bien que le code est facile et fait exactement ce qu'on voulait. Le thread tourne tant qu'il y a des passagers qui attendent dans les stations **ou** qu'il y a encore des passagers dans l'autobus **ou** qu'il y a des passagers dans le métro. Effectivement même s'il n y a plus de passagers dans les stations, il faut que l'autobus continue à travailler car il transporte peut être des passagers ou encore que le métro transporte des passagers qui prendront plus tard l'autobus suite à un transfert. Enfin, le modulo permet seulement de rester dans l'intervalle [0;4] même quand notre variable station dépasse 4.

Pour le métro c'est exactement le même principe, seulement on a utilisé 2 boucles for() pour parcourir d'abord depuis la station 5 à la station 7, ensuite de la station 7 à la 5 (de **files[10] à files[8]**).

Voyons maintenant comment marchent les fonctions *monter_bus()* et *descendre() :*

```
void monter_bus(file_t* f, uint32_t station)
{
  passager_t passagerTmp;
  maillon_t* dernier = files[station]->tete;
  while(dernier != NULL && f->taille < CAPACITE_BUS)
  {
    passagerTmp = files[station]->tete->value;
    rem_tete(files[station]);
    add_tete(f,passagerTmp);
    dernier = files[station]->tete;
    printf("BUS : embarque le passager %u à la station %u.\n", passagerTmp.id, station);
    fflush(stdout);
    // On incrémente le profit
     pthread_mutex_lock(&mutex);
    profit += 1;
     pthread_mutex_unlock(&mutex);
  }
}
```

Mohamed FISSEL LARAKI
Sofiane TAÎBI

Quand un autobus s'arrête à une station **x** , tous les passagers qui étaient à cette station veulent monter dans l'autobus. Donc pour faire monter ces passagers, on va simplement retirer un passager de notre file **files[x]** et l'ajouter dans la file du bus. Il faut cependant respecter la limite de la capacité du bus qui est définie plus haut dans notre code grâce à un #define mais il faut aussi faire attention à ce qu'on ne dépasse pas la taille d'une file d'attente d'où le *dernier != NULL*. On remarque que pour retirer un passager depuis une station, on a utilisé rem_tete() au lieu de rem_queue(), et ceci s'explique simplement par le fait que dans une station, le premier arrivé est le premier servi.

```
void descendre(file_t* f, uint32_t station)
{
  if (f->taille>0)
  {
    passager_t passagerTmp;
    maillon_t* dernier = f->tete;
    while(dernier != NULL)
    {
      passagerTmp = dernier->value;
      if (dernier->value.arrivee == station)
      {
        rem_maillon(f, dernier);
        if (station<NB_STATIONS_BUS){
          printf("BUS : debarque le passager %u à la station %u.\n", passagerTmp.id, station);
          fflush(stdout);
        }
        else{
          printf("METRO : debarque le passager %u à la station %u.\n", passagerTmp.id, station);
          fflush(stdout);
        }
      }

      if ((station==0 || station == NB_STATIONS_BUS || station == NB_STATIONS_BUS + NB_STATION_METRO) &&
passagerTmp.transfert )
      { // eviter les erreurs d'entrees
        printf("transfert passager %u \t vers station: %u\n",passagerTmp.transfert,station);
        fflush(stdout);
        transferer(dernier, station,passagerTmp);
      }
      dernier = dernier->suivant;
    }
  }
}
```

La fonction descendre() est légèrement plus compliqué que la fonction monter() car quoique son principe est simple, elle nous a causé beaucoup de problème, car elle contenait beaucoup d'erreurs de segmentation au début, et à chaque fois il a fallu passer par valgrind pour debug notre programme. Elle prend en paramètre une station **x** et une file, soit celle du métro, soit celle de l'autobus, et va la parcourir de la tête à la queue, si un passager est arrivé à destination, c'est à dire que l'arrivée du passager == **x** alors elle le supprime. En plus de supprimer les passagers arrivés à destination, elle va pouvoir transférer ceux qui le souhaitaient. Donc si la station **x** est en fait la station 0 ou (5 ou 8) alors elle va transférer ce passager en fonction de **x** :

```
void transferer(maillon_t* mtmp, uint32_t station_src, passager_t p)
{
  p.transfert = 0;
  if (station_src == 0)
  {
    p.depart = NB_STATIONS_BUS;
    rem_maillon(passagers_bus, mtmp);
    add_queue(files[NB_STATIONS_BUS], p);
  }
  else if( station_src == NB_STATIONS_BUS || station_src == NB_STATIONS_BUS + NB_STATION_METRO )
  {
    p.depart = 0;
```

Mohamed FISSEL LARAKI
Sofiane TAÎBI

```
    rem_maillon(passagers_metro, mtmp);
    add_queue(files[0], p);
  }
  else
  {
    perror("erreur lors de la descente d'un passager");
    exit(0);
  }
}
```

La fonction transférer prend en paramètre un maillon, dont elle aura besoin pour le supprimer, une station pour savoir dans quelle station on se trouve actuellement et finalement du passager. Elle va d'abord changer la variable transfert du passager de 1 à 0. Ensuite en fonction de la station elle transférera à la bonne station.

