

INF2610

Noyau d'un système d'exploitation

Révision pour le contrôle périodique

Département de génie informatique et génie logiciel

POLYTECHNIQUE
MONTREAL

AFFILIEE A
L'UNIVERSITE DE MONTREAL

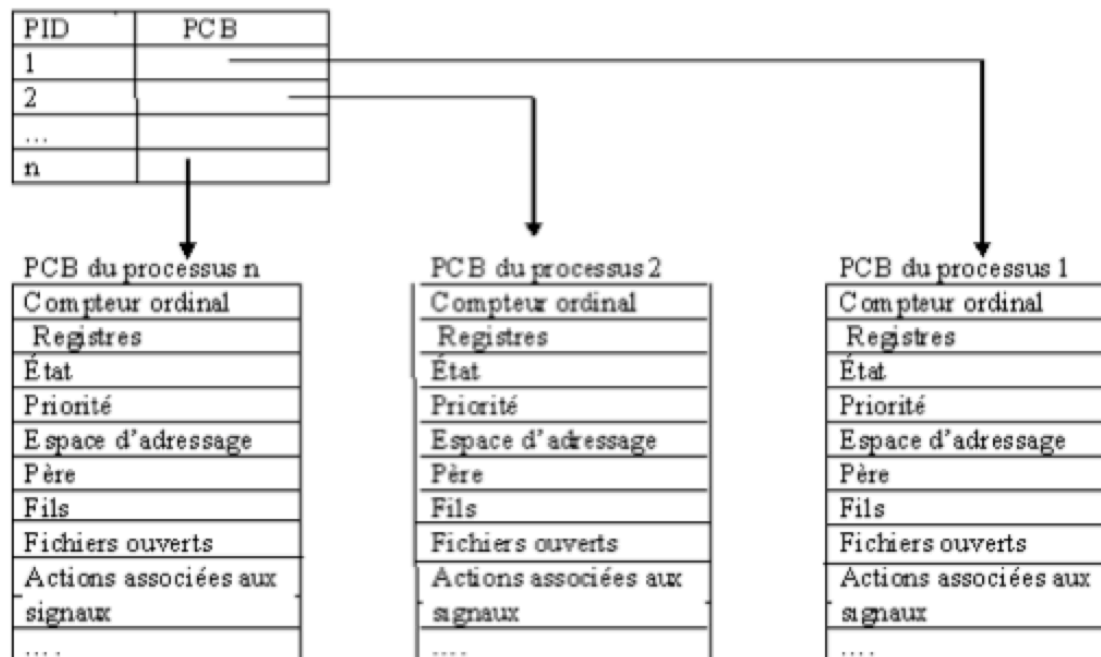


Révision pour le contrôle périodique

- Processus
- Threads
- Communication interprocessus
- Synchronisation
- Exercices

Processus = Programme en cours d'exécution

- Pour un système d'exploitation, un processus = un bloc de contrôle de processus (PCB)

