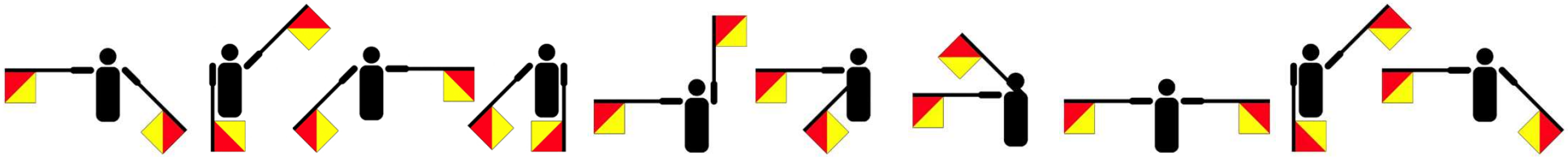
UNIVERSITÉ TOULOUSE III PAUL SABATIER

Les sémaphores



Définition

- Concept introduit par Dijkstra en 1965 pour synchroniser des processus
  - Section critique
  - Exclusion mutuelle
  - Synchronisation répondant à la logique d'une application

- Principe : un sémaphore représente un distributeur de tickets
  - Un processus peut
    - Prendre un ticket
    - Déposer un ticket
  - En l'absence de ticket, un processus demandeur est bloqué
  - Lorsqu'un ticket est déposé
    - S'il y a un processus en attente de ticket, celui-ci est débloqué
    - S'il n'y a personne en attente, le ticket sera pris par le prochain demandeur

Implémentation (Syntaxe « à la C++ »)

- **Un sémaphore est le regroupement d'un compteur et d'une file d'attente**

```
class Semaphore {
   private :                          // valeur > 0 = nombre de tickets disponibles
     int valeur;                      // valeur < 0 = nombre de processus en attente d'un ticket
     File processusEnAttente;  // les processus demandeurs bloqués en attente d'un ticket
};
```

- **Trois opérations y sont associées**

```
void initialiser (Semaphore &S, unsigned int nbTicketsInitial);
      // fixer le nombre initial de tickets au démarrage de l'application
void P (Semaphore &S);        // prendre un ticket :
     // le processus appelant ne termine cette opération que lorsqu'il a obtenu un ticket
void V (Semaphore &S);        // déposer un ticket :
    // ce qui peut conduire à débloquer un processus bloqué en attente d'un ticket
```

Implémentation

- Le système gère les opérations sur un même objet sémaphore en exclusion mutuelle, afin de garantir la cohérence de cet objet partagé
- Comment ?
  - En utilisant les mécanismes d'exclusion mutuelle précédemment présentés !
- On construit donc un mécanisme de synchronisation « évolué » avec attente passive en utilisant des mécanismes primitifs existants. Par exemple :
  - Masquage des interruptions durant l'exécution de ces opérations en monoprocesseur
  - Problèmes dans le cas des multiprocesseurs
    - Action atomique avec mémoire partagée
    - Instruction test-and-set
      - Si un bit est à 0 alors met à 1 et retourne 0
      - Sinon retourne 1
    - Instruction swap entre 1 et le bit de verrou
      - Si le bit est à 0 alors OK
      - Sinon re-tester ultérieurement

Un exemple d'implémentation

```
// Fixer le nombre initial de tickets au démarrage de l'application
void initialiser (Semaphore &S, unsigned int nbTicketsInitial) {


}
// Le processus courant prend un ticket
// Il ne termine cette opération que lorsqu'il a obtenu un ticket
void P (Semaphore &S) {
  // Tenter de prendre un ticket


    // Il n'y avait pas de ticket disponible, le processus courant est bloqué


  // Le processus a obtenu un ticket, immédiatement ou après avoir attendu
}
```

Un exemple d'implémentation (suite)

```
// Déposer un ticket, ce qui peut débloquer un processus en attente d'un ticket
void V (Semaphore &S) {
   // Un ticket de plus dans le distributeur


    // La valeur était donc négative, traduisant qu'au moins un processus attendait un ticket :
    // débloquer le processus en tête de la file

   // Le processus demandeur a été débloqué :
   // il peut avoir aussi pris l'UC au processus qui a déposé le ticket

}
```

Une autre implémentation du type sémaphore

```
class ImplantationSemaphore {
  private :
    unsigned int valeur;
            //compteur
    file processusEnAttente;
            //file d'attente
    . . .
}
```

```
void P (semaphore &S) {
    if (S.valeur > 0)
        S.valeur = S.valeur - 1;
    else
        Bloquer le processus courant dans S.processusEnAttente;
}

void V (semaphore &S) {
    if (S.processusEnAttente non vide)
        Débloquer le processus en tête de la file S.processusEnAttente;
    else
        S.valeur = S.valeur + 1;
}
```

Exemple 1 : synchronisation

Le sémaphore sem est initialisé à zéro

```
semaphore sem;
initialiser(sem, 0);
```

```
void P2 {
    …
    P(sem);
    …
    }
```

```
void P1 {
    …
    V(sem);
    …
    }
```

Exemple 2 : exclusion mutuelle

Le sémaphore mutex est initialisé à un

```
semaphore mutex;
initialiser(mutex, 1);
```

```
void P1 {
    . . .
    P(mutex);
    Section critique;
    V(mutex);
    . . .
    }
```

```
void P2 {
    . . .
    P(mutex);
    Section critique;
    V(mutex);
    . . .
    }
```

Exemple 3 : gestion d'un pool d'imprimantes

```
class GI {              // Gestion d'Imprimantes
private
  . . .
public
    // Demander une imprimante
    // et récupérer le numéro de l'imprimante allouée
    int demanderImprimante (void);

    void rendreImprimante (int numImp);
};
```

Exemple 3 : synchronisation des accès aux imprimantes

```
#define NB_IMP . . .
{       // Variables supposées partagées
        bool occupe[NB_IMP];
        Semaphore ImpLibre;
}
```

```
int demanderImprimante (void) {
      int i;

      P(ImpLibre);
      i = 1;
      while (occupe[i]) i++ ;
      occupe[i] = true;
      return (i);
}
```

```
void rendreImprimante (int numImp) {
  occupe[numImp] = false;
  V(ImpLibre);
}

void init (void) {
    int i;
    initialiser(ImpLibre, NB_IMP);
    for (i = 1; i < NB_IMP; i++)
      occupe[i] = false;
   }
}
```

Cela marche-t-il ?

Exemple 3 : synchronisation des accès aux imprimantes

```
#define NB_IMP . . .
{       // Variables supposées partagées
        bool occupe[NB_IMP];
        Semaphore ImpLibre,
}
```

```
int demanderImprimante (void) {
      int i;

      i = 1;

      while (occupe[i]) i++;
      occupe[i] = true;

      return (i);
}
```

```
void rendreImprimante (int numImp) {

  occupe[numImp] = false;



}
void init (void) {
    int i;


    for (i = 1; i < NB_IMP; i++)
      occupe[i] = false;
    }
}
```

Evaluation

- Concept de bas niveau
- L'utilisation reste simple dans le cas de l'exclusion mutuelle
- Une utilisation systématique peut être source d'erreurs
  - Un seul oubli ou un appel mal situé perturbe l'application complète
  - L'exclusion peut ne pas être assurée ou un interblocage peut apparaître dans un cas très rare
- Des outils plus structurés sont nécessaires dans les cas plus complexes
- Aucun langage évolué ne s'appuie uniquement sur les sémaphores

Variantes sémantiques

- **Sémaphore booléen**
  - Équivalent à un sémaphore à un seul jeton
  - Utilisation par une alternance de P et de V

- **Sémaphore privé**
  - Seul le processus propriétaire se bloque derrière ce sémaphore (opération P)
  - La file d'attente n'est plus nécessaire
  - Tout processus peut effectuer une opération V sur le sémaphore

Suite du cours

- **Exercices théoriques avec P et V**
  - Modèle du producteur-consommateur
  - Rendez-vous
- **Comment synchroniser des threads Posix [utilisé en TP]**
  - Avec des sémaphores d'exclusion mutuelle Posix ou verrous : type pthread_mutex_t
  - Avec des sémaphores à compteur Posix : type sem_t
- **Comment synchroniser des Processus Unix [Hors programme]**
  - Présentation de la bibliothèque des IPC (InterProcess Communication) Unix System V
    - Segment de mémoire partagée : pour enfin partager des données entre processus (parents ou non) !
    - Sémaphores : pour synchroniser des processus
    - Files de messages : pour aller plus loin que les tubes en échangeant des informations entre processus non parents

UNIVERSITÉ TOULOUSE III PAUL SABATIER

Exercices « théoriques »

[Ces exercices seront aussi concrètement réalisés en TP
en considérant que les activités parallèles sont des threads Posix]

Exercice 1 – Affichage alterné

- On suppose qu'une activité parallèle possède le comportement suivant :

```
Activité Afficheur {
 while (1) {
   Effectuer un traitement ;
   Afficher un message de plusieurs lignes à l'écran ;
   Effectuer un traitement ;
 }
}
```

- On souhaite synchroniser, à l'aide de sémaphores, plusieurs activités de ce type pour qu'elles alternent leurs messages à l'écran
- Proposer une solution pour 2 activités parallèles
- Généraliser la solution à N activités parallèles

Exécution souhaitée pour 2 activités parallèles :

. . . .
Activité 1 : début de mon message
Activité 1 : fin de mon message
Activité 2 : début de mon message
Activité 2 : fin de mon message
Activité 1 : début de mon message
Activité 1 : fin de mon message
Activité 2 : début de mon message
Activité 2 : fin de mon message
. . .
. . .

Exercice 2 – Modèles des producteurs/consommateurs

- On considère un buffer partagé par des activités parallèles de deux types :
  - Des producteurs qui déposent des messages dans ce buffer
  - Des consommateurs qui retirent des messages de ce buffer
- Le buffer comporte N cases et est géré circulairement

- Variante de base
  - Les dépôts se font dans l'ordre croissant des indices de cases, de manière circulaire
  - Les retraits se font dans l'ordre des dépôts, de manière circulaire aussi

- Proposer une solution utilisant des sémaphores pour que ces activités parallèles déposent et retirent leurs messages de manière cohérente

Exercice 2 – Exemple de dépôts/retraits

Demandes de dépôts/retraits

Producteur1

Producteur2

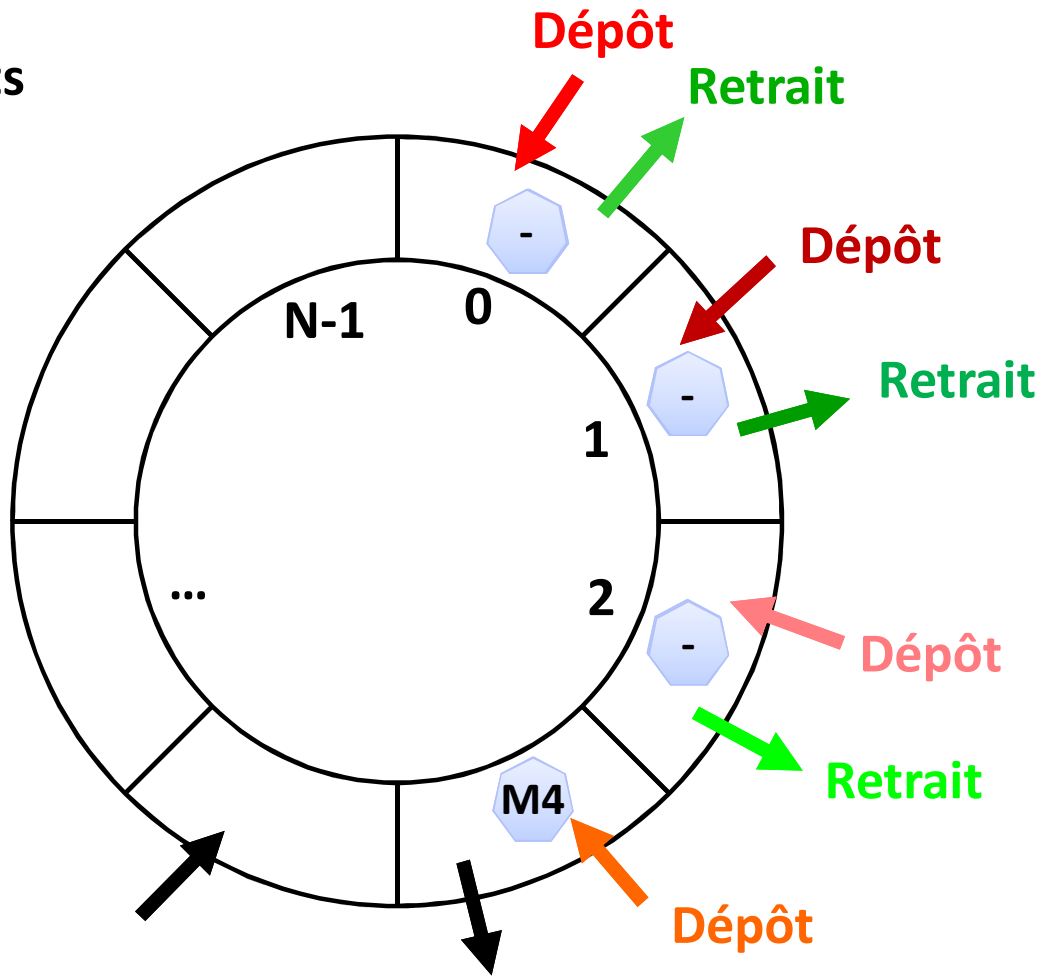
Consommateur1

Consommateur2

Consommateur3

Producteur3

Producteur4



Exercice 2 – Modèles des producteurs/consommateurs

Variante 1

- Les messages sont de deux types (0/1, blanc/noir, recto/verso…)

- On veut que les dépôts soient alternés dans le buffer
  - Un message d'un 1er type, un message du second, etc.
  - Les dépôts se font toujours dans l'ordre croissant des indices de cases, de manière circulaire
  - Les retraits se font toujours dans l'ordre des dépôts, de manière circulaire aussi

- Proposer une solution utilisant des sémaphores pour que ces activités parallèles déposent et retirent leurs messages de manière cohérente

Exercice 2 – Exemple de dépôts alternés/retraits

Demandes de dépôts/retraits

Producteur1(0)

Producteur2(0)

