UNIVERSITÉ TOULOUSE III PAUL SABATIER

**Licence 3 Informatique**

Université Toulouse III – Paul Sabatier
118 route de Narbonne
31062 Toulouse cedex 9

# Programmation Système
## Contrôle terminal – Session 1 – 17 décembre 2018

**Durée : 1h30** **Aucun document autorisé**

**Attention : On veillera à la présentation et aux commentaires**

## Partie 1 : Processus UNIX

On se propose de simuler le déplacement d'un véhicule autonome de niveau 4 c'est-à-dire où le conducteur peut se détendre (mode « détente ») lorsque le trafic est « à l'arrêt » ou « fluide » et doit être en alerte (mode « conduite »), prêt à reprendre le contrôle, lorsque le trafic devient « chargé ».

L'objectif de la simulation est de faire se déplacer le véhicule entre une position initiale et une position finale. Les positions du véhicule sont enregistrées dans un GPS intégré au véhicule et accessibles via les trois fonctions fournies à la fin du sujet. Ces positions sont exprimées grâce au type *Position* ainsi défini : *typedef struct { int x ; int y ; } Position* ;

La simulation est réalisée à l'aide de trois **processus cycliques** s'exécutant en parallèle :

- **Conducteur** : Le conducteur commence par enregistrer la position de sa destination dans le GPS intégré avant de faire démarrer le véhicule. Par la suite, à tout instant, ce processus se trouve en mode « détente » ou « conduite » selon les indications envoyées par le processus *Commande*.
- **Trajectoire** : Ce processus a pour rôle d'estimer périodiquement l'état du trafic à la position courante du véhicule, d'en déduire la prochaine position que doit atteindre la voiture, et d'en informer le processus *Commande*.
- **Commande** : Ce processus est chargé de piloter la voiture pour l'amener à la position spécifiée par le processus *Trajectoire*, tout en avertissant le processus *Conducteur* lorsqu'il doit passer en mode « conduite » ou « détente ». Lorsque le véhicule est arrivé à destination, *Commande* en avertit le *Conducteur* qui peut alors descendre du véhicule.

La simulation doit **se terminer** lorsque le véhicule est arrivé à destination.

### Fonctions fournies (que vous devez utiliser mais ne devez pas écrire) :

*int obtenirEtatTrafic(Position p)* : retourne une valeur qualifiant le trafic à la position p.

*void calculerNewPosition(int etatTrafic, Position *p)* : retourne la position à atteindre en fonction d'une position p et d'un état du trafic.

*void piloter(Position p)* : amène la voiture à la position p.

*void setDestination(Position p)* : permet d'enregistrer la position p dans le GPS.

*Position getPositionCourante(void)* et *Position getDestination(void)* : permettent d'obtenir, respectivement, la position courante et la destination du véhicule en interrogeant le GPS.



*void detente(void)* et *void conduite(void)* : simulent l'état « détendu » ou « conduite » du conducteur.

### Questions

1. Faire une **description schématique claire, précise** et **commentée** de la solution proposée qui mettra en œuvre des **processus** Unix et utilisera **signaux** et **tubes** de communication.
2. Écrire le **code C** des différents **processus Unix** impliqués : *Conducteur*, *Trajectoire* et *Commande*.
3. Écrire le code C de la fonction **main** qui initialise l'application.

## Partie 2 : Synchronisation de processus par sémaphores de Dijkstra

Dans un club de parapente, les parapentistes partagent les NBA ailes disponibles et des navettes de transports.

On suppose qu'un parapentiste est simulé par un processus ayant le comportement suivant :

| **Parapentiste ()** |
|---|
| DemanderAile() |
| Voler() *// En toute indépendance des autres parapentistes* |
| RemonterAvecNavette() |
| RestituerAile() |

On veut synchroniser les accès des différents parapentistes aux ressources partagées disponibles, ce qui revient à écrire **DemanderAile**(), **RemonterAvecNavette**() et **RestituerAile**(). En effet, Voler() se fait de manière indépendante pour chaque parapentiste et ne présente pas de conflits d'accès.

### Version 1

On suppose qu'il existe NBNT navettes individuelles et qu'il suffit d'en avoir une pour pouvoir remonter restituer l'aile.

### Version 2

On suppose maintenant qu'il n'existe qu'**une seule** navette de NBP places existe et que toutes les places de cette navette doivent être occupées avant qu'elle ne remonte.

**Pour chaque version**, on demande d'**écrire les opérations demandées** en utilisant des sémaphores de **Dijkstra** pour assurer la synchronisation.

***Attention*** : Vous écrirez du **pseudo-code, comme en Cours-TD**.

## Quelques rappels UNIX pour la partie 1

**/* On suppose que les #include seront faits de la bonne manière, inutile de les préciser dans le code de la partie 1 */**

```
int main(int argc, char **argv, char **env);
```

```
pid_t getpid (void);
pid_t getppid (void);
uid_t getuid (void);
uid_t geteuid (void);
```

```
pid_t fork (void);

void exit (int compteRendu);
pid_t wait (int *circonstances);
pid_t waitpid (pid_t pidAttendu, int *circonstances, int options);
          /* options: WIFEXITED, WIFSIGNALED, WEXITSTATUS, WTERMSIG */
```

```
int creat (char *cheminAcces, mode_t droits);
int open (char *cheminAcces, int mode, mode_t droits);
          /* mode:   O_RDONLY, O_WRONLY, O_RDWR,
                     O_APPEND, O_CREAT, O_EXCL, O_TRUNC */

int write (int numeroInterne, char *adresse, unsigned nombreTransmis);
int read (int numeroInterne, char *adresse, unsigned nombreDemande);
int close(int numeroInterne);

int dup (int numeroInterne);
int dup2 (int origine, int destination);

int pipe(int tube[2]);
```

```
int sigemptyset (sigset_t *ensSignaux);
int sigaddset (sigset_t *ensSignaux, int unSignal);
int sigdelset (sigset_t *ensSignaux, int unSignal);
int sigfillset (sigset_t *ensSignaux);
int sigismember (sigset_t *ensSignaux, int unSignal);

int sigprocmask (int actionSouhaitee,
                 sigset_t *ensSignaux,
                 sigset_t *ancienEnsSignaux);
          /* actionSouhaitee: SIG_SETMASK, SIG_BLOCK, SIG_UNBLOCK */

int sigpending (sigset_t *ensSignaux);
```



```
typedef void (*traitement) (int leSignal);
traitement signal(int sigIntercepte, traitement monTraitement);
struct sigaction /* prédéfinie */ {
     traitement sa_handler; /* SIG_IGN, SIG_DFL, fonction */
     sigset_t   sa_mask;
     int        sa_flags;
};
int sigaction (int sigIntercepte,
               struct sigaction *nouvelleAction,
               struct sigaction *ancienneAction);

int pause();
int sigsuspend (sigset_t *ensSignaux);

int kill(int pidDestinataire, int sigEmis);

unsigned alarm(unsigned duree);
```