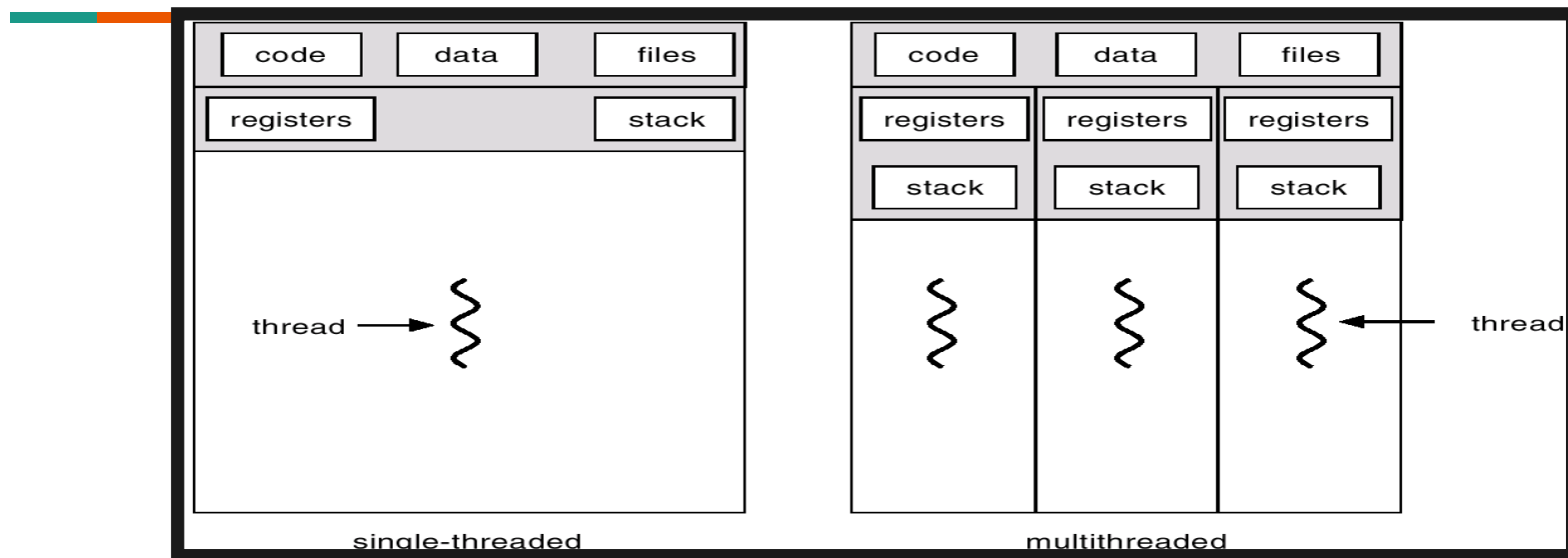


- Compteur ordinal** : adresse de la prochaine instruction à exécuter.
- Registres** : espace réservé à la sauvegarde des contenus des registres lors de la suspension de l'exécution.
- État** : prêt, en exécution, bloqué, etc.
- Priorité** : sert à l'ordonnancement des processus (choisir le plus prioritaire).
- Espace d'adressage** : regroupe le code, les données, la pile d'exécution, etc.
- Fichiers ouverts** : accessibles au processus.
- etc.

Processus (Linux – Unix)

- Création par duplication (selon le principe Copy-On-Write) : `pid_t fork();`
- Le père et le fils s'exécutent en concurrence à partir de l'instruction qui suit l'appel à `fork`.
- Terminaison : `void _exit(int valeur_retour);`
- Changement de code : `int execlp(const char *file, const char *argv,);`
`int execl(const char *path, const char *argv,); ...`
- Attente ou vérification de la terminaison d'un fils : `int wait(int * status);`
`int waitpid (pid_t pid, int * status, int options);`
- `fork` duplique plusieurs structures de données du père comme la table des pages, la table des descripteurs de fichiers, les tables associés aux signaux (masque, gestionnaires mais pas la table des signaux en attente), etc.

Threads



- **Le multithreading permet l'exécution en parallèle ou en pseudo-parallèle de plusieurs parties (fonctions) d'un même processus.**
- Les threads d'un processus partagent toutes les ressources du processus mais chacun a son propre contexte d'exécution.

Threads POSIX (bibliothèque pthread)

- **Création d'un thread :**
`int pthread_create(pthread_t* tid, const pthread_attr_t* attr, void* func, void* arg);`
- **Terminaison :** `void pthread_exit(void* pvalue); int pthread_cancel(pthread_t tid);`
- **Attente de la fin d'un thread :** `int pthread_join(pthread_t tid, void** pstatus)`
- **Pile de nettoyage (fonctions à exécuter lors de la terminaison) :**
`void pthread_cleanup_push(void* func, void* args);`
`void pthread_cleanup_pop(int execute);`

Communication interprocessus



- **Tubes de communication**
- **Signaux**
- **Segments de données partagés**

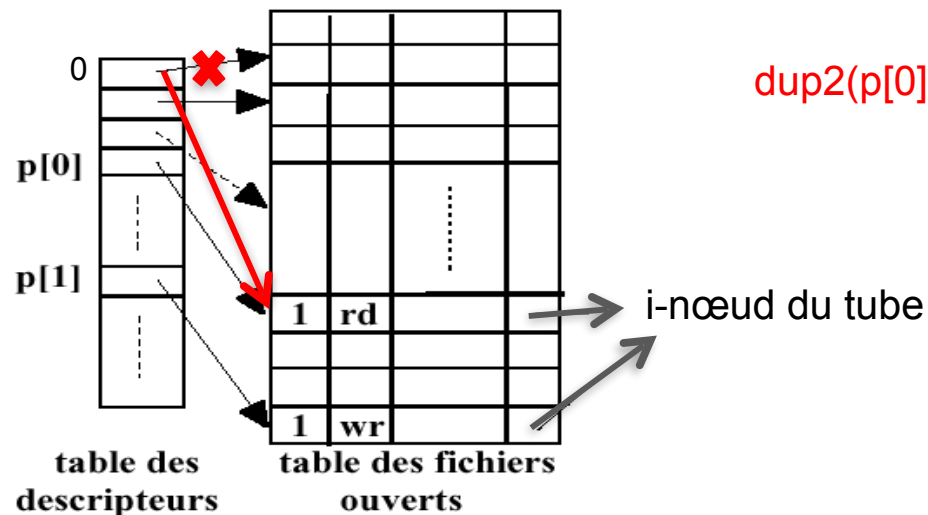
Tubes de communication



- Un tube de communication permet une **communication unidirectionnelle** entre des **processus écrivains et lecteurs**.
- Les lectures et les écritures **sont bloquantes par défaut (appels système read et write)**.
- Les informations déposés dans le tube sont lues selon la politique **FIFO**. La lecture est une **opération destructrice**.
- Chaque tube a **un nombre de lecteurs (nombre de descripteurs en lecture) et un nombre d'écrivains (nombre de descripteurs en écriture)**.
- La fin de fichier est atteinte **lorsque le tube est vide et son nombre d'écrivains est 0**.
- **L'écriture dans un tube rompu (0 lecteurs) par un processus** génère un signal **SIGPIPE** qui est envoyé au processus fauteur.
- Il existe deux types de tubes de communication: **anonymes et nommés**