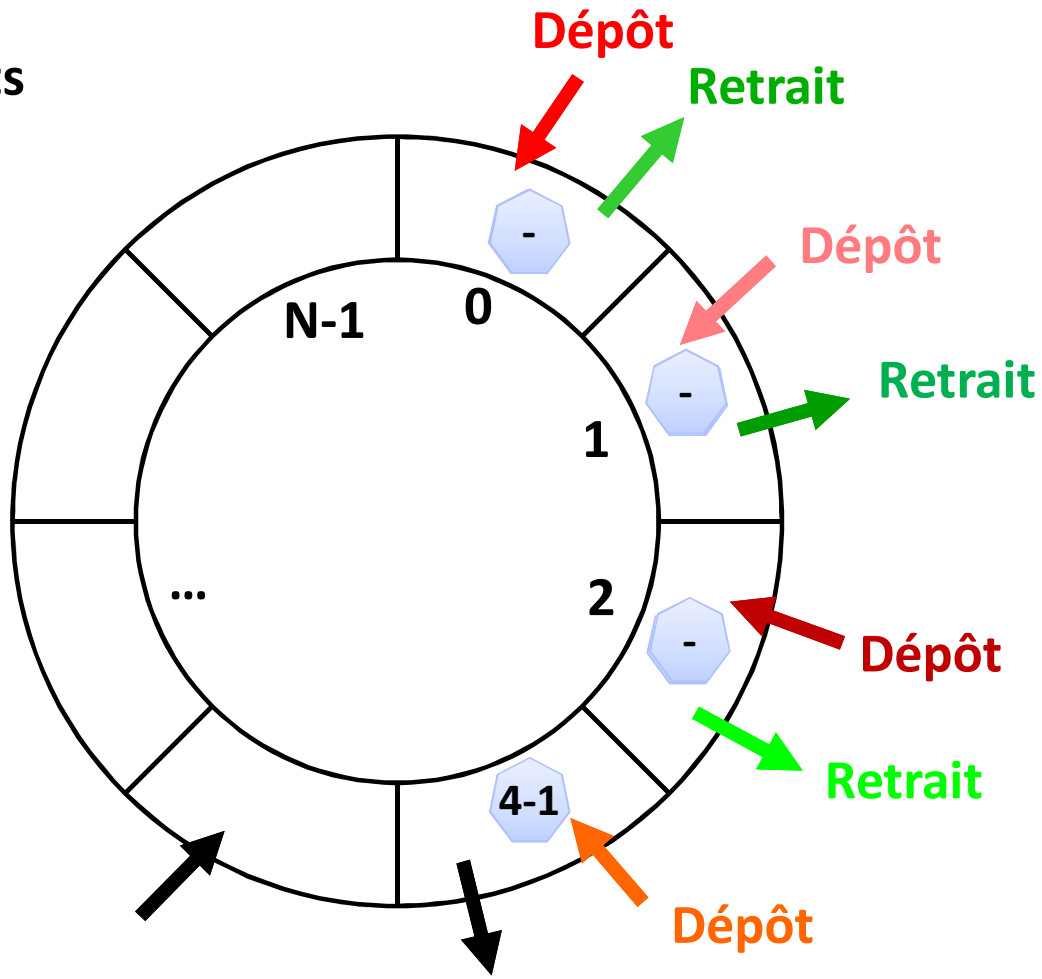
Consommateur1

Consommateur2

Consommateur3

Producteur3(1)

Producteur4(1)



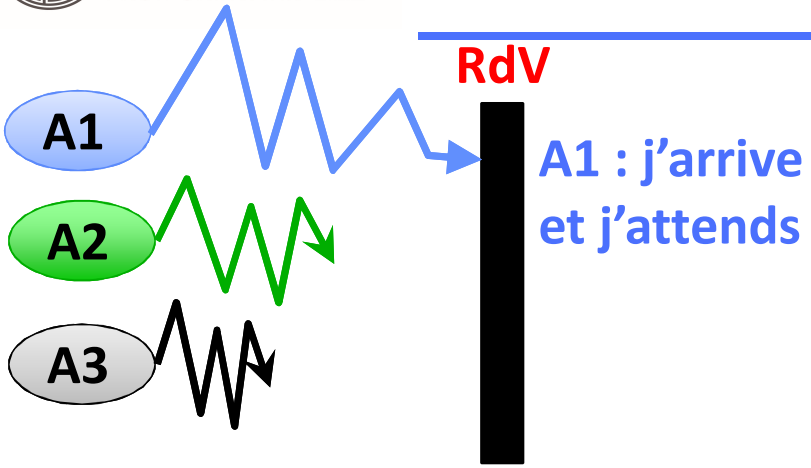
Exercice 2 – Modèles des producteurs/consommateurs

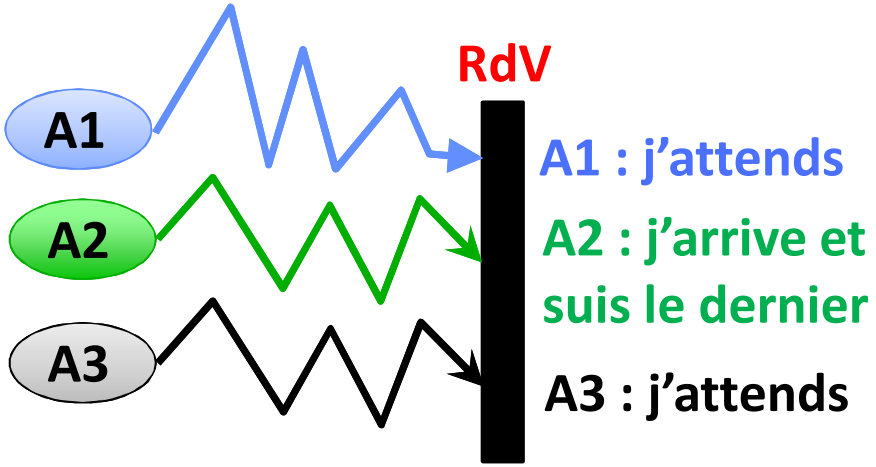
- **Variante 2**
- **Les messages sont de deux types (0/1, blanc/noir, recto/verso…)**
- **On n'impose plus de contrainte sur les dépôts**
  - **Les dépôts se font dans l'ordre croissant des indices de cases, de manière circulaire**
  - **Les retraits se font dans l'ordre des dépôts, de manière circulaire aussi**
- **Mais, les consommateurs précisent quel type de message ils veulent retirer**
  - **Et sont bloqués s'ils ne peuvent retirer un message du type attendu**
- **Proposer une solution utilisant des sémaphores pour que ces activités parallèles déposent et retirent leurs messages de manière cohérente**

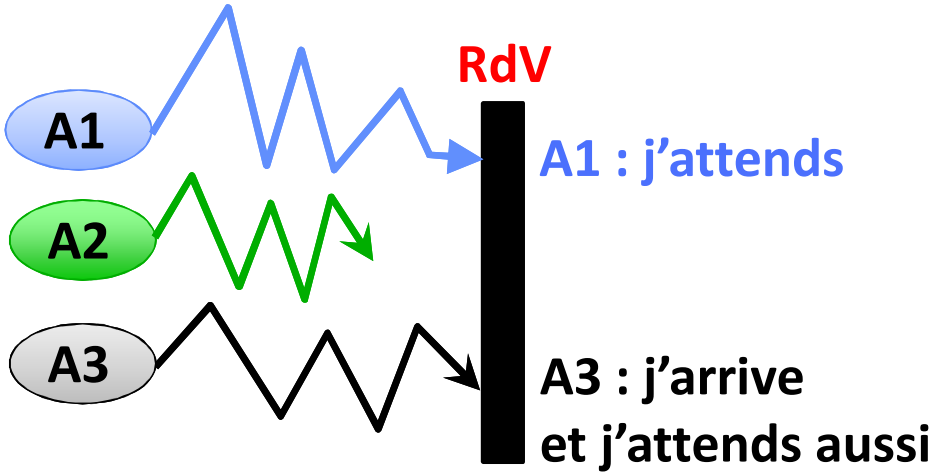
Exercice 3 – Rendez-vous

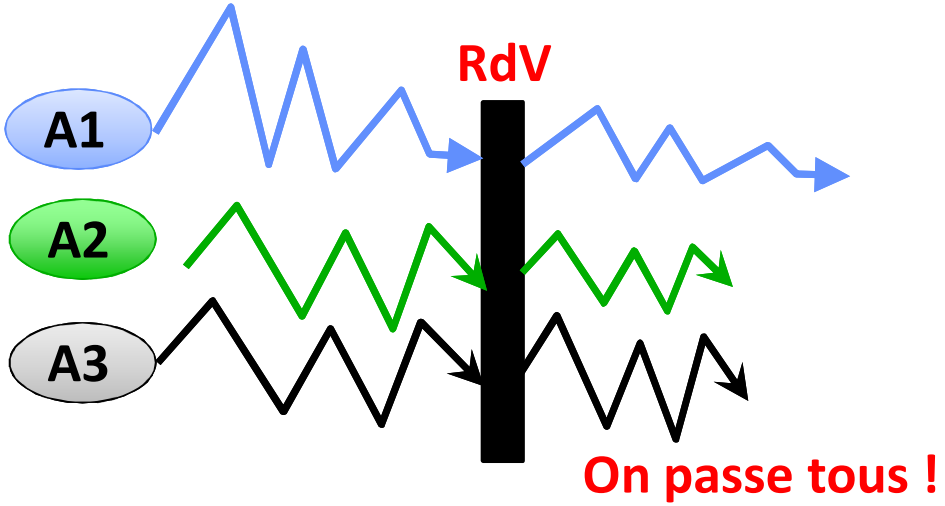
- On considère N activités parallèles qui doivent réaliser un rendez-vous
- Une activité arrivant au point de RdV doit se mettre en attente s'il existe au moins une autre activité qui n'y est pas arrivé
- Toutes les activités bloquées sur cette « barrière » peuvent la franchir lorsque la dernière y est arrivée
- Une activité a le comportement suivant
  - Je fais un certain traitement
  - J'arrive au point de RdV (j'attends les autres si elles n'y sont pas…)
  - Et je continue mon traitement
- Proposer une solution de synchronisation à l'aide des sémaphores

Exercice 3 – Exemple de RdV à 3









UNIVERSITÉ TOULOUSE III PAUL SABATIER

Synchronisation de threads par sémaphores

Sémaphores booléens d'exclusion mutuelle (ou verrous)

- Normaux
  - Un P est bloquant si le sémaphore a déjà été franchi par un autre thread
- Récursifs
  - Un thread qui a franchi le sémaphore en devient propriétaire et peut effectuer plusieurs P sans se bloquer

Sémaphores avec compteur

- Sémaphores classiques avec compteur (Dijkstra)

UNIVERSITÉ TOULOUSE III PAUL SABATIER

Sémaphores d'exclusion mutuelle POSIX : pthread_mutex_t

Bibliothèque pthread

- Synchronisation de base : mutex et condition
- Types (opaques) : <sys/types.h>
  - pthread_t, pthread_key_t
  - pthread_mutex_t
  - pthread_cond_t
  - pthread_once_t
- Attributs : standard + propres à l'implantation
  - pthread_attr_t
  - pthread_mutexattr_t
  - pthread_condattr_t
- Bibliothèque → #include <pthread.h> + compilation : -lpthread
- Gestion des erreurs
- Primitives « thread-safe »

Mutex Posix : Création

- Création avec les caractéristiques par défaut
  - Macro, utlisée lors de la déclaration

```
pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;
```

- Création en fixant les caractéristiques

```
int pthread_mutex_init(pthread_mutex_t           *mutex,
                       const pthread_mutexattr_t *attr);
```

  - mutex = descriptif du mutex, mis à jour
  - attr = attributs optionnels pour l'initialisation
  - Erreurs : EAGAIN, ENOMEM, EPERM, EBUSY, EINVAL
- Initialisé dans l'état non verrouillé (avec un ticket) !
- Éviter de faire une copie de mutex…

Mutex Posix : Destruction

```
int pthread_mutex_destroy(pthread_mutex_t *mutex);
```

- mutex = descriptif du mutex à détruire

- Possible si non verrouillé
- Ressources libérées
- Retourne 0 en cas de succès, un code d'erreur sinon
- Erreurs : EINVAL, EBUSY

Attributs d'un mutex Posix

- Changer les caractéristiques par défaut
- Créer et initialiser par défaut un attribut de mutex

```
int pthread_mutexattr_init(pthread_mutexattr_t *attr);
```

  - attr = attribut du mutex (valeur par défaut)
  - Erreurs : ENOMEM
- Détruire un attribut de mutex

```
int pthread_mutexattr_destroy(pthread_mutexattr_t *attr);
```

  - attr = descriptif
  - Objet devient non initialisé
  - Erreurs : EINVAL

Exemple d'attribut d'un mutex Posix : type de mutex

Positionner le type d'un mutex

```
int pthread_mutexattr_settype(pthread_mutexattr_t *attr, int type);
```

- attr = attributs du mutex (valeur par défaut)
- type = type du mutex :
  - PTHREAD_MUTEX_NORMAL
  - PTHREAD_MUTEX_ERRORCHECK
  - PTHREAD_MUTEX_RECURSIVE
  - PTHREAD_MUTEX_DEFAULT
- Erreurs : EINVAL

Obtenir le type d'un mutex

```
int pthread_mutexattr_gettype(pthread_mutexattr_t *attr, int *type);
```

- Erreurs : EINVAL

Mutex Posix : Opérations

- Verrouiller un mutex (opération P)

```
int pthread_mutex_lock(pthread_mutex_t *mutex);
```

  - Erreurs : EINVAL, (EAGAIN, EDEADLK)

- Tenter de verrrouiller un mutex (opération P non bloquante)

```
int pthread_mutex_trylock(pthread_mutex_t *mutex);
```

  - Erreurs : EINVAL, EBUSY, (EAGAIN, EDEADLK)

- Déverrouiller un mutex (opération V)

```
int pthread_mutex_unlock(pthread_mutex_t *mutex);
```

  - Erreurs : (EINVAL, EPERM)

UNIVERSITÉ TOULOUSE III PAUL SABATIER

Sémaphores avec compteurs POSIX : sem_t

Sémaphores Posix avec compteurs

- **Sémaphores Posix (1003.1b)**
  - #include <semaphore.h>
  - Type « opaque » : sem_t
- **Créer et initialiser un sémaphore Posix**

```
int sem_init(sem_t *sem, int pshared, unsigned int value);
```

  - sem : identificateur du sémaphore créé
  - pshared : 0 ➔ partage entre threads, > 0 ➔ partage entre processus (si supporté)
  - value : Valeur initiale
  - Succès : 0. Échec : -1 + errno : EINVAL, ENOSYS
- **Détruire un sémaphore Posix**
  - Succès : 0. Échec : -1 + errno : (ENOSYS)

```
int sem_destroy(sem_t *sem);
```

Sémaphores Posix : opérations P

Opération P bloquante

```
int sem_wait(sem_t *sem);
```

- Demande un ticket sur le sémaphore référencé
- Succès : 0. Échec : -1 + errno : EINVAL, EINTR, ENOSYS

Opération P non bloquante

```
int sem_trywait(sem_t *sem);
```

- Succès : 0. Échec : -1 + errno : EAGAIN

Opération P temporisée

```
int sem_timedwait (sem_t *sem, const struct timespec *abs_timeout);
```

- Succès : 0. Échec : -1 + errno : ETIMEDOUT

Sémaphores Posix : opération V

Opération V

```
int sem_post(sem_t *sem);
```

- Dépose un ticket dans le sémaphore référencé
- Succès : 0. Échec : -1 + errno : EINVAL

Variante d'implantation

- **Principe**
  - Le compteur est systématiquement décrémenté lors d'une opération P
- **La sémantique du compteur est alors la suivante :**
  - S'il est positif, le compteur indique le nombre de jetons disponibles
  - S'il est négatif, la valeur absolue du compteur indique le nombre de processus en attente derrière le sémaphore
- **Invariant**
  - S.C = S_C_Initial + Nombre_Total_V(S) – Nombre_Total_P(S)
- **Obtenir la valeur du compteur d'un sémaphore Posix**

```
int sem_getvalue(sem_t *sem, int *sVal);
```

  - Succès : 0. Échec : -1 + errno : EINVAL
  - sVal ne reflète pas forcément la réalité !

