Conseils de développement : pour chaque exercice, créez un répertoire en lui donnant le nom de l'exercice.
Veillez à tester systématiquement les valeurs retournées par les appels système et à gérer les erreurs afférentes.

Exercices (chapitre I, II et III)

Exercice ARGENV :

Récupération des arguments et de l'environnement d'un processus.

Fichier : **argenv.c** (répertoire **argenv**)

Afficher les arguments de la ligne de commande, ainsi que l'environnement d'un processus, reçus comme paramètres de la fonction principale *main*.

Exercice PIDUID :

Récupération des informations d'identification attachées à un processus.

Fichier : **piduid.c** (répertoire **piduid**)

Afficher toutes les informations disponibles du type *process identification* : PID, PPID, UID, GID, EUID, EGID, etc.
Dans ce programme, vous mettrez en place une gestion systématique des erreurs de programmation système, sous forme d'un sous-programme **erreur,** affichant les messages sur la sortie standard erreur (*file descriptor* 2). Vous pourrez le reprendre ensuite dans les autres exercices en le complétant au fur et à mesure des besoins.

Exercice GETPUT :

Gestion de l'environnement d'un processus.

Fichier : **getput.c** (répertoire **getput**)

Dans ce processus, à l'aide d'appels système, vous afficherez le contenu de la variable d'environnement PATH puis vous la modifierez en lui rajoutant le répertoire de vos exercices; réaffichez ensuite le contenu de PATH pour vérifiez la modification. Vous définirez aussi une variable TELECOM dans laquelle vous introduirez le texte "Institut Galilee" que vous récupérerez ensuite l'afficher également. Vérifiez que ces modifications de l'environnement disparaissent quand le processus est terminé.

Exercices (chapitres IV)

Exercice COPIE :

Copie de fichiers, sans écrasement de la destination.

Fichier : **copie.c** (répertoire **copie**)

Uniquement à l'aide d'appels système, écrire un programme de copie de deux fichiers, sans écraser la destination si elle existe, et dans lequel les noms des fichiers source et destination sont passés sur la ligne de commande.

Exercice NEWCAT :

Utilitaire d'affichage du contenu d'un fichier.

Fichier : **newcat.c** (répertoire **newcat**)

L'idée de ce programme est de réaliser une version simplifiée de la commande *cat*.

La structure générale de ce programme, s'appuyant uniquement sur des appels système, est de lire une ligne au clavier (entrée standard, *file descriptor* 0) et de la ressortir aussitôt à l'écran (sortie standard, *file descriptor* 1). S'il s'agit d'opérer avec des fichiers en entrée et/ou en sortie, il suffit d'utiliser les symboles de re-direction du système < et >.

Toutefois, comme la commande *cat*, si un nom de fichier est fourni comme argument sur la ligne de commande, alors la lecture des données ne doit plus s'opérer sur l'entrée standard 0, mais dans le fichier qui aura été préalablement ouvert (ne pas oublier de le refermer avant la fin de programme).

La principale différence avec la commande *cat* est que *cat* peut recevoir en argument plusieurs noms de fichiers et que cette commande opère alors une concaténation en sortie de tous les fichiers fournit en argument.

`newcat`	(lecture au clavier, sortie à l'écran)
`newcat < fich1`	(lecture depuis *fich1*, sortie à l'écran)
`newcat > fich2`	(lecture au clavier, sortie dans *fich2*)
`newcat < fich1 > fich2`	(lecture depuis *fich1*, sortie dans *fich2*)
`redir fich1`	(lecture depuis *fich1*, sortie à l'écran)
`redir fich1 > fich2`	(lecture depuis *fich1*, sortie dans *fich2*)

Rappel : une fin de données au clavier se code Ctrl-D.

Exercice REDIR :

Re-direction de la sortie standard.

Fichier : **redir.c** (répertoire **redir**)

Le fonctionnement de ce programme ressemble un peu à celui appelé **newcat** : tout ce qui est reçu sur l'entrée standard est aussitôt afficher, ligne après ligne, sur la sortie standard. Mais ce programme doit également être susceptible de récupérer, en argument sur ligne de commande, un nom de fichier qui servira alors de re-direction pour la sortie standard.