Tubes anonymes

Pour faire communiquer un processus avec ses descendants
`int p[2];`
`pipe (p);`



Pour associer l'entrée, la sortie standard, ou la sortie erreur au pipe :

`dup2(p[0],0);`,
`dup2(p[1],1);` ou
`dup2(p[1],2);`

Tubes nommés



- Un tube nommé a un nom qui existe dans le système de fichiers dès sa création. Il existera jusqu'à ce qu'il soit supprimé explicitement.
- Il peut être utilisé par des processus dépendants ou indépendants d'un même système.
- La commande « mkfifo » (ou l'appel système mkfifo()) permet de créer un tube nommé.
- Un processus ouvre un tube nommé en lecture ou (exclusif) en écriture.
- Par défaut, il y a une synchronisation sur l'ouverture d'un tube nommé. Cette synchronisation garantit qu'à l'ouverture du tube, il y a au moins un lecteur et un écrivain.

Signaux

- Les systèmes d'exploitation de la famille Unix offrent, au niveau processus, les signaux comme mécanisme de notification d'événements/erreurs et de réactions à ces événements/erreurs. Ils gèrent un ensemble fini de signaux. **Chaque signal :**
 - **a un nom, au moins un numéro,**
 - **a un gestionnaire (handler), et**
 - **est généralement associé à un événement/erreur.**
- Chaque processus a trois tables privées dédiées à la gestion des signaux :
 - **Table des gestionnaires des signaux (TGS)** qui indique pour chaque signal le traitement associé.
 - **Table de bits des signaux en attente (TSA)** qui indique les signaux reçus mais non encore traités.
 - **Table de bits des signaux à différer (MASK)** → masque des signaux du processus.

Signaux

- **Comment envoyer un signal à un processus ?**
`int kill(pid_t pid, int numsignal);`
- **Comment traite-t-il un signal ?** Pour traiter un signal reçu, un processus :
 - suspend temporairement son traitement en cours,
 - réalise celui associé au signal,
 - reprend le traitement suspendu ou se termine.
- Un processus peut associer un autre gestionnaire à un signal, à l'exception des signaux **SIGKILL** et **SIGSTOP** :
 - `sighandler_t signal(int sig, sighandler_t handler);`
 - `int sigaction(int sig, const struct sigaction* action, struct sigaction* oldact);`**Gestionnaire : SIG_DFL, SIG_IGN, ou une fonction du processus.**