UNIVERSITÉ TOULOUSE III PAUL SABATIER

Communication et synchronisation entre processus
InterProcess Communication (IPC) UNIX System V

Pour aller plus loin qu'en TP…

en faisant se synchroniser des processus

Les IPC UNIX System V - Généralités

- **Permettent à des processus sans lien de parenté**
  - De partager des données (via des segments de mémoire partagée – *shared memory*)
  - De communiquer par des échanges de messages (via des files de messages – *queues*)
  - De se synchroniser (via des sémaphores généraux)
- **Bibliothèques de base**
  - #include <sys/types.h>
  - #include <sys/ipc.h>
- **Un IPC possède**
  - Un identificateur externe ou clé (unique sur la machine) de type key_t
  - Un identificateur interne (unique au sein d'un processus) de type int
  - Des droits d'accès (user/group/others)
  - La possibilité de restreindre son utilisation à la descendance de son créateur (clé : IPC_PRIVATE)

Les IPC UNIX System V – Gestion des ressources

- **Un IPC est une ressource persistante sur une machine**
  - **Le nombre d'IPC est limité**
    - **Globalement**
    - **Pour chaque utilisateur**
- **Nécessité de détruire un IPC quand il n'est plus utilisé**
  - **Par commande Unix (voir le man)**
    - **ipcrm [-m id] [-q id] [-s id]**
      - **-m : détruire le segment de mémoire partagée identifié par id**
      - **-q : détruire la file de messages identifiée par id**
      - **-s : détruire l'ensemble de sémaphore identifié par id**
  - **Par programmation**
    - **Primitive associée à chaque type d'IPC**
      - **shmctl, msgctl, semctl**
- **Visualisation des IPC existant sur une machine**
  - **Commande UNIX ipcs (voir le man)**

UNIVERSITÉ TOULOUSE III PAUL SABATIER

Partager des variables entre processus
Segments de mémoire partagée
UNIX System V

Un IPC = une zone de mémoire partagée

- #include <sys/shm.h>

- Avant de manipuler un segment de mémoire partagée, un processus doit
  - Le créer ou l'ouvrir (si un autre processus l'a déjà créé) afin d'obtenir un identificateur interne pour ce segment
  - « L'attacher » à son espace d'adressage i.e. obtenir une adresse référençant ce segment
    - Par exemple, pour une zone de mémoire partagée représentant un entier, il attache le segment à l'adresse int *p
- Il accède ensuite à cette zone partagée via cette adresse
  - Il peut consulter ou modifier la valeur de la zone de mémoire partagée
    - Par exemple : printf("Valeur partagée = %d\n", *p); ou : (*p)++; ou : *p = *p * 10;
- Quand il ne veut plus manipuler ce segment, il libère l'adresse le référençant en « détachant » ce segment

Création d'un segment de mémoire partagée

```
int shmget(key_t key, size_t size, int shmflg);
```

- **key = nom externe associé à ce segment de mémoire partagée**
  - Unique, obtenu notamment par ftok() [voir man], si des processus non parents veulent y accéder
  - Privé, égal à IPC_PRIVATE, si l'utilisation du segment est restreinte aux seuls descendants du créateur du segment
- **size = taille en octets allouée à ce segment**
  - Dépend de l'information à partager
    - Exemple : sizeof(int) pour partager une variable entière
    - Exemple : sizeof(struct maStructure) pour partager différentes informations regroupées dans une structure
- **shmflg = indicateur, suite de bits comprenant**
  - IPC_CREAT : création d'un nouveau segment
  - IPC_EXCL : indique si une éventuelle création doit échouer ou non
    - 1 : la création échoue si le nom externe est déjà utilisé
    - 0 : le processus obtient le numéro interne d'un segment déjà créé (ouverture)
  - 9 bits de faible poids spécifiant les droits d'accès au segment si on le crée

❑ **Retourne l'identificateur interne ou -1 (+ errno) si erreur**

Ouverture d'un segment de mémoire partagée

```
int shmget(key_t key, size_t size, int shmflg);
```

- key = nom externe associé au segment
  - Le segment doit exister
- size = taille en octets
  - Peu importe, fixé à la création
- shmflg = indicateur
  - Droits fixés à la création, doivent permettre à l'appelant d'utiliser le segment
  - IPC_EXCL et IPC_CREAT ne doivent pas être tous les deux positionnés

- Retourne l'identificateur interne ou -1 (+ errno) si erreur

Contrôle d'un segment de mémoire partagée (1)

- Changer les droits d'accès, le propriétaire, le groupe . . .
- Consulter les caractéristiques
  - Propriétaire, groupe
  - Droits
  - Dernière modification
  - Taille . . .
- Verrouiller / déverrouiller le segment en mémoire centrale
- Détruire le segment de mémoire partagée

```
int shmctl(int shmid, int cmd, struct shmid_ds *buf);
```

Contrôle d'un segment de mémoire partagée (2)

```
int shmctl(int shmid, int cmd, struct shmid_ds *buf);
```

- shmid = identificateur interne du segment
- cmd = action de contrôle
  - IPC_STAT : récupérer le descripteur du segment (et donc ses caractéristiques) dans la zone pointée par buf
  - IPC_SET : modifier les caractéristiques du segment à partir du descripteur à l'adresse buf
  - SHM_LOCK : verrouiller le segment en mémoire centrale
  - SHM_UNLOCK : déverrouiller le segment en mémoire centrale
  - IPC_RMID : détruire le segment de mémoire partagée
  - . . .

- Retourne 0 si succès, -1 sinon + errno

Attacher un segment de mémoire partagée

```
void *shmat(int shmid, const void *shmaddr, int shmflg);
```

- shmid = identificateur interne du segment
- shmaddr = adresse à laquelle attacher le segment ou NULL si on laisse choisir le système
- shmflg = indicateur (les 2 valeurs sont possibles : |)
  - SHM_RND : arrondir ou non l'adresse
  - SHM_RDONLY : segment en lecture seule ou non

- Retourne l'adresse d'attachement ou -1 (+ errno) en cas d'échec

- On peut voir cette étape d'attachement comme un « malloc » : le processus doit obtenir une adresse (dans son espace d'adressage) référençant le segment de mémoire partagée (situé en mémoire centrale) avant de pouvoir le manipuler

Détacher un segment de mémoire partagée

```
int shmdt(const void *shmaddr);
```

- shmaddr : adresse d'attachement

- Retourne 0 en cas de succès, -1 (+ errno) sinon

- On peut voir cette étape de détachement comme un « free » : le processus libère l'adresse (dans son espace d'adressage) référençant le segment de mémoire partagée (situé en mémoire centrale) car il n'a plus d'accès à y faire

UNIVERSITÉ TOULOUSE III PAUL SABATIER

Synchronisation de processus par sémaphores
IPC UNIX System V

Un IPC = un ensemble de sémaphores

- #include <sys/sem.h>

- Un IPC représente un ensemble de sémaphores et non un unique sémaphore !

- Économie de ressources
  - Regrouper tous les sémaphores utiles à une même application dans un seul IPC

- Un sémaphore dans cet ensemble est identifié par le couple :
  (numéro interne de l'ensemble, numéro du sémaphore dans l'ensemble)

  - Le 1er sémaphore de l'ensemble porte le numéro 0

Création d'un ensemble de sémaphores

```
int semget(key_t key, int nsems, int semflg);
```

- **key = nom externe associé à cet ensemble de sémaphores**
  - Unique, obtenu notamment par ftok() [voir man], si des processus non parents veulent y accéder
  - Privé, égal à IPC_PRIVATE, si l'utilisation de l'ensemble est restreinte aux seuls descendants du créateur
- **nsems = nombre de sémaphores dans cet ensemble**
  - Dépend de la synchronisation à mettre en place pour l'application
    - Exemple : 4 pour la version de base du modèle producteurs/consommateurs
- **semflg = indicateur, suite de bits comprenant**
  - IPC_CREAT : création d'un nouvel ensemble de sémaphores
  - IPC_EXCL : indique si une éventuelle création doit échouer ou non
    - 1 : la création échoue si le nom externe est déjà utilisé
    - 0 : le processus obtient le numéro interne d'un ensemble déjà créé (ouverture)
  - 9 bits de faible poids spécifiant les droits d'accès à l'ensemble si on le crée

- **Retourne l'identificateur interne ou -1 (+ errno) si erreur**

Ouverture d'un ensemble de sémaphores

```
int semget(key_t key, int nsems, int semflg);
```

- key = nom externe associé à l'ensemble de sémaphores
  - L'ensemble doit exister
- nsems = nombre de sémaphores
  - Peu importe, fixé à la création
- semflg = indicateur
  - Droits fixés à la création, doivent permettre à l'appelant d'utiliser l'ensemble
  - IPC_EXCL et IPC_CREAT ne doivent pas être tous les deux positionnés

- Retourne l'identificateur interne ou -1 (+ errno) si erreur

Contrôle d'un ensemble de sémaphores

```
int semctl(int semid, int semnum, int cmd, union semun arg);
```

- semid = identificateur interne de l'ensemble de sémaphores
- semun = numéro d'un sémaphore de l'ensemble
- cmd = action de contrôle
  - SETVAL : Initialiser la valeur du sémaphore de numéro semnum de l'ensemble de sémaphores
  - SETALL : Initialiser les valeurs des différents sémaphores de l'ensemble (la valeur de semnum importe peu)
  - IPC_RMID : Détruire l'ensemble de sémaphores
  - IPC_STAT, IPC_SET, GETALL : cf. man
- arg = union de types à définir *(à recopier du man)*

```
union semun {
  int              val;    /* Type utilisé si SETVAL             */
  struct semid_ds *buf;    /* Type utilisé si IPC_STAT & IPC_SET */
  ushort_t        *array;  /* Type utilisé si GETALL & SETALL    */
}
```

- Retourne >0 si succès (dépend de cmd) ou -1 (+ errno) si échec

Initialiser les valeurs d'un ensemble de sémaphores

```
int semctl(int semid, int semnum, int cmd, union semun arg);
```

- **Initialiser la valeur d'un sémaphore d'un ensemble**
  - cmd = SETVAL
  - Initialiser la valeur de arg.val avec le nombre de jetons voulu
  - arg.val sera affectée à la valeur du sémaphore identifié par (semid, semnum)
- **Initialiser les valeurs des différents sémaphores d'un ensemble**
  - cmd = SETALL
  - Réserver la place mémoire pour le tableau arg.array (malloc en fonction du nombre de sémaphores de l'ensemble)
  - Initialiser ses valeurs (ce doivent être des entiers courts non signés) avec les nombres de jetons voulus (dans l'ordres des sémaphores de l'ensemble)
  - Les valeurs de arg.array seront affectées aux valeurs des sémaphores de l'ensemble

Description d'une opération de blocage / déblocage

```
struct sembuf {
    u_short_t sem_num; /* Numéro du sémaphore dans l'ensemble */
    short     sem_op;  /* Opération à réaliser sur ce sémaphore */
    short     sem_flg; /* Indicateurs */
}
```

- Le champ sem_op décrit l'opération à réaliser
  - sem_op > 0 : dépôt de sem_op jetons supplémentaires dans le sémaphore sem_num
    jetons éventuellement consommés par des processus bloqués en attente
    ➔ opération V
  - sem_op < 0 : retrait de |sem_op| jetons du sémaphore de numéro sem_num
    blocage éventuel tant que tous les jetons ne sont pas disponibles
    ➔ Opération P
  - sem_op = 0 : blocage de l'appelant tant que le nombre de jetons n'est pas nul

Opérations P & V

```
int semop (int semid, struct sembuf *array, size_t nops);
```

- semid = identificateur interne de l'ensemble de sémaphores
- array = tableau dont chacune des nops cases décrit une opération (P ou V) à réaliser sur l'ensemble
- nops = nombre d'opérations (P ou V) à réaliser (sans ressortir du mode noyau)

❑ Retourne 0 si succès, -1 (+ errno) sinon

UNIVERSITÉ TOULOUSE III PAUL SABATIER

Les files de messages
Ou
boîtes aux lettres
IPC UNIX System V

Un IPC = une file de messages

- #include <sys/msg.h>

- Permet d'échanger, de manière synchronisée, des messages via une file
- Une file de messages possède une capacité limitée
  - Fixée à MSGMNB octets par défaut
  - Pouvant être modifiée par une opération de contrôle
- Un processus qui veut « poster » un message est bloqué s'il n'y a pas la place
- Un processus qui veut « retirer » un message d'un certain type est bloqué si aucun message de ce type n'est disponible
- Possibilité de rendre non bloquante, les opérations de dépôts et de retrait (IPC_NOWAIT)

Création / Ouverture d'une file de messages

```
int msgget(key_t key, int msgflg);
```

- key = nom externe associé à cette file
  - Unique, obtenu notamment par ftok() [voir man], si des processus non parents veulent y accéder
  - Privé, égal à IPC_PRIVATE, si l'utilisation de la boîte à lettres est restreinte aux seuls descendants du créateur
- msgflg = indicateur, suite de bits comprenant
  - IPC_CREAT : création d'une nouvelle file de messages
  - IPC_EXCL : indique si une éventuelle création doit échouer ou non
    - 1 : la création échoue si le nom externe est déjà utilisé
    - 0 : le processus obtient le numéro interne d'une boîte à lettres déjà créée (ouverture)
  - 9 bits de faible poids spécifiant les droits d'accès à la file si on la crée

- Retourne l'identificateur interne ou -1 (+ errno) si erreur
- Pour l'ouverture
  - La file de messages doit avoir été créée
  - IPC_EXCL et IPC_CREAT ne doivent pas être tous les deux positionnés

Contrôle d'une file de messages

```
int msgctl(int msgid, int cmd, struct msgid_ds *buf);
```

- msgid = identificateur interne de la file de messages
- cmd = action de contrôle
  - IPC_RMID : Détruire la file de messages
  - IPC_STAT, IPC_SET, IPC_INFO, MSG_INFO, MSG_STAT : cf. man
- Buf = adresse d'un descripteur de file de messages pour récupérer ou positionner des caractéristiques pour la file identifiée par msgid