`redir`	(lecture au clavier, sortie à l'écran)
`redir < fich1`	(lecture redirigée par le système depuis *fich1*, sortie à l'écran)
`redir fich2`	(lecture au clavier, sortie par re-direction dans *fich2*)
`redir fich2 < fich1`	(lecture redirigée par le système depuis *fich1*, sortie par re-direction dans *fich2*)

Dans les exemples de fonctionnement ci-dessus, le re-direction d'entrée est assurée automatiquement par le système et vous n'avez pas en vous en occuper dans votre programme. En revanche, la re-direction de sortie, provoquée par la présence d'un nom de fichier en argument sur ligne de commande, relève entièrement de votre programmation : vous devrez réaliser cette opération à l'aide de l'un des deux appels système *dup* ou *dup2*.

Exercice FICH_MAP :

Utilisation d'un fichier "mappé" en mémoire.

Fichier : **fichmap.c** (répertoire **fichmap**)

Ecrire un programme permettant de lire un fichier nommé *fichmap.txt,* en le "mappant" en mémoire, afin d'afficher son contenu à l'écran (à vous de créer le fichier *fichmap.txt* et de placer un texte dedans).

Exercice COPIEMAP :

Programme de copie de fichiers en les "mappant" en mémoire, sans écraser la destination.

Fichier : **copiemap.c** (répertoire **copiemap**)

Dans son principe, ce programme est identique à celui appelé **copie**. La seule différence est que les deux fichiers source et destination doivent être manipulés sous forme "mappés" en mémoire.

Exercices (V)

Exercice KILLPROC :

Destruction d'un processus.

Fichier : **killproc.c** (répertoire **killproc**)

Ecrire un programme pour envoyer le signal de terminaison SIGTERM (15) à un processus dont le PID est passé en argument sur ligne de commande. Vous afficherez un compte-rendu de succès ou d'échec de l'opération.

Exercice BOUCLE :

Programme bouclant à l'infini.

Fichier : **boucle.c** (répertoire **boucle**)

Ecrire un programme qui boucle à l'infini en affichant toutes les deux secondes un message contenant son PID. Ce programme doit être insensible à Clrl-C et Ctrl-\ (signaux 2 et 3). Il faudra l'arrêter à l'aide du programme **killproc**.

Exercice CTRL-C :

Gestion de signaux.

Fichier : **ctrl-c.c** (répertoire **ctrl-c**)

Sur le modèle du programme **boucle**, écrire un programme qui boucle à l'infini en affichant un message toutes les secondes avec indication de son PID.

Ce programme devra gérer Ctrl-C (signal 2) de la façon suivante : au premier signal Ctrl-C reçu, vous afficherez un message indiquant l'interception du signal, mais vous devrez modifier la gestion des signaux pour qu'au deuxième Ctrl-C reçu, ce soit l'action "standard" prévue pour Ctrl-C qui s'exécute, c'est-à-dire l'arrêt du programme (restauration de l'action standard).

En outre, ce programme devra être insensible au signal de terminaison SIGTERM (15), mais quand vous lui enverrez l'un des signaux SIGUSR1 ou SIGUSR2, à l'aide de la commande **kill**, il devra afficher un message de compte-rendu de réception de ce type de message.

Exercice TIMER :

Gestion du signal SIGALRM pour constituer un "timer".

Fichier : **timer.c** (répertoire **timer**)

Le programme **timer** affiche en permanence un message toute les secondes (boucle infinie). Mais toute les trois secondes, provenant de lui-même, il doit recevoir un signal SIGALRM (14) qui provoque l'apparition d'un autre message.

Pour y parvenir, la fonction de gestion de l'arrivée d'un signal SIGALRM doit de nouveau réarmer le déclenchement d'un signal SIGALRM dans un délai de trois secondes.

Vous arrêterez ce programme à l'aide du programme **killproc**.

Exercice THRTXT :

Mise en oeuvre d'une unité d'exécution (thread).

Fichier : **thrtxt.c** (répertoire **thrtxt**)

Le but de cet exercice est de mettre en oeuvre une fonction sous forme d'une unité d'exécution, afin qu'elle se déroule en partage avec la fonction principale.