Signaux

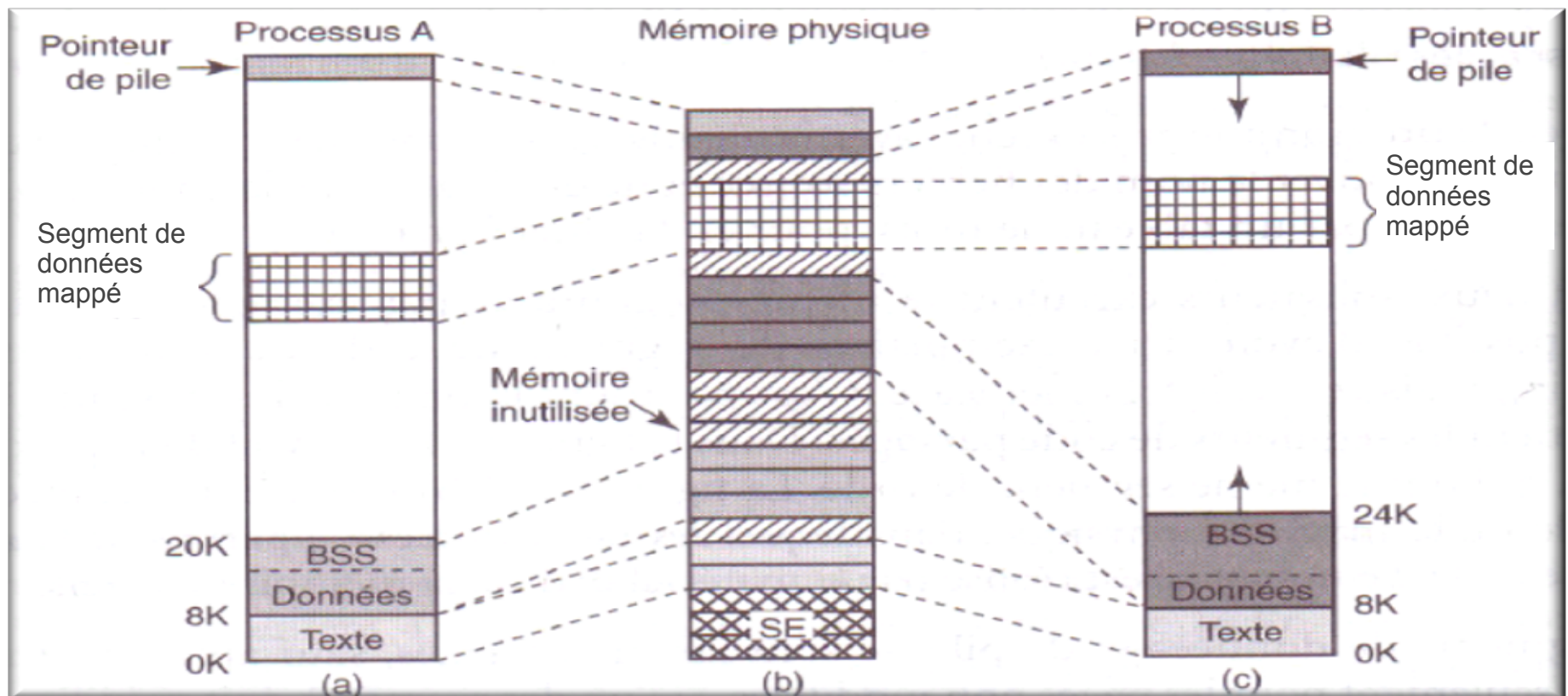


- Il peut différer le traitement d'un signal en l'ajoutant dans le masque des signaux :
int sigprocmask(int how, const sigset_t* set, sigset_t* oldset)
Les signaux **SIGKILL** et **SIGSTOP** ne peuvent pas figurer dans le masque.
- Il peut récupérer les signaux en attente : **int sigpending(sigset_t * set).**
- Il peut se mettre en attente d'un signal :
int pause(), unsigned int sleep(unsigned int secs) ou
int sigsuspend(const sigset_t *sigmask).

Segments de données partagés

- Unix-Linux offrent plusieurs appels système pour **créer, annexer et détacher dynamiquement des segments de données ou fichiers à l'espace d'adressage d'un processus.**
- Les appels système de POSIX pour les segments de données partagés sont dans la librairie `<sys/mman.h>` (man 7 shm_overview) :
 - **shm_open** permet de créer ou de retrouver un segment de données. Il retourne un descripteur de fichier.
 - **mmap** permet d'attacher un segment de données à un processus.
 - **munmap** permet de détacher un segment de données d'un processus.
 - **shm_unlink** permet de supprimer le segment de données lorsqu'il sera détaché de tous les espaces d'adressage.

Segments de données partagés



Synchronisation de processus

- Les accès simultanés en lecture et écriture à un objet partagé peuvent conduire à des résultats incohérents.
- Les accès à l'objet partagé doivent être réalisés en **exclusion mutuelle**.
- Quatre conditions sont nécessaires pour réaliser correctement une exclusion mutuelle :
 1. **Deux processus ne peuvent être en même temps dans leurs sections critiques.**
 2. **Aucune hypothèse ne doit être posée sur les vitesses, le nombre de processeurs, etc.**
 3. **Aucun processus suspendu en dehors de sa section critique ne doit bloquer les autres processus.**
 4. **Aucun processus ne doit attendre trop longtemps avant d'entrer en section critique (attente bornée).**

Comment assurer l'exclusion mutuelle?

- **Masquage des interruptions**
→ dangereuse pour le système.
- **Fonctions atomiques**
→ limitée au cas où chaque section critique est une fonction atomique.
- **Verrous actifs (verrou actif = fonction atomique + attente active)**
→ attente qui consomme de temps CPU.
- **Sémaphores (blocage / déblocage de processus, appels système)**

Verrous actifs – (spinlocks)

Implémentation d'un verrou actif (spin-lock)
en utilisant TSL :

```
int lock = 0;
```

```
Processus P1  
while (1)  
{ while(TSL(lock)!=0);  
  section_critique_P1();  
  lock=0;  
}
```


```
int TSL(int &lock)  
{  
  int r = lock;  
  lock = 1;  
  return r;  
}
```

```
Processus P2  
while (1)  
{ while(TSL(lock)!=0);  
  section_critique_P2();  
  lock=0;  
}
```

```
#include <pthread.h>  
int pthread_spin_lock(pthread_spinlock_t *lock);  
int pthread_spin_unlock(pthread_spinlock_t *lock);
```

La boucle active se répète tant
que le verrou n'est pas libre.

Sémaphores

- Un sémaphore est un compteur entier qui désigne le nombre de jetons d'accès disponibles pour un objet, une ressource, etc.
- Deux opérations sur les sémaphores (qui s'exécutent en exclusion mutuelle) :
 - P(S): si la valeur de $S > 0$ alors décrémenter cette valeur sinon bloquer le processus/thread appelant (en le transférant dans la file d'attente de S).
 - V(S): si la file d'attente de S est vide alors incrémenter la valeur de S sinon transférer un processus/thread de la file d'attente de S vers celle des processus/threads prêts).

Sémaphores

- **Exclusion mutuelle : sémaphore binaire**

Semaphore S = 1 ;

```
Processus P1 :  
{  
    P(S)  
    Section_critique _de_P1() ;  
    V(S) ;  
}
```

```
Processus P2 :  
{  
    P(S)  
    Section_critique_de_P2();  
    V(S) ;  
}
```

- Type `pthread_mutex_t` : `pthread_mutex_init()`, `pthread_mutex_lock()`, `pthread_mutex_unlock()`, etc.
- Type `sem_t` : `sem_init()`, `sem_wait()`, `sem_post()`, etc.