- Retourne >0 si succès (dépend de cmd) ou -1 (+ errno) si échec

Définition d'un message

- Message défini par
  - Un pointeur sur le premier octet de la suite (texte du message)
  - La longueur du message
  - Le type du message…
- Exemples de structures légales

```
struct msgbuf {
        long mtype;          /* type du message  */
        char mtext[ ];       /* texte du message */
}
struct msgbuf {
        long mtype;          /* type du message  */
        int n;
        char mtext[5];
}
```

Dépôt d'un message dans une file

```
int msgsnd(int msgid, const void *msgp, size_t msgsz, int msgflg);
```

- msgid = identificateur interne de la file de messages
- msgp = adresse du message à envoyer (structure définie par l'utilisateur)
- msgsz = taille du message (en octets)
- msgflg = indicateur
  - IPC_NOWAIT = 0 : Bloquer l'appelant jusqu'à ce que le dépôt soit effectué
  - IPC_NOWAIT = 1 : Laisser l'appelant poursuivre son exécution en l'avertissant que son message n'a pas été déposé (errno = EAGAIN)

- S'il n'y a assez de place dans la file pour déposer, l'appelant est bloqué (sauf si IPC_NOWAIT positionné) jusqu'à ce qu'il y ait assez de place
- Retourne 0 si succès, -1 (+ errno) sinon

Retrait d'un message dans une file (1)

```
ssize_t msgrcv(int msgid, void *msgp, size_t msgsz,
               long msgtyp, int msgflg);
```

- msgid = identificateur interne de la file de messages
- msgp = adresse du message récupéré
- msgsz = taille maximale du message attendu (en octets)
- msgtyp = type du message attendu
  - 0 : premier message de la file
  - > 0 : premier message de la file du type msgtyp
  - < 0 : premier message de la file d'un type inférieur ou égal à msgtyp
- msgflg = indicateur
  - IPC_NOWAIT : non bloquant si pas de message du bon type disponible
  - MSG_EXCEPT : Premier message qui diffère de msgtyp si msgtyp > 0
  - MSG_NOERROR : Tronquer le message s'il est plus long que msgsz

Retrait d'un message dans une file (2)

```
ssize_t msgrcv(int msgid, void *msgp, size_t msgsz,
               long msgtyp, int msgflg);
```

- S'il n'y a assez de message du bon type disponible, l'appelant est bloqué jusqu'à ce qu'un message puisse être retiré
  - En tenant compte des indicateurs positionnés dans msgflg
- Retourne la taille effective du message retiré si succès, -1 (+ errno) sinon
- Remarque
  - Si on récupère le message dans une variable du type struct msgbuf *
    - Le champ mtype contient le type du message
    - Le champ mtext contient le texte du message

UNIVERSITÉ TOULOUSE III PAUL SABATIER

Tubes de communication UNIX (ou « Pipes » UNIX)

- Chaque processus dispose d'un segment de données
- Deux processus quelconques ne disposent

MAIS

- Il arrive que des processus produisent des informations qui devront être utilisées par d'autres processus

Définition d'un tube de communication

- **en mémoire centrale**
  - Bien que de structure analogue à une simple variable en mémoire centrale, les opérations pour manipuler un tube sont (à l'exception de la création et de l'ouverture)
    - read, write, dup, close
  - Des entrées dans la table des fichiers ouverts sont associées aux tubes

- **Fichier de taille**
  - Il ne sera pas possible d'écrire directement de très longues séquences d'octets dans un tube
- **Accès**
  - Les lectures et les écritures effectuées par les processus utilisateurs devront respecter certaines et aux conditions dans lesquelles elles devront être effectuées
- **Fichier ayant des**
  - A l'inverse d'une lecture classique dans un fichier, une lecture dans un tube entraîne la des informations lues (Il s'agit d'une extraction)

Création d'un tube de communication

- Via la primitive pipe

```
int pipe (int Tube[2]);
/* En retour : */
/* Tube[0] est ouvert en lecture  */
/* Tube[1] est ouvert en écriture */
```

- Ce tube est connu et sera utilisable par le processus appelant ainsi que tous les processus de sa

- Sémantique
  - Ouverture de dans la FDT
    - Une entrée correspond à une ouverture du tube en
    - L'autre entrée correspond à l'ouverture du tube en
  - Les deux entrées sont mémorisées dans les deux éléments du tableau paramètre
  - pipe retourne la valeur -1 si la création du tube échoue

Exemple de création d'un tube de communication

Rappel :

Le retour des primitives Unix (en général, 0 si succès, -1 si échec) doit toujours être testé !

Dans certains des exemples ou des corrigés, cela n'est pas fait par manque de place

```
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <unistd.h>

int main(void) {
 int tube[2];

 if (pipe(tube) == -1)  {
  perror("Problème création tube");
  exit(1);
 }
 switch (fork())  {
  case -1 :
   printf("Échec création processus fils \n");
   exit(2);
  case 0 : // Contexte du fils
   printf("Code du fil \n");
   . . . .
   exit(0);
  default : // Contexte du père
   . . . .
  }
 . . . . .    // Code du père (si le fils a fait exit())
 exit(0);
}
```

Lecture dans un tube de communication

- Un lecteur lit dans un tube en utilisant la primitive read (voir cours de L2)

```
#include <unistd.h>

int   read(
   int Nom_Interne,
   void *Adresse,
   unsigned Nombre_Demande);
```

- au tube
  - Cas 1 : Les N octets demandés sont présents dans le tube
    - Les N octets sont retirés et transmis au processus appelant dans la zone définie par Adresse
    - read le nombre N d'octets transmis
  - Cas 2 : Le tube n'est pas vide mais contient moins des N octets demandés
    - Les octets présents sont retirés et transmis au processus appelant dans la zone définie par Adresse
    - read le nombre d'octets effectivement présents et transmis
  - Cas 3 : Le tube est
    - Le processus appelant est ce que l'une des deux conditions suivantes soit vérifiée
      - Il y a un nombre non nul d'octets présents dans le tube (le tube n'est plus vide)
      - Le tube n'est plus ouvert en écriture par processus

Très Important

Écriture dans un tube de communication

- Un rédacteur écrit dans un tube en utilisant la primitive write (cf. cours de L2)

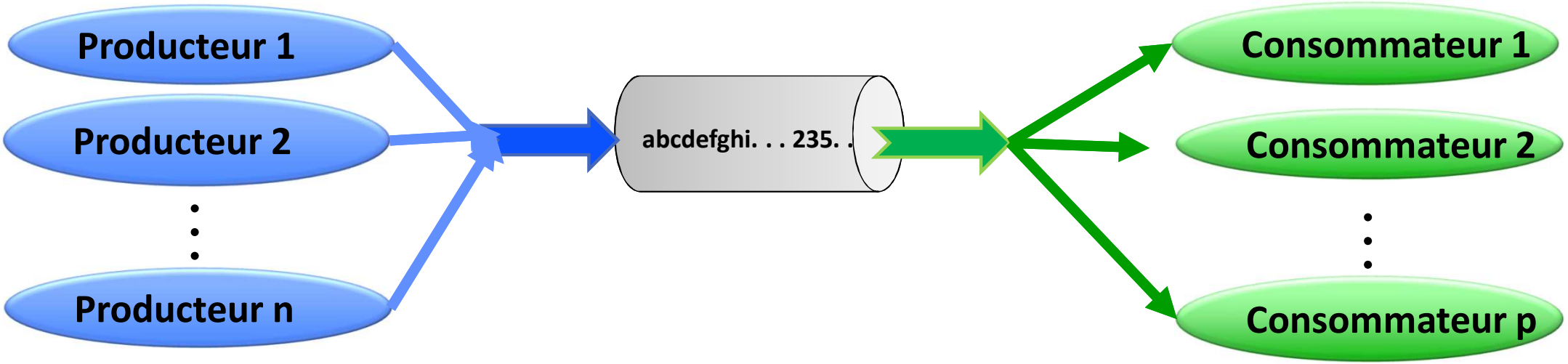
```
#include <unistd.h>

int  write(
  int Nom_Interne,
  void *Adresse,
  unsigned Nombre_Transmis);
```

Très Important

- au tube
  - L'opération write ne se termine que lorsque les Nombre_Transmis octets transmis par le processus appelant dans la zone désignée par Adresse ont été écrits dans le tube
    - Le transfert est réalisé au fur et à mesure que la place se libère dans le tube
    - Lorsque le tube est plein, le processus appelant est bloqué jusqu'à ce qu'un lecteur libère de la place par appel à read
  - L' dans un tube qui n'est en lecture génère une (signal « SIGPIPE », voir le cours sur les signaux)
- Si un est détecté lors d'un write (réception d'un signal, capacité maximale du support atteinte, etc.), l'écriture est interrompue
- write retourne le nombre d'octets écrits ou -1 en cas d'erreur

Tubes de communication – Synchronisation

- Certains processus sont amenés à écrire dans un tube :
- Certains processus sont amenés à lire dans un tube :



- de la synchronisation
  - Une opération qui n'est pas possible est pour le processus appelant
  - La est bloquante si le tube est
  - est bloquante si le tube est

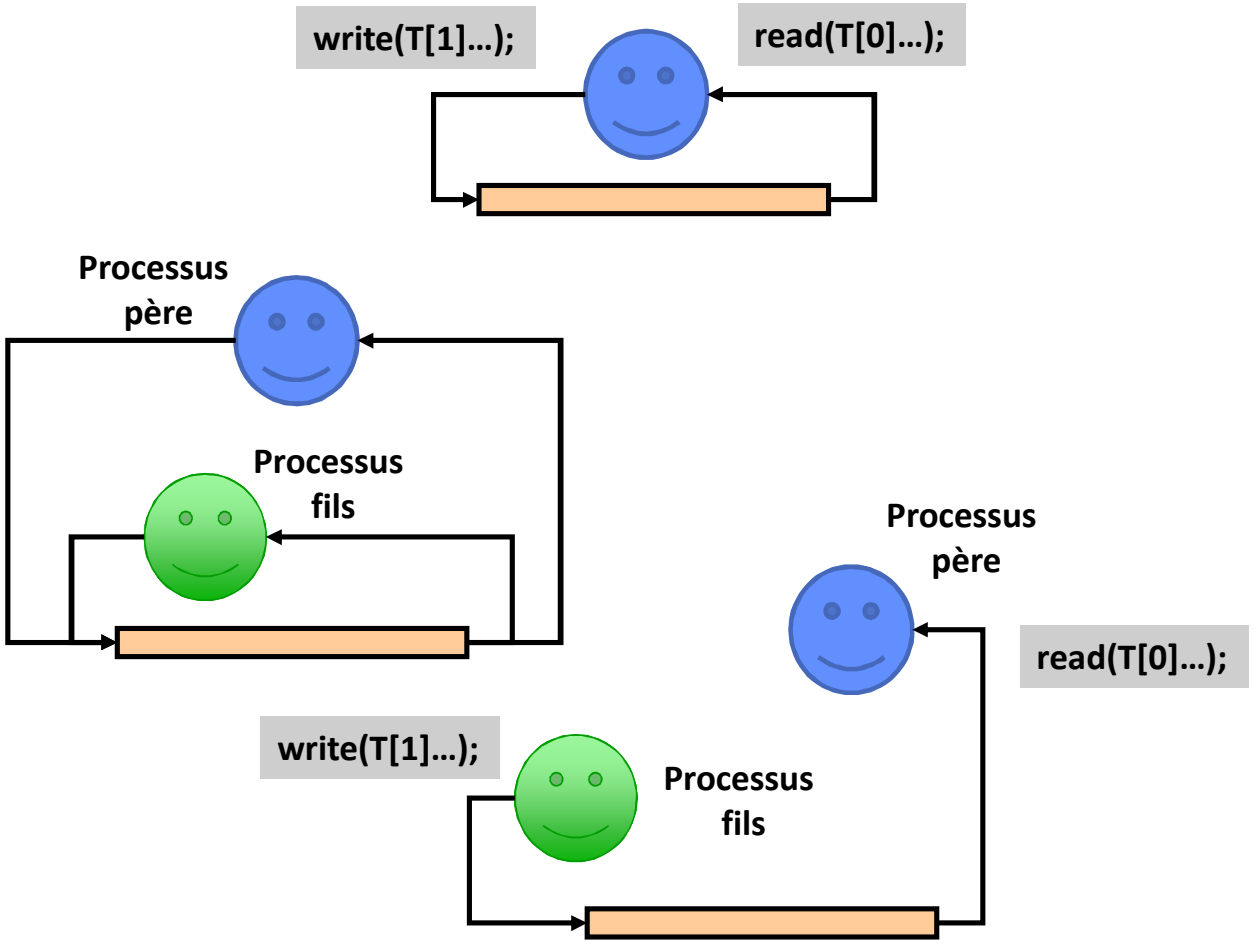
« Orientation » des tubes de communication

- Il est préférable les tubes entre les deux familles de processus utilisateurs
  - L'utilisation d'un même tube en lecture ET en écriture, par un même processus, est dangereuse
  - Elle peut conduire à un et doit être effectuée avec précaution
  - Chaque processus devra donc
- Attention
  - Il est important de la réponse de la primitive pipe afin de vérifier que le tube souhaité a correctement été créé
  - La de l'entrée en est , sinon cela a des conséquences sur le déblocage des lecteurs en attente
  - La de l'entrée en est mais elle a l'avantage de libérer une entrée de la FDT dont le nombre d'entrées est limité