Fonction principale : la fonction principale de ce programme lance, sous forme d'une unité d'exécution, un sous-programme *afficher* en lui passant en argument un message à afficher. Ensuite cette fonction principale affiche 5 fois un message avec une pause d'une seconde entre chaque message. A la sortie de cette boucle, elle attend alors la terminaison du sous-programme *afficher* et fournit un compte-rendu de terminaison avec le code de retour de cette unité d'exécution.

Fonction *afficher* : cette fonction est lancée sous forme d'une unité d'exécution (*thread*) par la fonction principale du programme. Elle reçoit en argument un message qu'elle doit afficher 10 fois avec également une pause d'une seconde entre chacun de ces affichages. A la sortie de cette boucle, cette unité d'exécution se termine en envoyant un code de retour de zéro qui sera récupéré par la fonction principale pour l'afficher.

Exercice EXECUTE :

Enchaînement de processus.

Fichiers : **execlist.c**, **execvect.c** et **argpid.c** (répertoire **execute**)

Chacun de deux premiers programmes doit lancer le troisième programme **argpid** à l'aide d'un appel système de type *exec* (enchaînement de processus avec auto-destruction du processus en cours).

Chacun de ces deux lancements devra se faire avec simulation de passage de trois arguments sur ligne de commande. Le premier programme **execlist** devra utiliser l'appel *execlp* et le deuxième **execvect** l'appel *execvp*. Dans chacun de ces deux programmes, vous afficherez leur PID, avant l'appel de type *exec*.

Le programme **argpid** est le même que le programme **argenv**, mais auquel vous rajouterez comme premier affichage, celui de son PID.

Pour chaque lancement, vous comparerez les deux PID affichés afin de constater qu'ils sont identiques.

Exercice FOURCHE :

Engendrement d'un nouveau processus par clonage.

Fichier : **fourche.c** (répertoire **fourche**)

Recopier et exécuter l'exemple ci-dessous. Analysez les résultats affichés.

```
  #include <sys/types.h>
  #include <unistd.h>
  #include <stdio.h>

int main(void)
{
  pid_t pid;

  printf("*** DEBUT *** (PID:%d PPID:%d)\n",getpid(),getppid());

  pid=fork();
  if (pid > 0)
      printf("Je suis le PERE (PID:%d PPID:%d)\n",getpid(),getppid());
    else if (pid == 0)
      printf("Je suis le FILS (PID:%d PPID:%d)\n",getpid(),getppid());
    else
      perror(" impossible de lancer un nouveau processus");

  printf("*** FIN *** (PID:%d PPID:%d)\n",getpid(),getppid());

  return 0;
}
```

Exercice PEREFILS :

Lancement d'un processus à partir d'un autre processus.

Fichiers : **procpere.c** et **procfils.c** (répertoire **perefils**)

Le programme **procpere** lance le programme **procfils** en lui passant en argument le *file descriptor* d'un fichier *perefils.txt* qu'il aura préalablement ouvert.

Dans le programme **procpere**, vous commencerez par afficher un message comportant son PID. Ensuite, après avoir ouvert en mode *low-level* le fichier *perefils.tx,t* (avec écrasement si nécessaire), vous écrirez dedans le même message indiquant qu'il s'agit du processus père et en mentionnant son PID. Puis au moment du lancement du fils, vous lui transmettrez le *file descriptor* de ce fichier qui sera récupérez comme argument par le processus **procfils**. Après le lancement du fils, le processus père devra fermer le *file descriptor* du fichier qu'il aura ouvert et se mettra en attente de la terminaison de son fils. Il affichera un compte-rendu de la terminaison de ce fils avec indication du code de retour, puis s'arrêtera.

Dans le fils **procfils**, après récupération du descripteur du fichier *perefils.txt* ouvert dans le père, le fils écrira à son tour un message dedans sur le même modèle que le père, en mentionnant son propre PID et celui de son père, puis il fermera ce fichier. Avant de se terminer, il affichera également à l'écran les mêmes informations que celles écrites dans le fichier, puis se mettra en sommeil pendant 3 secondes. Il se terminera avec un code de retour zéro.

Exercice NEWSHELL :

Réalisation d'un petit interpréteur de commandes.