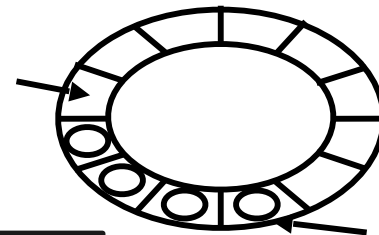
Problème producteurs/consommateurs

```
int tampon [N];  
int ip=0,ic=0;  
Semaphore libre=N, occupe=0, mutex=1;
```

```
Producteur ()  
{  
  while(1)  
  { P(libre) ;  
    P(mutex);  
    produire(tampon, ip);  
    ip = (ip +1)%N;  
    V(mutex);  
    V(occupe);  
  }  
}
```

```
Consommateur ()  
{  
  while(1)  
  { P(occupe);  
    P(mutex);  
    consommer(tampon,ic);  
    ic= (ic+1)% N;  
    V(mutex);  
    V(libre);  
  }  
}
```

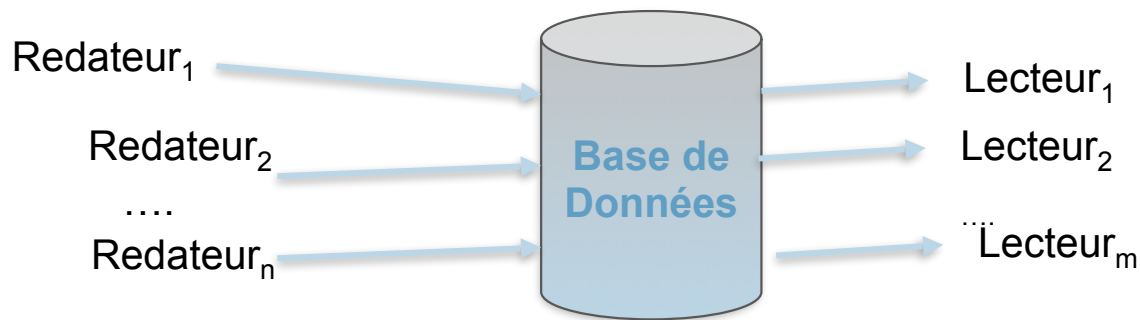
ip
Producteurs



ic
Consommateurs

Lecteurs - Rédacteurs

- Plusieurs processus concurrents partagent une base de données (BD) en lecture/écriture.



- Pour assurer la cohérence des données de la BD, il faut **bloquer tout accès à la BD, si un rédacteur (écrivain) est déjà dans la BD** (accède à la base de données en mode écriture).
 - Par contre, **les lecteurs peuvent accéder à la base de données, en même temps, en mode lecture.**
- un **accès partagé en lecture** et un **accès exclusif en écriture**.

redact : jeton d'accès à la base de données.
mutex : jeton d'accès à la variable partagée NbL.
tour : jeton d'accès pour imposer PAPS.

Lecteurs – Rédacteurs (sans famine)

```
int NbL = 0;  
Semaphore redact=1, mutex=1, tour =1;
```

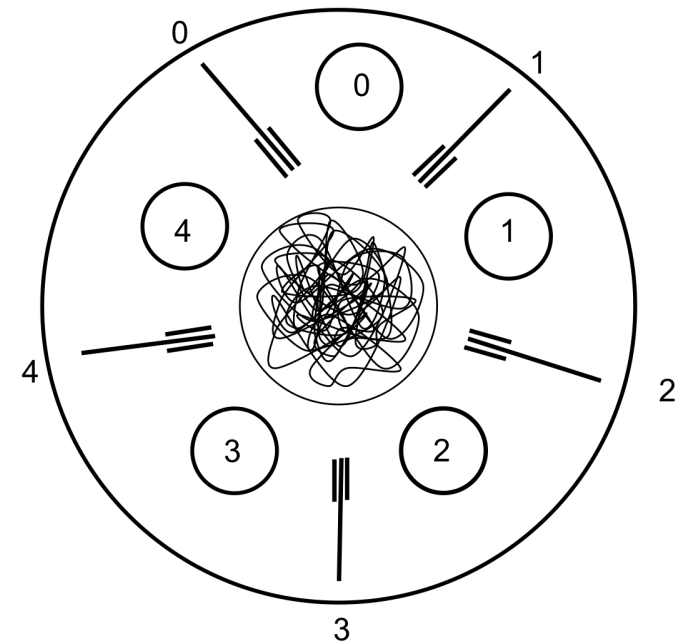
```
Redacteur( )  
{ while(1) {  
    P(tour);  
    P(redact);  
    V(tour);  
    ecrire();  
    V(redact);  
}  
}
```




```
Lecteur() {  
    while(1) { P(tour);  
        P(mutex);  
        if (NbL == 0) P(redact);  
        V(tour);  
        NbL++;  
        V(mutex);  
        lire();  
        P(mutex);  
        NbL--;  
        if(NbL == 0) V(redact);  
        V(mutex);  
    }  
}
```

Problème des philosophes

- Cinq philosophes sont assis autour d'une table. Sur la table, il y a alternativement 5 assiettes et 5 fourchettes.
- Pour manger, un philosophe a besoin de **2 fourchettes qui sont de part et d'autre de son plat**.
- **Si tous les philosophes prennent en même temps chacun une fourchette**, aucun d'entre eux ne pourrait prendre une seconde **→ interblocage**.