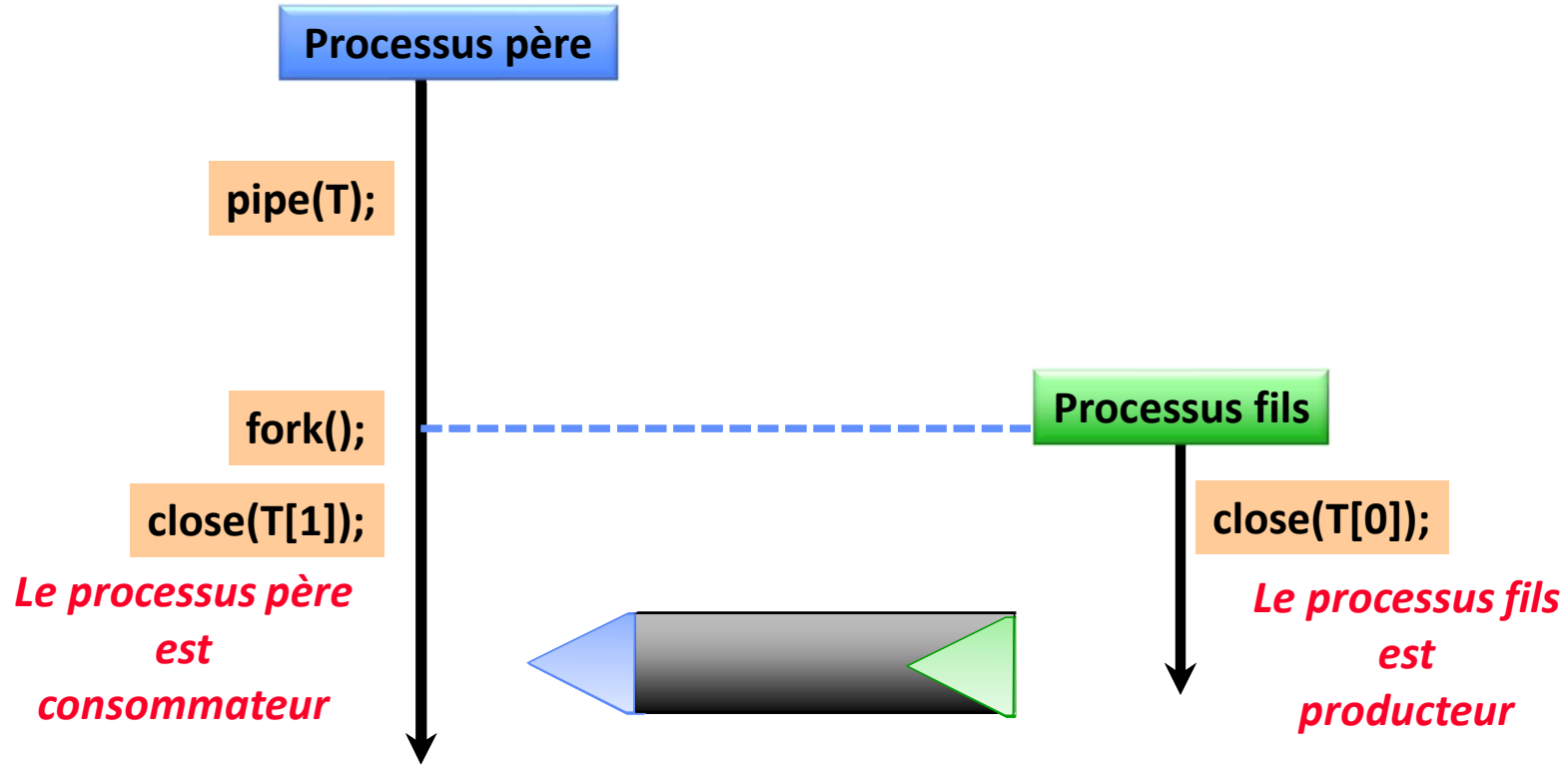
➔ Toujours fermer une entrée d'un tube dès lors qu'elle n'est plus utilisée

Relation producteur/consommateur

Entre deux processus père et fils



Relation producteur/consommateur – Autre illustration



Exemple : Production

```
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <unistd.h>

char msg[] = "Tu as le bonjour de ton fils ";
int main() {
 int tube[2];
 char car;
 printf("*** Début du travail processus principal\n");
 if (pipe(tube) == -1)  {
  perror("Problème création tube");
  exit(99);
 }
 switch (fork())  {
  case -1 : {
   printf("ERREUR: Création processus fils \n");
   exit(98);
  }
  case 0 :  {
   printf("FILS démarre \n");
   close(tube[0]); /* Fils est rédacteur */
   write(tube[1], msg, strlen(msg));
   close(tube[1]);
   printf("FILS se termine\n");
   exit(0);
  }
```

Exemple : Consommation

```c
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <unistd.h>

char msg[] = "Tu as le bonjour de ton fils ";
int main() {
 int tube[2];
 char car;
 printf("*** Début du travail processus principal\n");
 if (pipe(tube) == -1)  {
  perror("Problème création tube");
  exit(99);
 }
 switch (fork())  {
  case -1 : {
   printf("ERREUR: Création processus fils \n");
   exit(98);
  }
  case 0 :  {
   printf("FILS démarre \n");
   close(tube[0]); /* Fils est rédacteur */
   write(tube[1], msg, strlen(msg));
   close(tube[1]);
   printf("FILS se termine\n");
   exit(0);
  }
  default :  {
   break;
  }
 }
 close(tube[1]); /* Père est lecteur */
 printf("*** Le père reçoit \n");
 while (read (tube[0], &car, 1))  {
  printf("%c", car);
 }
 close(tube[0]);
 printf("  \n");
 printf("*** Fin du travail du père \n");
 exit(0);
}
```

Exemple : Production / Consommation – Exécution

```c
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <unistd.h>

char msg[] = "Tu as le bonjour de ton fils ";
int main() {
 int tube[2];
 char car;
 printf("*** Début du travail processus principal\n");
 if (pipe(tube) == -1)  {
  perror("Problème création tube");
  exit(99);
 }
 switch (fork())  {
  case -1 : {
   printf("ERREUR: Création processus fils \n");
   exit(98);
  }
  case 0 :  {
   printf("FILS démarre \n");
   close(tube[0]); /* Fils est rédacteur */
   write(tube[1], msg, strlen(msg));
   close(tube[1]);
   printf("FILS se termine\n");
   exit(0);
  }
  default :  {
   break;
  }
 }
 close(tube[1]); /* Père est lecteur */
 printf("*** Le père reçoit \n");
 while (read (tube[0], &car, 1))  {
  printf("%c", car);
 }
 close(tube[0]);
 printf("  \n");
 printf("*** Fin du travail du père \n");
 exit(0);
}
```

```
$ ./ex_pipe
*** Début du travail processus
principal
*** Le père reçoit
FILS démarre
FILS se termine
Tu as le bonjour de ton fils
*** Fin du travail du père
$
```

Exercice 1

- Écrire une application dans laquelle le processus père lit les caractères au clavier et le processus fils affiche ces mêmes caractères à l'écran de l'utilisateur

- Étape 1 : Fournir un schéma indiquant la manière dont le tube est utilisé par les différents processus

- Étape 2 : Écrire le code de l'application

Lecture et écriture non bloquantes

- Le paramètre commande précise l'action à réaliser sur l'entrée de la table précisée par nomInterne
- Les paramètres supplémentaires vont dépendre de la commande spécifiée
  - F_GETFL : Permet de récupérer les valeurs des flags associés au descripteur de fichier ouvert
  - F_SETFL : Permet de réécrire les valeurs des flags associés au descripteur de fichier ouvert

```
#include <unistd.h>
#include <fcntl.h>

int  fcntl(
   int nomInterne,
   int commande,
   …);
```

- Pour modifier le flag de non blocage :
  - F = fcntl(fd, F_GETFL); /* Récupération des flags */
  - F = F | O_NONBLOCK; /* Positionnement du flag de non blocage */
  - fcntl(fd, F_SETFL, F); /* Réécriture */
  - Avec les tests associés sur les erreurs possibles…

Lecture et écriture non bloquantes

Lecture non bloquante

- **Sémantique de la primitive read après avoir positionné le flag O_NONBLOCK :**
  - Si des processus disposent d'un accès en écriture sur le tube et que le tube est vide, -1 est retourné par la primitive read et errno contient la valeur EAGAIN (sinon, le processus resterait bloqué dans la lecture)

Écriture non bloquante

- **Sémantique de la primitive write après avoir positionné le flag O_NONBLOCK :**
  - La primitive write écrira 0 ou N octets selon les cas, mais le processus ne restera pas bloqué (consulter man pour plus de détails)

Tubes de communication – Exercice 2

- **Programmer la commande**

  compter HIERARCHIE UTILISATEUR

qui réalise le traitement suivant:

  ls -Ral HIERARCHIE | grep UTILISATEUR | wc –l

- **Étape 1 : Fournir un schéma indiquant la manière dont le(s) tube(s) est(sont) utilisé(s) par les différents processus**

- **Étape 2 : Écrire le code de l'application**

Tubes de communication - Exercice 3

- On se propose d'écrire un programme C dans lequel interviennent trois processus :
  - Le processus *père* reçoit sur son entrée standard un flot de caractères. Un caractère alphabétique est transmis au fils *Alphabetiques*. Un caractère chiffre est transmis au fils *Chiffres*. Les autres caractères sont ignorés. Lorsque la fin du flot est détectée (fin de fichier au clavier ou fin d'un fichier disque), le processus père attend la terminaison de ses fils avant de se terminer lui-même
  - Le processus *Alphabetiques* reçoit de son *père* des caractères alphabétiques. Il doit compter le nombre d'occurrences de chaque caractère alphabétique indépendamment de la casse des caractères (3 caractères 'A' et 2 caractères 'a' donnent 5 caractères 'a'). Lorsque la fin du flot est détectée, le processus *Alphabétiques* affichent ses statistiques, puis se termine
  - Le processus *Chiffres* reçoit de son *père* des chiffres. Il doit calculer la somme des chiffres reçus. Lorsque la fin du flot est détectée, le processus *Chiffres* affiche la somme calculée, puis se termine
- On pourra utiliser les fonctions suivantes de la bibliothèque *<ctype.h>* :
  - `int isalpha(char C) ;  /* 1 si alphabétique, 0 sinon */`
  - `int isdigit(char C) ;  /* 1 si numérique, 0 sinon */`
  - `char tolower(char C) ;  /* si C est alphabétique, retourne la minuscule correspondante sinon retourne C */`
- Proposer un schéma de communication entre les processus mettant en jeu un(des) tube(s) de communication avant d'écrire l'application

UNIVERSITÉ TOULOUSE III PAUL SABATIER

Tubes nommés

Tubes nommés – Introduction

- Les tubes nommés associent les propriétés des tubes avec certaines propriétés des fichiers
  - Ils sont de taille limitée
  - Les lectures et écritures sont effectuées suivant un ordre FIFO
  - Les lectures sont destructrices
  - Ils sont utilisables par des processus qui ne sont pas obligatoirement de la même famille

Tubes nommés – Création

Primitive mkfifo

- Crée un tube de nomExterne spécifié
- mode précise les droits d'accès aux trois classes d'usagers
- mkfifo retourne 0 en cas de succès, -1 en cas d'erreur (voir errno dans ce cas)

```
#include <unistd.h>
#include <sys/stat.h>

int  mkfifo(
  const char *nomExterne,
  mode_t      mode);
```

Utilisation

- Tout processus qui veut utiliser un tube nommé doit connaître son nom
- Un processus utilisateur doit ouvrir le tube avant de l'utiliser
- Un processus qui essaie d'ouvrir un tube nommé en lecture sera bloqué jusqu'à ce qu'un autre processus ouvre ce même tube en écriture
- et vice-versa

Exemple de création d'un tube nommé

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/stat.h>

int main(int argc, char**argv) {
 printf("*** Debut du processus creation tube nomme\n");

 if (mkfifo(argv[1], 0600) == -1)  {
  perror("Probleme creation tube nomme ");
  exit(99);
 }
 printf("Le tube nomme %s a ete cree \n",
        argv[1]);
}
```

Création du tube nommé
Son nom externe est en paramètre de la fonction main

Les droits d'accès précisent lecture et écriture pour le propriétaire seul
Ceci est vérifié après exécution

```
$ ./ex_mkfifo TN
*** Debut du processus creation tube nomme
Le tube nomme TN a ete cree
$ ls -l
total 384
prw-------   1 jean-mar  jean-mar      0 Apr  7 17:34 TN
-rwxr-xr-x   1 jean-mar  jean-mar  17380 Apr  7 09:37 ex_exec
-rwxr-xr-x   1 jean-mar  jean-mar  17268 Apr  6 16:46 ex_fork
-rw-r--r--   1 jean-mar  jean-mar    902 Apr  7 14:13 pipe.c
-rw-r--r--   1 jean-mar  jean-mar    490 Apr  7 17:17 tube_n_ecrit.c
-rw-r--r--   1 jean-mar  jean-mar    497 Apr  7 17:21 tube_n_lect.c
…
$
```

Exemple d'écriture dans un tube nommé

```c
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>

int main(int argc, char ** argv) {
  char C;
  int TN;
  printf("*** Debut du processus ecriture dans le tube nomme\n");

  if ((TN = open(argv[1], O_WRONLY, 0)) == -1)  {
    perror("Probleme ouverture tube nomme ");
    exit(99);
  }
  printf("Le tube nomme %s a ete ouvert \n", argv[1]);

  while ( (C = getchar()) != EOF)  {
    write(TN, &C, 1);
  }
  close(TN);
}
```

Ouverture en écriture du tube nommé
Son nom externe est en paramètre
de la fonction main

Lecture de caractères au clavier
et écriture de ces caractères dans le tube

Exemple de lecture dans un tube nommé

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

int main(int argc, char ** argv) {
  char C;
  int TN;
  printf("*** Debut du processus lecture dans le tube nomme\n");

  if ((TN = open(argv[1], O_RDONLY, 0)) == -1)  {
    perror("Probleme ouverture tube nomme ");
    exit(99);
  }
  printf("Le tube nomme %s a ete ouvert \n", argv[1]);
  while (read(TN, &C, 1))  {
    write(1,&C,1);
  }
  printf(  "\n");
  close(TN);
}
```

Ouverture en écriture du tube nommé
Son nom externe est en paramètre
de la fonction main

Lecture des caractères dans le tube
et écriture de ces caractères dans le fichier de
nom interne 1 (écran)

Exécution de l'exemple

```
$ ./ex_tube_e
*** Debut du processus ecriture dans le tube nomme
Probleme ouverture tube nomme : Bad address
$ ./ex_tube_e TN
*** Debut du processus ecriture dans le tube nomme
Probleme ouverture tube nomme : No such file or directory
$
```

Lancement du processus « Producteur »
ERREUR
Le tube n'est pas précisé en paramètre

Lancement du processus « Producteur »
ERREUR
Le tube n'existe pas (le tube créé précédemment a été supprimé)

Exécution de l'exemple

```
$ ./ex_tube_e
*** Debut du processus ecriture dans le tube nomme
Probleme ouverture tube nomme : Bad address
$ ./ex_tube_e TN
*** Debut du processus ecriture dans le tube nomm
Probleme ouverture tube nomme : No such file or d
$
```

Création du tube nommé
Son nom externe est en paramètre
de la fonction main

```
$ ./ex_mkfifo TN
*** Debut du processus creation tube nomme
Le tube nomme TN a ete cree
$ ls -l
total 384
prw-------   1 jean-mar  jean-mar      0 Apr  7 17:34 TN
-rwxr-xr-x   1 jean-mar  jean-mar  17380 Apr  7 09:37
ex_exec
…
$
```

Les droits d'accès précisent
lecture et écriture pour
le propriétaire seul

Exécution de l'exemple

```
$ ./ex_tube_e
*** Debut du processus ecriture dans le tube nomme
Probleme ouverture tube nomme : Ba
$ ./ex_tube_e TN
*** Debut du processus ecriture da
Probleme ouverture tube nomme : No
$
```

```
$ ./ex_mkfifo TN
*** Debut du processus creation tube nomme
Le tube nomme TN a ete cree
$ ls -l
total 384
prw-------    1 jean-mar  jean-mar      0 Apr  7 17:34 TN
wxr-xr-x    1 jean-mar  jean-mar  17380 Apr  7 09:37 ex_exec

$
```