Fichier : **newshell.c** (répertoire **newshell**)

Cet exercice a pour but de réaliser un "shell" simplifié, construit autour de l'ossature suivante (rien ne s'oppose à ce que vous le rendiez plus complexe) :

.

```
afficher le "prompt"
lire une ligne de commande
TANT-QUE non fin (fin de fichier ou commande exit)
  séparer les arguments de la ligne de commande
  cloner le processus shell
  dans le fils, exécuter la commande avec ses arguments
  dans le shell, attendre la terminaison du fils
  afficher le "prompt"
  lire une ligne de commande
FIN-TANT-QUE
```

Exercices (chapitre VI)

Exercice PIPE :

Utilisation d'un canal de communication non nommé entre deux processus.

Fichiers : **pipe.c**, **piperead.c** et **pipewrit.c** (répertoire **pipe**)

Programme **pipe** : ce programme crée un canal de communication non nommé, puis lance deux processus **piperead** et **pipewrit** qui reçoivent respectivement, le *file descriptor* de lecture du "*pipe*" et le *file descriptor* d'écriture du "*pipe*". Juste après les clonages préalables au lancement respectif de chacun des deux programme fils, il est opportun de fermer les *file descriptor* de "pipe" inutiles : *file descriptor* de lecture pour le programme **pipewrit**, *file descriptor* d'écriture pour le programme **piperead**. Une fois les deux programmes fils lancés, le programme **pipe** attend la terminaison des deux processus avant lui-même de s'arrêter.

Programme **pipewrit** : ce programme reçoit le *file descriptor* d'écriture du "*pipe*" et renvoie tout ce qu'il lit au clavier dans le canal de communication non nommé via ce *file descriptor* d'écriture. Il s'arrête quand l'entrée au clavier est terminée.

Programme **piperead** : ce programme reçoit le *file descriptor* de lecture du "*pipe*" et lit, via ce *file descriptor* de lecture, tout ce qui est envoyé dans le canal de communication non nommé pour l'afficher à l'écran. Il s'arrête quand les envois dans le canal de communication sont terminés.

Exercice PIPECAT :

Canal de communication non nommé entre deux processus, via des re-directions d'entrée/sortie.

Fichiers : **pipecat.c** et **newcat.c** (répertoire **pipecat**)

Programme **pipecat** : ce programme, qui ressemble au précédent, crée un canal de communication non nommé, puis lance deux processus avec chaque fois le même programme **newcat** dedans. L'idée de cet exercice et de rediriger, d'un côté la sortie standard 1 dans le *file descriptor* d'écriture du "pipe", et de l'autre l'entrée standard 0 dans le *file descriptor* de lecture du "pipe". De cette façon, le premier processus **newcat** s'occupera de la saisie au clavier pour la rediriger dans le "pipe", et le second processus **newcat** récupèrera les données dans le "pipe" pour les afficher à l'écran, et ceci sans avoir à modifier quoi que ce soit dans le programme **newcat** qui se contente de lire sur l'entrée standard 0 et d'écrire sur la sortie standard 1.

Programme **newcat** : c'est le programme **newcat** qui a déjà été écrit.

Exercice PIPENOM :

Mise en œuvre d'un échange entre un client et un serveur, via un canal de communication nommé.

Fichiers : **pipinoms.c** et **pipenomc.c** (répertoire **pipenom**)

Le principe de cet exercice est de permettre l'échange de messages entre un processus client et un processus serveur. Les deux processus sont lancés sans filiation dans deux fenêtres différentes, mais sur la même machine. Les canaux de communications nommés ne permettant pas de gérer des échanges sous forme de messages "structurés" mais uniquement sous forme de flots d'octets, l'idée est de reconstituer des échanges sous forme de messages à partir du modèle suivant :

un en-tête de message sur 4 octets :

2 octets pour la longueur du message (un entier de type *short int*)
2 octets pour le type de message (un entier de type *short int*)

le message proprement dit :

message de longueur quelconque (sans dépasser 32 Ko),
mais dont la longueur exacte est stockée dans l'en-tête du message

7	0	bonjour

36	0	exemple de message envoyé au serveur

9	1	au revoir

Programme **pipenoms** : programme serveur de communication de type canal de communication nommé qui reçoit des messages de longueur indéterminée à partir d'un seul programme client **pipenomc**. Les messages sont reçus "structurés" selon le modèle ci-dessus, ce qui permet de d'abord lire les 4 octets d'en-tête et de les décoder, puis de lire ensuite le nombre exact d'octets correspondant au message. Le type d'un message sert à établir une convention entre le serveur et le client, par exemple pour indiquer que c'est le dernier message envoyé.