Problème des philosophes (1^{ère} solution)



```
#define N 5
#define G (i+1)%N
#define D i
#define libre 1
#define occupe 0

int fourchettes[N] =
    { libre, libre, libre, libre, libre };

sem_t mutex;
```

Pas de détention et attente

→ Risque de famine

```
void* philosophe(void* arg) {
    int i = *(int*) arg;
    while (1) {
        sleep(1); // penser
        sem_wait(&mutex);
        if (fourchettes[G] == libre && fourchettes[D] == libre) {
            fourchettes[G] = occupe;
            fourchettes[D] = occupe;
            sem_post(&mutex);
            printf("le philosophe %d mange\n", i);
            sleep(1); // manger
            printf("le philosophe %d a fini de manger\n", i);
            sem_wait(&mutex);
            // liberer les fourchettes
            fourchettes[G] = libre;
            fourchettes[D] = libre;
            sem_post(&mutex);
        } else sem_post(&mutex);
    }
}
```

Problème des philosophes - Famine

Une solution au problème de famine consiste à :

- associer un sémaphore binaire à chaque fourchette,
- **ordonner les fourchettes** (par exemple, $f_0 < f_1 < f_2 < f_3 < f_4$), et
- imposer à chaque philosophe de demander les deux fourchettes nécessaires pour manger, une à une, **en commençant par la plus petite** (selon l'ordre établi).

```
#define N 5           // nombre de philosophes
#define G (i+1)%N     // fourchette gauche de i
#define D i           // fourchette droite de i
```

```
Semaphore f[N]={1,1,1,1,1};
```

```
Philosophe(int i) {
    while (1) {
        sleep(1);           // penser
        if(G<D) { P(f[G]); P(f[D]);}
        else { P(f[D]); P(f[G]); }
        printf("philosophe %d mange \n", i);
        sleep(1);           // manger
        printf("philosophe %d a fini \n", i);
        V(f[G]); V(f[D]);
    }
}
```



Exercices

Exercice 1 - Processus

Complétez le code ci-dessous pour qu'il réalise le traitement suivant :

- le processus principal incrémente la variable v , crée deux processus fils $F1$ et $F2$, puis se met en attente de la fin de ses fils avant de se terminer.
- Chaque fils Fi , pour $i=1,2$, crée un fils $Fi1$, incrémente la variable v , puis se met en attente de la fin de son fils avant de se terminer.
- Enfin, chaque petit fils $Fi1$, pour $i=1,2$, affiche à l'écran son numéro, celui de son père et la valeur de v , puis se termine. Indiquez la valeur de v affichée à l'écran par chaque petit fils.

```
int v=10 ;  
int main()  
{  
  
    return 0;  
}
```

Exercice 1 - Processus

```
int v=10 ;  
int main() {
```

```
    return 0;}
```

- le processus principal **incrémente la variable v , crée deux processus fils $F1$ et $F2$** , puis se met en attente de la fin de ses fils avant de se terminer.
- Chaque fils Fi , pour $i=1,2$, crée un fils $Fi1$, incrémente la variable v , puis se met en attente de la fin de son fils avant de se terminer.
- Enfin, chaque petit fils $Fi1$, pour $i=1,2$, affiche à l'écran son numéro, celui de son père et la valeur de v , puis se termine.

Indiquez la valeur de v affichée à l'écran par chaque petit fils.

Exercice 1 - Processus - Solution

```
int v=10;
int main() { // PP
    v=v+1;
    if (fork() == 0) { // F1
        if (fork()==0) { // F11
            printf("pid=%d, ppid=%d, v=%d \n",getpid(), getppid(), v);
            _exit(0);
        }
        v=v+1; wait(NULL) ; _exit(0);
    }
    if (fork() == 0) { // F2
        if (fork()==0) { // F21
            printf("pid=%d, ppid=%d, v=%d \n",getpid(), getppid(), v);
            _exit(0);
        }
        v=v+1; wait(NULL) ; _exit(0);
    }
    wait(NULL); wait(NULL);

    return 0;}
```

- le processus principal **incrmente la variable v , crée deux processus fils $F1$ et $F2$** , puis se met en attente de la fin de ses fils avant de se terminer.
- Chaque fils Fi , pour $i=1,2$, crée un fils $Fi1$, incrmente la variable v , puis se met en attente de la fin de son fils avant de se terminer.
- Enfin, chaque petit fils $Fi1$, pour $i=1,2$, affiche à l'écran son numéro, celui de son père et la valeur de v , puis se termine.

Indiquez la valeur de v affichée à l'écran par chaque petit fils.

La valeur de v affichée par F11 est 11.

La valeur de v affichée par F21 est 11.