Lancement du processus « Producteur »

```
$ ./ex_tube_e TN
*** Debut du processus ecriture dans le
tube nomme
Le tube nomme TN a ete ouvert
a
sdfg
$
```

Le producteur lit les caractères au clavier et les écrit dans le tube

Lancement du processus « Consommateur »

```
$ ./ex_tube_l TN
*** Debut du processus lecture dans
le tube nomme
Le tube nomme TN a ete ouvert
a
sdfg

$
```

Le consommateur lit les caractères dans le tube et les écrit à l'écran

Exécution de l'exemple

```
$ ./ex_tube_e
*** Debut du processus ecriture dans le tube nomme
Probleme ouverture tube nomme : Ba
$ ./ex_tube_e TN
*** Debut du processus ecriture da
Probleme ouverture tube nomme : No
$
```

```
$ ./ex_mkfifo TN
*** Debut du processus creation tube nomme
Le tube nomme TN a ete cree
$ ls -l
total 384
prw-------   1 jean-mar  jean-mar      0 Apr  7 17:34 TN
-rwxr-xr-x   1 jean-mar  jean-mar  17380 Apr  7 09:37 ex_exec
```

```
$ ./ex_tube_e TN
*** Debut du processus ecriture dans le
tube nomme
Le tube nomme TN a ete ouvert
a
sdfg
$
```

```
$ ./ex_tube_l TN
*** Debut du processus lecture dans
le tube nomme
Le tube nomme TN a ete ouvert
a
sdfg
```

```
$ rm TN
$
```

Le tube nommé est un fichier permanent ;
il doit être détruit après utilisation

UNIVERSITÉ TOULOUSE III PAUL SABATIER

Les signaux UNIX

Les signaux – Introduction

- Un signal est émis à destination d'un processus afin de lui notifier l'occurrence d'un le concernant
- Le concept de signal est à rapprocher du
  - concept d'interruption du niveau matériel
  - concept d'exception du niveau langage
- Il s'agit de la survenue d'un événement conduisant à un traitement spécifique
  - le sous-programme d'interruption
  - le traite-exception

- Le processus doit traiter le signal le plus rapidement possible
  - Un signal qui se produit alors que le processus est est traité
  - Un signal qui se produit alors que le processus est
    - Il sera traité dès que le processus deviendra actif, avant de reprendre l'exécution du code interrompu
    - Une de ce signal est mémorisée

Principes fondamentaux

- Un processus peut associer un à exécuter lorsque le signal doit être traité : le (gestionnaire)
- Dans le cas où aucun traitement n'est prévu par l'utilisateur, un traitement est prévu par le système pour traiter le signal (en général, la)

- Un processus peut un signal
  - Le déclenchement de ce signal n'a alors aucun effet sur le processus
  - Un tel signal est dit « »

- Un processus peut un signal
  - Le traitement de ce signal est alors retardé
  - Un signal qui s'est produit mais n'a pas encore été traité est dit « » ou « » (pending)
- L'ensemble des signaux susceptibles d'être bloqués est défini par un masque de signaux

Exemples de signaux

- Définis dans <signal.h>

SIGINT	Interruption d'un programme par l'utilisateur (touche DELETE ou ^C)
SIGQUIT	Abandon du programme
SIGKILL	Destruction d'un processus (ne peut pas être modifié par le processus)
SIGBUS	Erreur bus (adresse incorrecte)
SIGALRM	Expiration d'un délai
SIGUSR1	Signal utilisateur
SIGUSR2	Signal utilisateur

Obtenir les signaux existants

```
mamachine%kill -l
 1) SIGHUP       2) SIGINT       3) SIGQUIT      4) SIGILL
 5) SIGTRAP      6) SIGABRT      7) SIGEMT       8) SIGFPE
 9) SIGKILL     10) SIGBUS      11) SIGSEGV     12) SIGSYS
13) SIGPIPE     14) SIGALRM     15) SIGTERM     16) SIGURG
17) SIGSTOP     18) SIGTSTP     19) SIGCONT     20) SIGCHLD
21) SIGTTIN     22) SIGTTOU     23) SIGIO       24) SIGXCPU
25) SIGXFSZ     26) SIGVTALRM   27) SIGPROF     28) SIGWINCH
29) SIGINFO     30) SIGUSR1     31) SIGUSR2
mamachine%
```

Le type « ensemble de signaux »

- Définit un ensemble de signaux avec lequel il sera possible d'effectuer certaines actions
- Quelques opérations sur le type sigset_t

```
#include <signal.h>

int sigemptyset (sigset_t *set);

int sigaddset   (sigset_t *set, int sigNum);

int sigdelset   (sigset_t *set, int sigNum);

int sigfillset  (sigset_t *set);

int sigismember (const sigset_t *set, int sigNum);
```

Blocage des signaux

- Un processus peut la prise en compte de signaux lors de l'exécution de certains traitements (section critique par exemple)
  - Dans ce cas, les signaux seront dits « »
  - Un signal masqué sera à son arrivée
- Le des signaux définit l'ensemble des signaux qui peuvent être bloqués à un instant donné
- La primitive sigprocmask installe un masque de signaux (si non NULL)
  - Elle retourne le masque de signaux précédemment associé

```
int sigprocmask (int how, const sigset_t *set, sigset_t *oldset);

/* Valeurs possibles de how (l'action souhaitée) */
/* SIG_SETMASK : nouveau masque                  */
/* SIG_BLOCK   : signaux supplémentaires (union) */
/* SIG_UNBLOCK : signaux à exclure (moins)       */
```

Signaux en attente

- Lorsqu'un masque de signaux est mis en place, l'ensemble des signaux bloqués peut être connu par la primitive sigpending

```
int sigpending (sigset_t *set);
```

Interception d'un signal

- Afin de prendre en compte un signal, il est nécessaire d'associer un traitement à ce signal, ce traitement sera effectué [« trap » du shell]
  - lors de la réception du signal associé si le signal arrive alors que le processus est actif
  - ou dès la reprise d'activité si le signal arrive alors que le processus n'est pas actif
- Méthodes
  - Non portable : la primitive signal()
  - Portable : la primitive sigaction()
- Le signal SIGKILL (9) ne peut pas être intercepté
  - Il permet de toujours pouvoir agir contre un processus
- Le traitement associé peut être
  - Le du signal (en général, destruction du processus)
  - du signal
  - Un défini dans le programme sous la forme d'une fonction C

La primitive sigaction (Posix)

Association permanente

```
int sigaction (int                          signum,
               const struct sigaction *act,
               struct sigaction       *oldact);
```

Utilise une structure décrivant

- l'action à réaliser
- le contexte dans lequel cette action doit être réalisée

```
struct sigaction {
    sighandler_t sa_handler;
    sigset_t     sa_mask;
    int          sa_flags;
};
```

La structure sigaction

- La valeur de sa_handler peut être
  - SIG_IGN : Ignorer le signal
  - SIG_DFL : Traitement par défaut
  - Un pointeur vers une fonction de l'utilisateur
- sa_mask précise le masque des signaux durant le traitement
- sa_flags définit le contexte d'exécution du traitement associé
  - SA_NOCLDSTOP, SA_NOCLDWAIT, SA_RESTART… (cf. signal.h)

```
struct sigaction {
    sighandler_t   sa_handler;
    sigset_t       sa_mask;
    int            sa_flags;
};
```

Interruption par signal – Héritage

Signaux et effets

- Si un signal est reçu durant l'exécution d'une primitive, cette primitive est interrompue et le traitement associé au signal est effectué
  - A la suite de ce traitement, le contrôle repart chez le processus appelant avec -1 comme retour de primitive
  - La variable errno précise la raison de ce retour en erreur en contenant la valeur EINTR
  - Il est alors de la responsabilité du programmeur de l'exécution de la primitive interrompue
  - Modification du comportement possible par SA_RESTART

Signaux et opérations sur les processus

- Un processus fils des associations de signaux effectués par son père
- A la suite d'une commutation d'image (ou recouvrement)
  - Les associations explicites de fonctions utilisateur sont (car le code est remplacé)
  - Les associations avec SIG_DFL et SIG_IGN sont conservées

Exercice 1

- En utilisant la primitive sigaction, écrire un programme qui boucle et qui ne peut être détruit qu'après avoir reçu 3 caractères CTRL_C (signal SIGINT)

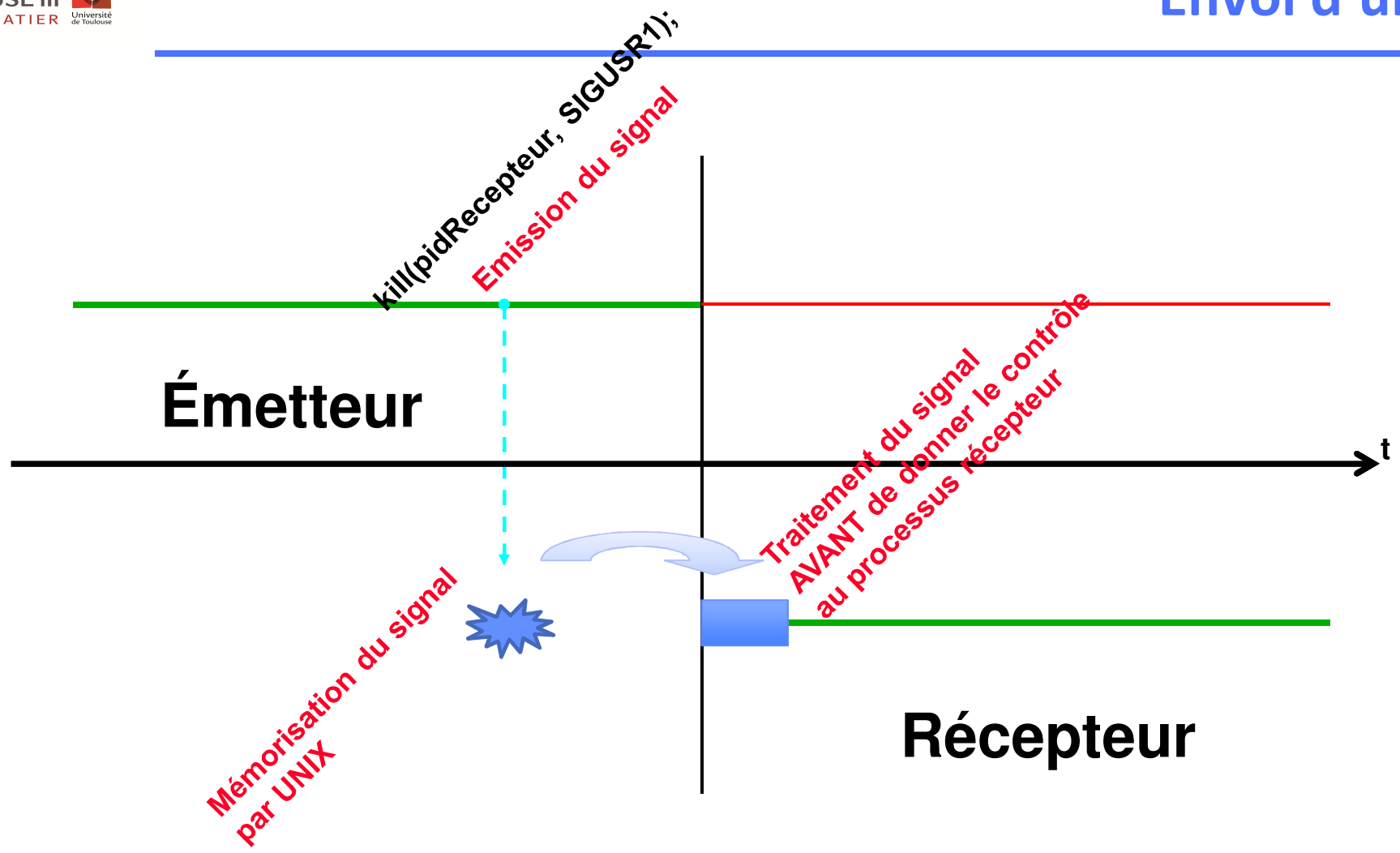
- Attention : Dans les exemples donnés par la suite – et sans doute dans le code donné au tableau – tous les tests des retours des primitives ne sont pas explicitement faits – pour gagner de la place – mais vous devez nécessairement les faire !

Émettre un signal

- La plupart des signaux sont émis par le système
- Un processus peut envoyer un signal
  - à un autre processus appartenant au même propriétaire que celui du processus émetteur
  - à tous les processus appartenant au même groupe de processus que celui de l'émetteur

```
#include <sys/types.h>
#include <signal.h>

int kill (pid_t pid, int sig)

/* Valeurs possibles de pid                  */
/* > 0 : processus ayant ce pid              */
/* = 0 : tous les processus du groupe !      */
```

Envoi d'un signal



Les signaux – Exemple – Processus sans parenté (1)

Émetteur

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <signal.h>
#include <errno.h>

int main (int argc, char *argv[]) {
 int pidMin, pidMax, pidRecepteur;

 if (argc != 3) {
  printf("Usage : %s <Min pid> <Max pid>\n", argv[0]);
  exit(1);
 }

 pidMin = atoi(argv[1]);
 pidMax = atoi(argv[2]);

 for (pidRecepteur = pidMin; pidRecepteur <= pidMax; pidRecepteur++) {
   printf("Emetteur (%d) : J'envoie au recepteur %d le signal %d \n",
                                getpid(), pidRecepteur, SIGINT);
   kill(pidRecepteur, SIGINT);
 }

 printf("Emetteur (%d) : Je me termine\n", getpid());
 return 0;
}
```

Récepteur

```
#include . . .

char c;

void traiterSignal (int sig) {
 printf("*** Réception du signal %d par le processus %d\n", sig, getpid());
 printf("Taper une touche pour me débloquer\n");  c = getchar();
}

int main (int argc, char *argv[]) {
 struct sigaction action;

 printf("Récepteur (%d) : Je démarre\n", getpid());
 for (int i = 0; i < 10000; i++){
   printf("Récepteur (%d) : Je boucle un peu\n", getpid());
 }

 action.sa_handler = traiterSignal;
 sigemptyset(&(action.sa_mask));
 action.sa_flags = 0;
 sigaction(SIGINT, &action, NULL);
 printf("Récepteur (%d) : Je  suis protégé \n", getpid());

 for (i = 0; i < 10000; i++){
   printf("Récepteur (%d) : Je reboucle un peu\n", getpid());
 }
 printf("Récepteur (%d) : Je me bloque en lecture\n", getpid());
 printf("Taper une touche pour me débloquer\n");  c = getchar();

 printf("Récepteur (%d) : Je me termine\n", getpid());
 return 0;
}
```

Les signaux – Exemple – Processus sans parenté (1)