Maintenant regardons ce que fait le thread Vérificateur :

Le thread vérificateur va parcourir toutes les files d'attentes, **files[0] … files[10],** et incrémente le temps d'attente de chaque passager. Si le temps d'attente d'un passager atteint le temps max de ce même passager, il va alors l'envoyer au processus **taxis** à travers un tube nommé créé plus tôt dans le main.

```
void* verificateur_maj(void* i)
{
  int entreeTube;
  maillon_t *avant_temp, *temp;
  passager_t passagerTmp;
  char s[TAILLE];
  if((entreeTube = open(nomTube, O_WRONLY)) == -1)
        {
                perror("Impossible d'ouvrir l'entrée du tube nommé.");
                exit(0);
        }
  while(!files_vides(files) || !file_vide(passagers_bus) || !file_vide(passagers_metro) )
  {
    sem_wait(&sVerifMetro);
    sem_wait(&sVerifBus);

    puts("VERIFICATION EN MARCHE");


    for (int i=0; i<NB_FILES; i++)
    {
      for (temp=files[i]->tete; temp!=NULL; temp=temp->suivant)
        {
          temp->value.tps_attente++;
          if (temp->value.tps_attente == temp->value.tps_max)
          {
            passagerTmp = temp->value;

            //On le supprime
            rem_maillon(files[i], temp);
            //On le transfere
            sprintf(s,"%d %d %d %ld %d %ld\
n",passagerTmp.id,passagerTmp.depart,passagerTmp.arrivee,passagerTmp.tps_attente,passagerTmp.transfert,passagerTmp
.tps_max );
            write(entreeTube,s,TAILLE);
            printf("verificateur : transfert du passager %d vers le taxi\n",passagerTmp.id);
            fflush(stdout);
            // On incremente le profit de 3
             pthread_mutex_lock(&mutex);
            profit += 3;
             pthread_mutex_unlock(&mutex);
          }
        }
    }
    sem_post(&sBus);
    sem_post(&sMetro);
  }
  sem_post(&sBus);
  sem_post(&sMetro);
  for (uint32_t i = 0 ; i < NB_TAXIS ; i++ ){
```

```
    write(entreeTube,"1",20);
  }
  pthread_exit(NULL);
}
```

Le while() permet de faire en sorte que le vérificateur ne s'arrête pas tant que tous les passagers ne sont pas arrivés à destination. Ensuite le premier for() permet de parcourir notre tableau **files** et le deuxième permet de parcourir la file **file[i]**. Si on trouve un passager dont le temps d'attente est égal au temps d'attente maximal, on va remplir un buffer **s** avec les infos du passager grâce à un sprintf() et enfin on l'envoie dans notre tube nommé grâce à un write().

II Processus Taxis

Le processus **taxis** ne contient qu'une seule fonctions taxi_ligne() qui sera associée aux 3 threads taxi:

```
void* taxi_ligne(void* i)
{
  int taille;
  int id_taxi = (int) i;
  int sortieTube        ;
  char r[TAILLE];
  passager_t p;
  if((sortieTube = open (nomTube, O_RDONLY)) == -1)
        {
                perror("Impossible d'ouvrir la sortie du tube nommé.");
                exit(EXIT_FAILURE);
        }
  while(1)
  {
    read(sortieTube,r,TAILLE);
    if (strlen(r) == 1)
      break;
    sscanf(r,"%d %d %d %ld %d %ld\n",&p.id,&p.depart,&p.arrivee,&p.tps_attente,&p.transfert,&p.tps_max);
    //p = atop(r,TAILLE);
    usleep(MAX_ATTENTE);
    printf("taxi#%d : passager %d est rendu a la station %d\n",id_taxi,p.id,p.arrivee);
    fflush(stdout);
  }
  pthread_exit(NULL);
}
```

cette fonction va tourner à l'infini et reste bloquée en écoutant à la sortie du tube nommé jusqu'à recevoir un passager envoyé par le vérificateur. Ensuite une fois cette chaîne r reçue on la transforme en type passager_t grâce à la fonction sscanf(). Ensuite va s'arrêter un certain temps avant d'afficher le message. Ce thread ne tourne en réalité pas à l'infini car si on regarde le thread **vérificateur**, avant de quitter, ce dernier envoie '1'. Donc quand le thread taxi reçoit une chaîne dont la longueur est 1, il s'arrête.