Exercice 2 - Processus

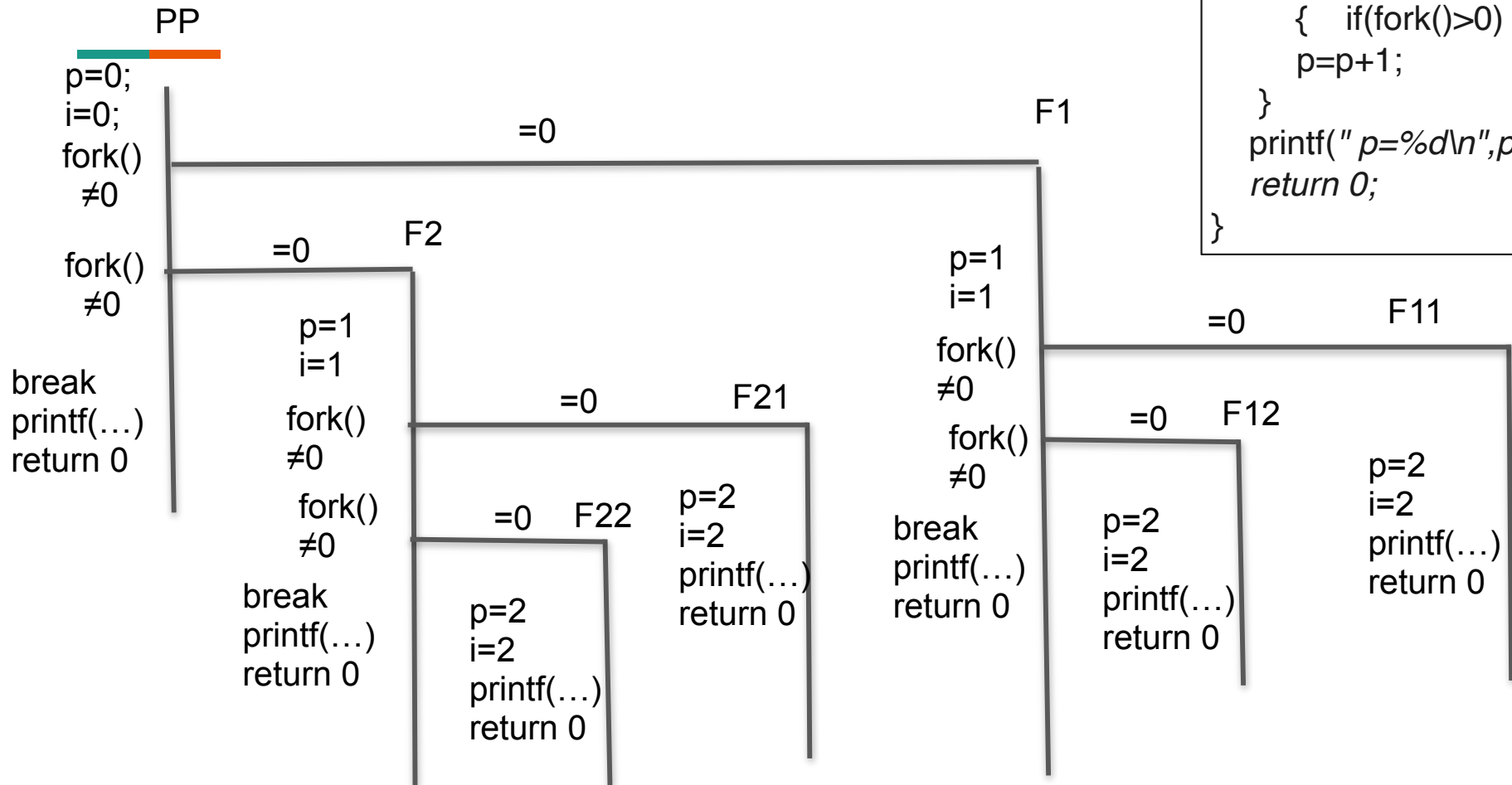
Considérez le programme suivant :

```
const int n=2;
int main(){
    int i,p=0;
    for(i=0; i<n; i++) {
        if(fork()>0)
        {   if(fork()>0) break; }
        p=p+1;
    }
    printf(" p=%d\n",p);
    return 0;
}
```

Supposez que tous les appels système ne retournent pas d'erreur.

1. Donnez la séquence d'instructions exécutées par chaque processus (y compris le processus principal).
2. Donnez l'arborescence des processus créés par ce programme.
3. Donnez la valeur de p affichée par chacun des processus, y compris le processus principal.