```
#include <st
#include <st
#include <u
#include <si
#include <e

int main (int
 int pidMin

 if (argc != 3) {
  printf("Usage : %s <Min pid> <Max pid>\n", argv[0]);
  exit(1);
 }

 pidMin = atoi(argv[1]);
 pidMax = atoi(argv[2]);

 for (pidRecepteur = pidMin; pidRecepteur <= pidMax; pidRecepteur++) {
   printf("Emetteur (%d) : J'envoie au recepteur %d le signal %d \n",
                                getpid(), pidRecepteur, SIGINT);
   kill(pidRecepteur, SIGINT);
```

```
Recepteur (378) : Je boucle un peu
Recepteur (378) : Je boucle un peu
Recepteur (378) : Je boucle un peu
Recepteur (378) : Je boucle un peu
Recepteur (378) : Je boucle un peu
Recepteur (378) : Je boucle un p
carole@askja:/mnt/c/Users/bernon/Documents/PARTAGE_LINUX/PARTAGELINUX_CLOUD/LANGAGES/C/UNIX/L3_2019-20$
```

```
#include . . .

 printf("Récepteur (%d) : Je démarre\n", getpid());
 for (int i = 0; i < 10000; i++){
    printf("Récepteur (%d) : Je boucle un peu\n", getpid());
 }

 action.sa_handler = traiterSignal;
 sigemptyset(&(action.sa_mask));
 action.sa_flags = 0;
 sigaction(SIGINT, &action, NULL);
 printf("Récepteur (%d) : Je  suis protégé \n", getpid());

 for (i = 0; i < 10000; i++){
```

```
carole@askja:/mnt/c/Users/bernon/Documents/PARTAGE_LINUX/PARTAGELINUX_CLOUD/LANGAGES/C/UNIX/L3_2019-20$ ./emetteur 377 387
Emetteur (379) : J'envoie au recepteur 377 le signal 2
Emetteur (379) : J'envoie au recepteur 378 le signal 2
Emetteur (379) : J'envoie au recepteur 379 le signal 2

carole@askja:/mnt/c/Users/bernon/Documents/PARTAGE_LINUX/PARTAGELINUX_CLOUD/LANGAGES/C/UNIX/L3_2019-20$
```

```
}
```

Les signaux – Exemple – Processus sans parenté (2)

```c
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <signal.h>
#include <errno.h>

int main (int argc, char *argv[]) {
 int pidMin, pidMax, pidRecepteur;

 if (argc != 3) {
  printf("Usage : %s <Min pid> <Max pid>\n", argv[0]);
  exit(1);
 }

 pidMin = atoi(argv[1]);
 pidMax = atoi(argv[2]);

 for (pidRecepteur = pidMin; pidRecepteur <= pidMax; pidRecepteur++) {
   if (pidRecepteur != getpid()){
     printf("Emetteur (%d) : J'envoie au recepteur %d le signal %d \n",
                                  getpid(), pidRecepteur, SIGINT);
     kill(pidRecepteur, SIGINT);
   }
 }

 printf("Emetteur (%d) : Je me termine\n", getpid());
 return 0;
}
```

```c
#include . . .

char c;

void traiterSignal (int sig) {
 printf("*** Réception du signal %d par le processus %d\n", sig, getpid());
 printf("Taper une touche pour me débloquer\n");  c = getchar();
}

int main (int argc, char *argv[]) {
 struct sigaction action;

 printf("Récepteur (%d) : Je démarre\n", getpid());
 for (int i = 0; i < 10000; i++){
   printf("Récepteur (%d) : Je boucle un peu\n", getpid());
 }

 action.sa_handler = traiterSignal;
 sigemptyset(&(action.sa_mask));
 action.sa_flags = 0;
 sigaction(SIGINT, &action, NULL);
 printf("Récepteur (%d) : Je  suis protégé \n", getpid());

 for (i = 0; i < 10000; i++){
   printf("Récepteur (%d) : Je reboucle un peu\n", getpid());
 }
 printf("Récepteur (%d) : Je me bloque en lecture\n", getpid());
 printf("Taper une touche pour me débloquer\n");  c = getchar();

 printf("Récepteur (%d) : Je me termine\n", getpid());
 return 0;
}
```

Les signaux – Exemple – Processus sans parenté (2)

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <signal.h>
#include <errno.h>

int main (int argc, char *
 int pidMin, pidMax, pid

 if (argc != 3) {
  printf("Usage : %s <Min pid> <Max pid>\n", argv[0]);
  exit(1);
 }

 pidMin = atoi(argv[1]);
 pidMax = atoi(argv[2]);

 for (pidRecepteur = pidMin; pidRecepteur <= pidMax; pidRecepteur++) {
   if (pidRecepteur != getpid()){
     printf("Em

     kill(pidRec
   }
 }

 printf("Emette
 return 0;
}
```

```
Recepteur (368) : Je reboucle un peu
Recepteur (368) : Je reboucle un peu
Recepteur (368) : Je reboucle un peu
Recepteur (368) : Je reboucle un peu
Recepteur (368) : Je reboucle un peu
Recepteur (368) : Je reboucle un peu
*** Reception du signal 2 par le processus 368
Taper une touche pour me debloquer
```

```
#include . . .

                                                  processus %d\n", sig, getpid());
                                                  uer\n");  c = getchar();

 printf("Récepteur (%d) : Je démarre\n", getpid());
 for (int i = 0; i < 10000; i++){
    printf("Récepteur (%d) : Je boucle un peu\n", getpid());
 }

 action.sa_handler = traiterSignal;
 sigemptyset(&(action.sa_mask));
 action.sa_flags = 0;
 sigaction(SIGINT, &action, NULL);
```

```
carole@askja:/mnt/c/Users/bernon/Documents/PARTAGE_LINUX/PARTAGELINUX_CLOUD/LANGAGES/C/UNIX/L3_2019-20$ ./emetteur 360 370
Emetteur (369) : J'envoie au recepteur 360 le signal 2
Emetteur (369) : J'envoie au recepteur 361 le signal 2
Emetteur (369) : J'envoie au recepteur 362 le signal 2
Emetteur (369) : J'envoie au recepteur 363 le signal 2
Emetteur (369) : J'envoie au recepteur 364 le signal 2
Emetteur (369) : J'envoie au recepteur 365 le signal 2
Emetteur (369) : J'envoie au recepteur 366 le signal 2
Emetteur (369) : J'envoie au recepteur 367 le signal 2
Emetteur (369) : J'envoie au recepteur 368 le signal 2
Emetteur (369) : J'envoie au recepteur 370 le signal 2
Emetteur (369) : Je me termine
carole@askja:/mnt/c/Users/bernon/Documents/PARTAGE_LINUX/PARTAGELINUX_CLOUD/LANGAGES/C/UNIX/L3_2019-20$
```

Les signaux – Exemple – Processus sans parenté (3)

```
Recepteur (362) : Je reboucle un peu
Recepteur (362) : Je reboucle un peu
Recepteur (362) : Je reboucle un peu
Recepteur (362) : Je reboucle un peu
Recepteur (362) : Je me bloque en lecture
Taper une touche pour me debloquer
*** Reception du signal 2 par le processus 362
Taper une touche pour me debloquer

Recepteur (362) : Je me termine
carole@askja:/mnt/c/Users/bernon/Documents/PARTAGE_LINUX/PARTAGELINUX_CLOUD/LANGAGES/C/UNIX/L3_2019-20$
```

```
carole@askja:/mnt/c/Users/bernon/Documents/PARTAGE_LINUX/PARTAGELINUX_CLOUD/LANGAGES/C/UNIX/L3_2019-20$ ./emetteur 360 370
Emetteur (365) : J'envoie au recepteur 360 le signal 2
Emetteur (365) : J'envoie au recepteur 361 le signal 2
Emetteur (365) : J'envoie au recepteur 362 le signal 2
Emetteur (365) : J'envoie au recepteur 363 le signal 2
Emetteur (365) : J'envoie au recepteur 364 le signal 2
Emetteur (365) : J'envoie au recepteur 366 le signal 2
Emetteur (365) : J'envoie au recepteur 367 le signal 2
Emetteur (365) : J'envoie au recepteur 368 le signal 2
Emetteur (365) : J'envoie au recepteur 369 le signal 2
Emetteur (365) : J'envoie au recepteur 370 le signal 2
Emetteur (365) : Je me termine
carole@askja:/mnt/c/Users/bernon/Documents/PARTAGE_LINUX/PARTAGELINUX_CLOUD/LANGAGES/C/UNIX/L3_2019-20$
```

Les signaux – Exemple – Processus parents

```
. . . . .
#include <signal.h>
#include <errno.h>

int recu = 0;

void traiterSignal (int sig) {
  printf("*** Réception du signal %d par le processus %d\n", sig, getpid());
  recu = 1;
}

void fils (void) {
  struct sigaction action;

  // Se protéger contre SIGINT
  action.sa_handler = traiterSignal;
  sigemptyset(&(action.sa_mask));
  action.sa_flags = 0;
  sigaction(SIGINT, &action, NULL);

  printf("Fils (%d) : Je suis protégé\n", getpid());

  while (1)
    printf("Fils (%d) : Je boucle, reçu = %d\n", getpid(), recu);

  printf("Fils (%d) : Je me termine\n", getpid());
  exit(0);
}
```

```
int main (int argc, char *argv[]) {
  pid_t pidFils;

  printf("Pere (%d) : Je démarre\n", getpid());


  // Créer le fils
  switch(pidFils = fork()) {
    case - 1 : perror("Echec fork");
               exit(1);
    case 0   : fils();
  }

  printf("Père (%d) : J'envoie le signal à mon fils %d \n", getpid(), pidFils);
  kill(pidFils, SIGINT);

  printf("Pere (%d) : Je me termine\n", getpid());
  return 0;
}
```

Les signaux – Exemple – Processus parents

```
.....
#include <signal.h>
#include <errno.h>

int recu = 0;

void traiterSignal (int sig) {
  printf("*** Réception du signal %d par le processus %d\n", sig, getpid());
  recu = 1;
}

void fils (v
  struct sig

  // Se pro
  action.sa
  sigempty
  action.sa
  sigaction

  printf("Fils

  while (1)
    printf("Fi

  printf("Fils
  exit(0);
}
```

```
carole@askja: /mnt/c/Users/bernon/Documents/PARTAGE_LINU
Pere (68) : Je demarre
Pere (68) : J'envoie le signal a mon fils 69
Pere (68) : Je me termine
carole@askja:/mnt/c/Users/bernon/Documents/PArgv[]) {
```

```
printf("Pere (%d) : Je démarre\n", getpid());
```

```
Sélection carole@askja: /mnt/c/Users/bernon/Documents/PARTAGE_LINUX/PARTAGELINUX_CLOUD/LANGAGES/C/UNIX/L3_2019-20
Pere (423) : Je demarre
Pere (423) : J'envoie le signal a mon fils 424
Pere (423) : Je me termine
carole@askja:/mnt/c/Users/bernon/Documents/PARTAGE_LINUX/PARTAGELINUX_CLOUD/LANGAGES/C/UNIX/L3_2019-20$ clear;./pf
Pere (426) : Je demarre
Pere (426) : J'envoie le signal a mon fils 427
Pere (426) : Je me termine
*** Reception du signal 2 par le processus 427
carole@askja:/mnt/c/Users/bernon/Documents/PARTAGE_LINUX/PARTAGELINUX_CLOUD/LANGAGES/C/UNIX/L3_2019-20$ Fils (427) : Je
suis protege
Fils (427) : Je boucle, recu = 1
Fils (427) : Je boucle, recu = 1
Fils (427) : Je boucle, recu = 1
Fils (427) : Je boucle, recu = 1
Fils (427) : Je boucle, recu = 1
Fils (427) : Je boucle, recu = 1
Fils (427) : Je boucle, recu = 1
```

Attendre un signal

- Un processus doit pouvoir
  - attendre l'arrivée d'un événement sans dépenser de l'énergie à ne rien faire
  - ➔ se bloquer explicitement en attente d'un signal

```
#include <unistd.h>

int pause(void);
```

- Déblocage par n'importe quel signal

Attendre un signal

- Un processus doit pouvoir
  - attendre l'arrivée d'un événement sans dépenser de l'énergie à ne rien faire
  - ➔ se bloquer explicitement en attente d'un signal
  - sans être débloqué par d'autres signaux

```
int sigsuspend(const sigset_t *mask);
```

- Installe le masque de signaux fourni en paramètre et attend l'arrivée d'un signal non masqué
- Ces deux actions sont effectuées de manière

- Le masque original est après traitement du signal
- Valeur retournée : toujours -1

On veut être réveillé par LE signal S

```
#include <unistd.h>
#include <signal.h>

int signalRecu = 0;                          /* mise à jour par handler */
…
sigset_t set, oldSet;
int      s;
…
sigprocmask(SIG_SETMASK, NULL, &set);          /* Récupération masque */
sigaddset(&set, s);                      /* Préparation masque temporaire*/
sigprocmask(SIG_SETMASK, &set, &oldSet);              /* Installation */

sigdelset(&set, s);                      /* Préparation masque d'attente */

while (signalRecu == 0)
  sigsuspend(&set);                         /* Attente avec masque adapté */

sigprocmask(SIG_SETMASK, &oldSet, NULL);
```

Exercice 3 – Alternance d'affichage

- Écrire un programme permettant à deux processus d'afficher un message à l'écran en alternance
  - Version 1 : Les deux processus sont parents
  - Version 2 : Les deux processus n'ont pas de lien de parenté

Comportement d'un processus :

```
Boucler {
        Attendre de pouvoir afficher
        Afficher son message
        Donner la permission à l'autre d'afficher
}
```

Exécution souhaitée :

```
. . . .
Processus 1 : Je suis le processus de pid 2442
Processus 2 : Je suis le processus de pid 2546
Processus 1 : Je suis le processus de pid 2442
Processus 2 : Je suis le processus de pid 2546
Processus 1 : Je suis le processus de pid 2442
Processus 2 : Je suis le processus de pid 2546
Processus 1 : Je suis le processus de pid 2442
Processus 2 : Je suis le processus de pid 2546
. . .
. . .
```

Exercice 4 – Premier terminé

- On se propose d'écrire une application dans laquelle on désire effectuer un calcul complexe. Pour cela, deux algorithmes, réalisés par les appels à deux fonctions f1() et f2(), peuvent conduire au résultat souhaité avec une incertitude sur les temps de calcul de chacun de ces algorithmes.
- Un processus père va confier à deux fils le soin d'appliquer chacun un algorithme (les deux calculs seront lancés en parallèle) avant de continuer son traitement.
- Le processus fils qui aura terminé le premier son calcul (mais pas forcément son exécution) avertira le processus père de son succès.
- Le processus père signalera alors au fils qui n'a pas encore terminé son calcul qu'il doit terminer son exécution. Puis, le père continuera son traitement.

UNIVERSITÉ TOULOUSE III PAUL SABATIER

La gestion du temps
UNIX

Fonction sleep (Bibliothèque standard)

```
#include <unistd.h>

unsigned sleep (unsigned int secondes);
```

- Suspend le processus courant durant un délai MAXIMAL défini en secondes

- Attention : le processus est réveillé à l'arrivée du premier événement parmi
  - La fin du délai spécifié
    - Retour : 0
  - L'arrivée d'un signal quelconque non masqué
    - Retour : temps restant
- Ne garantit pas une attente exacte de N secondes !