Programme **pipenomc** : programme client qui communique avec un serveur de type canal de communication nommé pour lui envoyer des messages (<32Ko) qui sont lus au clavier. Ce programme doit constituer des messages "structurés" selon le modèle ci-dessus, afin de faciliter le travail de lecture du serveur **pipenoms**.

On pourrait également introduire dans l'en-tête le PID du client, afin que lors de la fin des échanges, le serveur puisse attendre proprement la terminaison du client. Une autre solution serait de transmettre comme tout dernier message, le PID du client.

Exercices (chapitre VII)

Exercice SEM :

Signalisation par sémaphore entre processus "proches".

Fichier : **sem.c** (répertoire **sem**)

Ce programme jouera le rôle d'un processus père qui donnera naissance à un nouveau processus fils dont le code sera lui-même contenu dans ce programme. Entre les deux processus issus de ce programme, vous établirez un lien par l'intermédiaire d'un sémaphore disponible dans chacun de ces deux processus.

La partie processus père du programme réalisera les opérations suivantes :

```
création d'un sémaphore
possession du sémaphore

lancement du processus fils

BOUCLE 4 fois

   BOUCLE  3 fois
      pause de 1 seconde
      afficher : "(PERE)  Je suis seul."
   FINBOUCLE

   libérer la possession du sémaphore

   BOUCLE  3 fois
      pause de 1 seconde
      afficher : "(PERE)  Je ne suis plus seul..."
   FINBOUCLE

   requérir la possession du sémaphore

FINBOUCLE

attente de la terminaison du fils
destruction du sémaphore
```

La partie processus fils du programme réalisera les opérations suivantes :

```
BOUCLE infinie
   attente de la libération du sémaphore
   pause de 1 seconde
   afficher : "  (FILS)  Je suis la aussi !"
FINBOUCLE
```

Exercice QUEUE :

Mise en œuvre d'une file d'attente de messages.

Fichiers : **queuesrv.c** et **queuecli.c** (répertoire **queue**)

Dans cet exercice, il s'agit d'établir une communication de type file d'attente de messages entre un programme serveur **queuesrv** et un programme client **queuecli**. Les deux processus correspondant à ces programmes ne sont pas en filiation l'un de l'autre.

Programme **queuesrv** : ce programme serveur crée une file d'attente de messages et attend les messages du client pour les afficher à l'écran. Ce programme se termine avec l'arrivée du dernier message du client : il attend alors la terminaison de ce client et s'arrête à son tour après avoir détruit la file d'attente.

Programme **queuecli** : ce programme client se connecte à la file d'attente de messages créée par le serveur. Il lit ensuite des messages au clavier et les envoie au fur et à mesure au serveur par le biais de la file d'attente. Quand il n'y a plus de messages à lire au clavier, ce programme envoie un dernier message indiquant au serveur la fin des messages, puis il s'arrête.

Exercice MEMSHARE :

Utilisation d'une mémoire partagée.

Fichiers : **memshrwr.c** et **memshrrd.c** (répertoire **memshare**)

Avec cet exercice, il s'agit de mettre en partage, entre deux processus **memshrwr** et **memshrrd,** une zone de mémoire afin de reconstituer un mécanisme d'échange de messages.

Programme **memshrrd** : ce programme joue le rôle d'un serveur qui reçoit des messages depuis le client et les affiche à l'écran. La mise à disposition d'un message sera signalée au serveur par le client via l'envoi d'un signal de type SIGUSR1. La fin de l'envoi de messages est signalée par le client via l'envoi d'un signal SIGUSR2.

Programme **memshrwr** : ce programme joue le rôle d'un client qui lit des messages au clavier et les envoie au serveur en les déposant dans la mémoire partagée. Pour indiquer au serveur qu'un message a été déposé dans la mémoire partagée et qu'il est disponible, le client lui envoie alors un signal de type SIGUSR1. Lorsqu'il n'y a plus de message à lire au clavier, ce programme client envoie un dernier signal SIGUSR2 au serveur pour lui indiquer la fin des messages, puis s'arrête.