III Remarques

Dans cette partie, on va survoler les variables importantes dans notre programme.

```
/*FILES*/
file_t* files[NB_FILES];
file_t* passagers_bus;
file_t* passagers_metro;
```

Mohamed FISSEL LARAKI
Sofiane TAÎBI

```
/*SEMAPHORE*/
sem_t sVerifMetro, sVerifBus, sMetro,sBus;

/*NOM TUBE*/
char nomTube[20] = "tube.fifo";

/*VARIABLES*/
int profit = 0;


/*MUTEX*/
pthread_mutex_t mutex ;
```

Pour ce qui est des files, on a déjà tout expliqué. Un buffer nomTube[20] qui va seulement stocker le nom du tube nommé, il est en variable globale, pour que les 2 processus puissent y avoir accès. Enfin, notre variable profit, qui compte le profit effectué par le métro, l'autobus et les taxis. Elle aussi est globale car les 3 threads du processus principal devront pouvoir la modifier. Cette variable s'incrémente de 1 à chaque fois que le métro/bus fait monter un passager et de 3 avant qu'un vérificateur envoi un passager au taxi, ceci permet de compter le nombre de passagers pris par un taxi.

IV Questions

1)

On a 4 sémaphores qui nous permettent d'effectuer la communication bilatéral entre les threads du processus principal. La variable **files**, quoique accessible par les 3 threads, n'est pas en danger. Le thread **vérificateur** ne tourne pas en même temps que les threads **métro** et **autobus** grâce à la communication bilatéral. Ensuite le thread **métro** et **autobus**, même s'ils tournent en même temps, n'accèdent pas aux même stations donc pas aux même files.

Les sémaphore sBus et sMetro permettent de laisser s'exécuter respectivement les thread **autobus** et **métro**. Tandis que les sémaphore sVerifMetro et sVerifBus permettent au **vérificateur** de s'éxecuter. Ainsi, au début sBus et sMetro sont initialisés à 1 tandis que sVerifMetro et sVerifBus sont à 0. De cette façon, on commence par exécuter les thread **autobus** et **métro** qui vont à la fin incrémenter les sémaphores du **vérificateur**, ensuite se dernier va à la fin de son exécution incrémenter les sémaphores du **autobus** et **métro** et ainsi de suite.

On a utilisé un mutex, pour sécuriser la variable profit qui elle peut être incrémenter en même temps par le métro et le bus.

2) Pour le cas de plusieurs bus circulant dans le même sens, il aurait fallut faire attention à l'utilisation indispensable d'un mutex car les bus ne devraient pas lire dans les files des stations en même temps.

V Problèmes rencontrés

Mohamed FISSEL LARAKI
Sofiane TAÎBI

Lors de l'implémentation de ce système en langage C, nous avons rencontrés pas mal de difficultés tout au long de la programmation. A chaque fois qu'on résolvait un problème, un autre survenait, et c'était à chaque fois très difficile de repérer la source de l'erreur même avec l'aide du debugger valgrind.

Au final la plupart des erreurs été causées par de notre structure file_t qui était mal implémentée au niveau de certaines fonctions telle que rem_maillon(maillon_t*), rem_tete() …

Un autre problème auquel nous avons été confronté été de trouver une manière d'incrémenter notre variable profit lorsqu'un passager prend un taxi. En effet, les taxis sont dans un processus à part, et donc modifier la variable *profit* depuis le processus taxis n'affectait pas la variable pour le processus principal. Mais au final nous avons trouvé une solution très simple qui était d'incrémenter le *profit* depuis le thread **vérificateur**.

Nous avons aussi eu affaire pour la première fois dans toute notre expérience avec le langage C à cette erreur : * *stack smashing detected *: <unknown> terminated*. D'après internet, il s'agissait d'une erreur due à un dépassement de la capacité du buffer. Ce qui s'est avéré vraie car notre fonction atop (array to passager) qui permettait de transformer une chaîne '# 103 3 4 0 1 30' en passager_t avait été mal implémenté. C'est pour cela qu'on a arrêté de l'utiliser et on a eu recours à la fonction sscanf().

CONCLUSION

Ce projet fut un excellent moyen pour nous de non seulement approfondir nos connaissance mais aussi de les mettre en pratique. On se sent tous les deux beaucoup plus à l'aise avec les notions vues en cours. Le projet fut un réel plaisir pour nous et on est très fier de l'avoir fini. Cependant on pense qu'on pourrait encore améliorer notre programme pour le rendre plus rapide. Par exemple notre fonction rem_queue() va parcourir toute la liste de la tête à la queue pour retirer la queue car notre liste est simplement chaînée. On pourrait aussi réviser nos allocations et nos free() car d'après valgrind on n'a pas free tout l'espace mémoire alloué.