Exercice 5a – « Métronome » – V1

- **On se propose d'écrire un programme qui envoie périodiquement le signal SIGUSR1 à plusieurs processus**
- **La périodicité en secondes et les numéros des processus sont reçus en paramètres du "main" (sous forme de chaînes de caractères)**

- **Exemple d'utilisation :** ./metronome 5 2034 2035 2036

Enverra toutes les secondes, le signal SIGUSR1 aux processus de pid 2034, 2035 et 2036

Remarque : **sscanf(chaine, "%d", &entier)** où *chaine* est de type char* et *entier* de type int, retourne dans entier la valeur entière correspondant à la chaîne de caractères chaine. On peut utiliser aussi la fonction **int atoi(const char *)**

Signal SIGALRM & Primitive alarm (Posix)

- Certains processus effectuent des ou demandent à être réveillés après un laps de temps précis
- La gestion de ces délais est assurée par le primitive alarm
- A l'issue de ce délai, le processus reçoit le signal SIGALRM
  - Un traitement spécifique a donc dû être prévu pour traiter ce signal
  - Sinon, le traitement par défaut prévu par le système détruira le processus
- La primitive alarm précise une durée en

```
#include <unistd.h>

unsigned int alarm (unsigned int seconds);
```

- Cette primitive retourne le temps qui restait jusqu'au prochain déclenchement du délai

Les différents scénarios

- **Après avoir activé un délai, un processus a le choix entre**
  - **Le traitement synchrone**
    - Attendre l'expiration de ce délai en utilisant la primitive sigsuspend
  - **Le traitement asynchrone**
    - Poursuivre son exécution sachant que la réception du signal SIGALRM interrompra cette exécution et exécutera la fonction de traitement associée
  - **L'annulation ou la modification**
    - Annuler ou modifier le délai initialement spécifié en appelant une nouvelle fois la primitive alarm avec une nouvelle valeur de délai
    - Dans ce cas, la primitive alarm retourne la valeur de délai (en secondes) restant au moment de cette interruption
    - L'annulation de l'alarme est réalisée par l'appel alarm(0) ;

Gestion du temps – Traitement périodique synchrone

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <time.h>
#include <signal.h>
#include <sys/types.h>
int signalRecu = 0;
time_t t;

void timeout (int sigRecu) {
  int i;
  printf("%s \tTraitement alarme en cours\n", (time(&t), ctime(&t)));
  signalRecu = 1;
}

int main (void) {
  sigset_t masqueOriginal, masqueAttente ;
  struct sigaction action;
  int cpt;

  printf("*** Debut du programme\n");
  // Interception du signal d'alarme
  action.sa_handler = timeout;
  sigemptyset(&action.sa_mask);
  action.sa_flags = 0;
  if (sigaction(SIGALRM, &action, NULL) < 0)
    perror("Echec sigaction");
```

```
  // Retarder le traitement eventuel de SIGALRM jusqu'au sigsuspend()
  // On suppose ici que SIGALRM n'est pas dans le masque original
  sigemptyset(&masqueAttente);
  sigaddset(&masqueAttente, SIGALRM);
  sigprocmask(SIG_BLOCK, &masqueAttente, &masqueOriginal);

  for (cpt = 0; cpt < 5; cpt++) {
    // Armer le delai a chaque tour
    alarm(3);
    printf("%s Alarme branchee\n", (time(&t), ctime(&t)));

    signalRecu = 0;
    // Traitement synchrone : on se bloque en attendant
    // Démasquer SIGALRM le temps de le recevoir
    while (!signalRecu)
      sigsuspend(&masqueOriginal);
  }
  // On revient au masque original en demasquant SIGALRM
  sigprocmask(SIG_UNBLOCK, &masqueAttente, NULL);

  printf("%s Programme se termine\n", (time(&t), ctime(&t)));
  return 0;
}
```

Remarque : Opérateur ",", revient à :

```
time(&t);
printf("%s \tTraitement alarme en cours \n", ctime(&t) );
```

Gestion du temps – Traitement périodique synchrone

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <time.h>
#include <signal.h>
#include <sys/types.h>
int signalRecu = 0;
time_t t;

void timeout (int sigRe
  int i;
  printf("%s \tTraitemer
  signalRecu = 1;
}

int main (void) {
  sigset_t masqueOrigin
  struct sigaction action
  int cpt;

  printf("*** Debut du p
  // Interception du sig
  action.sa_handler = ti
  sigemptyset(&action.s
  action.sa_flags = 0;
  if (sigaction(SIGALRM, &action, NULL) < 0)
    perror("Echec sigaction");
```

```
*** Debut du programme
Thu Oct 18 18:37:30 2018
 Alarme branchee
Thu Oct 18 18:37:33 2018
        Traitement alarme en cours
Thu Oct 18 18:37:33 2018
 Alarme branchee
Thu Oct 18 18:37:36 2018
        Traitement alarme en cours
Thu Oct 18 18:37:36 2018
 Alarme branchee
Thu Oct 18 18:37:39 2018
        Traitement alarme en cours
Thu Oct 18 18:37:39 2018
 Alarme branchee
Thu Oct 18 18:37:42 2018
        Traitement alarme en cours
Thu Oct 18 18:37:42 2018
 Alarme branchee
Thu Oct 18 18:37:45 2018
        Traitement alarme en cours
Thu Oct 18 18:37:45 2018
 Programme se termine
```

```
  // Retarder le traitement eventuel de SIGALRM jusqu'au sigsuspend()
  // On suppose ici que SIGALRM n'est pas dans le masque original
  sigemptyset(&masqueAttente);
  sigaddset(&masqueAttente, SIGALRM);
  sigprocmask(SIG_BLOCK, &masqueAttente, &masqueOriginal);

  for (cpt = 0; cpt < 5; cpt++) {
   // Armer le delai a chaque tour
   alarm(3);
   printf("%s Alarme branchee\n", (time(&t), ctime(&t)));

   signalRecu = 0;
   // Traitement synchrone : on se bloque en attendant
   // Démasquer SIGALRM le temps de le recevoir
   while (!signalRecu)
     sigsuspend(&masqueOriginal);
  }
  // On revient au masque original en demasquant SIGALRM
  sigprocmask(SIG_UNBLOCK, &masqueAttente, NULL);

  printf("%s Programme se termine\n", (time(&t), ctime(&t)));
  return 0;
}
```

Remarque : Opérateur ",", revient à :

```
time(&t);
printf("%s \tTraitement alarme en cours \n", ctime(&t) );
```

Gestion du temps – Traitement périodique asynchrone

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <time.h>
#include <signal.h>
#include <sys/types.h>
time_t t;
int cpt = 0;

void timeout (int sigRecu) {
  int i;
  printf("%s \tTraitement alarme en cours\n", (time(&t), ctime(&t)));
  // Armer la periode suivante
  alarm(3);
  cpt++;
  printf("%s \tAlarme (re)branchee (%d)\n", (time(&t), ctime(&t)), cpt);
}

int main (void) {
  struct sigaction action;

  printf("*** Debut du programme\n");
 // Interception du signal d'alarme
  action.sa_handler = timeout;
  sigemptyset(&action.sa_mask);
  action.sa_flags = 0;
  if (sigaction(SIGALRM, &action, NULL) < 0)
    perror("Echec sigaction");
```

```
  // Armer le premier délai du traitement périodique
  printf("%s Fonction connectee\n", (time(&t), ctime(&t)));
  alarm(3);
  printf("%s Alarme branchee\n", (time(&t), ctime(&t)));

  while (cpt < 5) {
    // Effectuer un traitement… si on doit faire un traitement…
    // sinon, traitement synchrone pour ne pas occuper l'UC pour rien

  }

  printf("%s Programme se termine\n", (time(&t), ctime(&t)));
  return 0;
}
```

Gestion du temps – Traitement périodique asynchrone

```c
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <time.h>
#include <signal.h>
#include <sys/types.h>
time_t t;
int cpt = 0;

void timeout (int sigRecu) {
  int i;
  printf("%s \tTraitement alarme en cours\n", (time(&t), ctime(&t)));
  // Armer la periode suivante
  alarm(3);
  cpt++;
  printf("%s \tAlarme (re)branchee (%d)\n", (time(&t), ctime(&t)), cpt);
}

int main (void) {
  struct sigaction action;

  printf("*** Debut du programme\n");
 // Interception du signal d'alarme
  action.sa_handler = timeout;
  sigemptyset(&action.sa_mask);
  action.sa_flags = 0;
  if (sigaction(SIGALRM, &action, NULL) < 0)
    perror("Echec sigaction");
```

```c
// Ar
printf                                                    ));
alarm
printf

while
  // E                                                    ent...
  // s                                                    UC pour rien

}

printf                                                    &t)));
return
}
```

```
*** Debut du programme
Thu Oct 18 17:54:21 2018
 Fonction connectee
Thu Oct 18 17:54:21 2018
 Alarme branchee
Thu Oct 18 17:54:24 2018
        Traitement alarme en cours
Thu Oct 18 17:54:24 2018
        Alarme (re)branchee (1)
Thu Oct 18 17:54:27 2018
        Traitement alarme en cours
Thu Oct 18 17:54:27 2018
        Alarme (re)branchee (2)
Thu Oct 18 17:54:30 2018
        Traitement alarme en cours
Thu Oct 18 17:54:30 2018
        Alarme (re)branchee (3)
Thu Oct 18 17:54:33 2018
        Traitement alarme en cours
Thu Oct 18 17:54:33 2018
        Alarme (re)branchee (4)
Thu Oct 18 17:54:36 2018
        Traitement alarme en cours
Thu Oct 18 17:54:36 2018
        Alarme (re)branchee (5)
Thu Oct 18 17:54:36 2018
 Programme se termine
```

Exercice 5b – « Métronome » - V2

- **On se propose d'écrire une commande qui envoie périodiquement le signal SIGUSR1 à plusieurs processus**
- **La périodicité en secondes et les numéros des processus sont reçus en paramètres du "main" (sous forme de chaînes de caractères)**

- **Le processus qui procède à l'envoi périodique peut réaliser un traitement autre entre deux périodes**

Remarque : **sscanf(chaine, "%d", &entier)** où *chaine* est de type char* et *entier* de type int, retourne dans entier la valeur entière correspondant à la chaîne de caractères chaine. On peut utiliser aussi la fonction **int atoi(const char *)**

Exercice 6 – Exercice de synthèse

- On se propose de simuler, par des processus cycliques, le déroulement d'une course automobile mettant en jeu N « pilotes » et un « contrôle ».
- Un processus Pilote conduit sa voiture en transmettant périodiquement sa position (x, y) au processus Contrôle. Pour modifier sa position après chaque envoi, on suppose qu'il peut utiliser une fonction :
  void progresser(int *x, int *y); qui modifie les valeurs passées de x et y.
  La fonction estArrive(int x, int y); lui permet aussi de savoir s'il a franchi la ligne d'arrivée.
- Le processus Contrôle est chargé de visualiser en permanence les positions reçues des différentes voitures sur un écran de contrôle selon le format :
  Numéro de voiture, Pid du processus associé, position x, position y

1- Faire un schéma explicitant la communication entre les différents types de processus impliqués ainsi que les éventuels signaux utilisés

2- Donner le code d'un processus Pilote

3- Donner le code du processus Contrôle

4- Donner le code du programme principal mettant en place la simulation. Il sera paramétré par le nombre de pilotes et la durée de la période de transmission

UNIVERSITÉ TOULOUSE III PAUL SABATIER

Compléments

Fonction raise

- L'appelant s'envoie un signal à lui-même

```
#include <signal.h>

int raise (int sig)
```

La gestion du temps – Définitions

- Un timer conserve une trace du temps qui s'écoule

- Un cadenceur génère une interruption régulièrement après un intervalle de temps donné
  - Un cadenceur peut être utilisé par le système pour mesurer des durées
  - Un cadenceur peut être utilisé par le scheduler pour mesurer le quantum de temps associé à chaque processus actif

- Un système implémente les multiples timers logiciels utilisés par les processus en utilisant les timers matériels disponibles

Représentation du temps en UNIX

- UNIX calcule l'heure en mémorisant le temps en secondes, écoulé depuis le 1er janvier 1970 0h GMT
- Un processus peut connaître l'heure en utilisant la primitive time

```
#include <time.h>

time_t time (time_t *t);
```

  - Si le pointeur t n'est pas NULL, l'heure est aussi retournée dans la variable pointée

- Questions…
  - Si le type time_t est un int sur 32 bits, à quelle date y-aura-t-il débordement ?
  - avec un unsigned int ?
  - avec un unsigned long sur 64 bits ?

Fonction ctime

- Convertit une heure de type time_t en une chaîne de 26 caractères, plus lisible

```
#include <time.h>

char *ctime (const time_t *t);
```

- La chaîne de caractères retournée par ctime contient un \n à sa fin.

- Exemple :

```
#include <stdio.h>
#include <time.h>
int main(int argc, char*argv[]) {
  time_t t;
  printf("Top a : %sCompris ?\n", (time(&t), ctime(&t)) );
  return 0;
}
```

→ Top a : Thu Sep 27 17:50:22 2018
  Compris ?

Primitives setitimer & getitimer

- Un processus dispose de trois timers qu'il peut activer sélectivement
  - Valeur initiale de déclenchement
  - Valeur de fréquence
  - Précision : microseconde

```
#include <sys/time.h>
int setitimer(int                          which,
              const struct itimerval *newValue,
              struct itimerval       *oldValue);
/* which
   ITIMER_REAL    → SIGALRM
   ITIMER_VIRTUAL → SIGVTALRM
   ITIMER_PROF    → SIGPROF */

int getitimer(int                  which,
              struct itimerval *currentValue);
```

- Retour : temps qui restait jusqu'au prochain déclenchement du délai

```
struct timeval {
  long tv_sec;  /* seconds */
  long tv_usec; /* microseconds */
}
```

```
struct itimerval {
   struct timeval_it interval; /* next value */
   struct timeval_it value;   /* current value */
}
```

Le temps virtuel – La primitive times

- Le temps virtuel d'un processus : temps écoulé dans l'état actif
- Les temps d'exécution sont exprimés en temps virtuel

```
#include <sys/times.h>

clock_t times (struct tms *buf);
```

- La primitive times fournit des informations sur les temps d'exécution du processus courant et de ses fils
- Retourne le nombre de tics
  Diviser par CLK_TCK pour obtenir le nombre de secondes

```
struct tms {
  clock_t tms_utime;  /* Temps CPU utilisateur */
  clock_t tms_stime;  /* Temps CPU système */
  clock_t tms_cutime; /* Temps CPU utilisateur des fils terminus et attendus */
  clock_t tms_cstime; /* Temps CPU système des fils terminus et attendus */
};
```