Exercise 2 - Processus - Solution 1.1

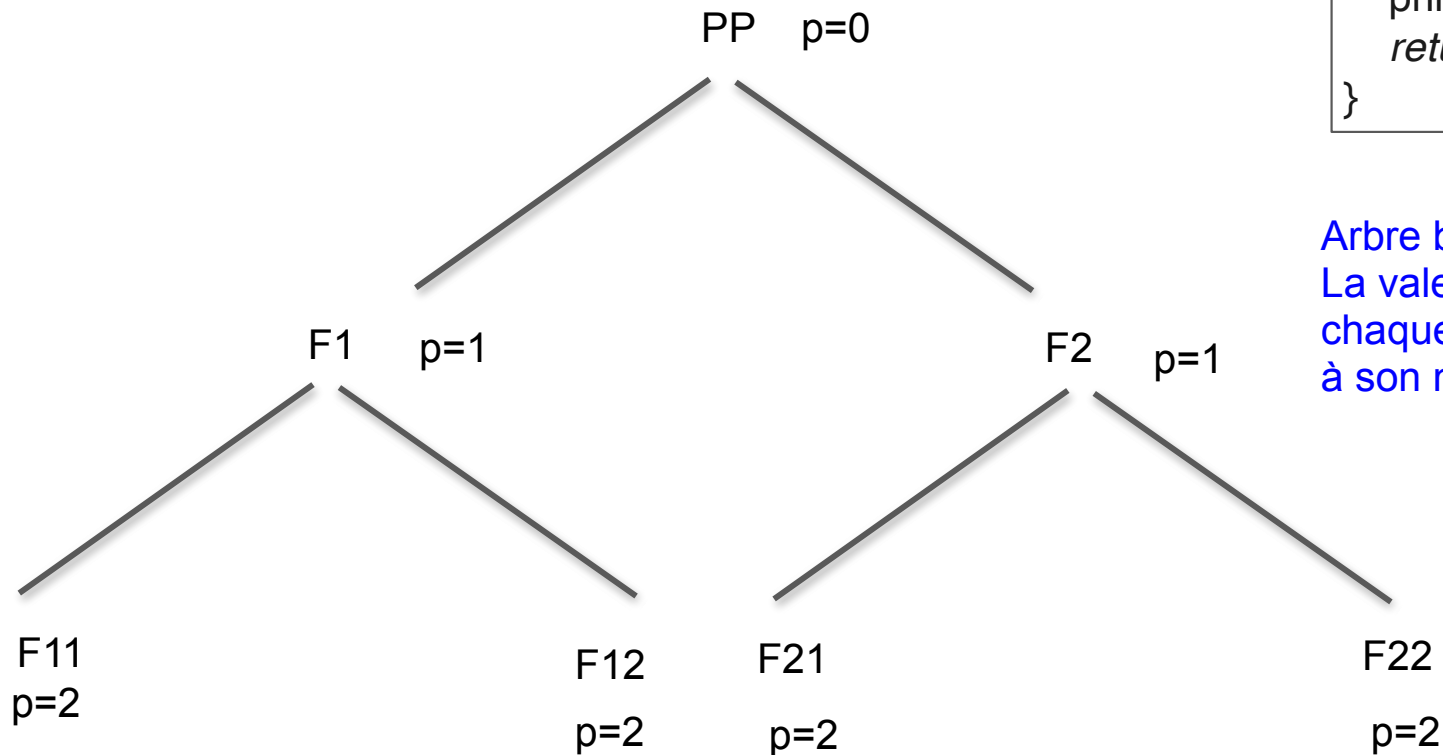


```

const int n=2;
int main(){ int i,p=0;
    for(i=0; i<n; i++) {
        if(fork())>0
        { if(fork())>0 break; }
        p=p+1;
    }
    printf(" p=%d\n",p);
    return 0;
}
  
```


Exercice 2 - Processus - Solutions 1.2 & 1.3

```
const int n=2;
int main(){  int i,p=0;
  for(i=0; i<n; i++) {
    if(fork()>0)
      { if(fork()>0) break; }
    p=p+1;
  }
  printf(" p=%d\n",p);
  return 0;
}
```



Arbre binaire de profondeur 2.
La valeur de p affichée par
chaque processus correspond
à son niveau dans l'arbre

Exercice 2 – Processus

Considérez le programme suivant :

```
const int n=2;
int main(){
    int i,p=0;
    for(i=0; i<n; i++) {
        if(fork()>0)
        {   if(fork()>0) break; }
        p=p+1;
    }
    printf(" p=%d\n",p);
    return 0;
}
```

Supposez que tous les appels système ne retournent pas d'erreur.

4. Complétez le code pour que les sorties standards du processus principal et de ses descendants soient associées au fichier « data.txt » que vous devez ouvrir avant la boucle for.
5. Complétez le code obtenu en 4 pour que les processus sans enfants se transforment pour exécuter le code « *foo.exe* », juste avant l'instruction « *return 0;* ». Cette transformation est réalisée par la fonction « *exec()* ». Le *fichier exécutable* « *foo.exe* » n'a aucun paramètre. Il est supposé existant dans le répertoire courant.
6. Complétez le code obtenu en 5 pour que les processus parents attendent la fin de leurs fils avant de se terminer.


Exercice 2 - Processus – Solutions 1.4, 1.5 & 1.6

```
const int n=2;
int main(){    int i, p=0;
               // pour 1.4
               int fd = open("data.txt", O_WRONLY|O_CREAT);
               dup2(fd,1); close(fd);
               for(i=0;i<n;i++) {
                   if(fork()>0)
                   {   if(fork()>0) break; }
                   p=p+1;
               }
               printf(" p=%d\n",p);
               if (p==n) // pour 1.5
               {   execl("./foo.exe","foo.exe", NULL);
               }
               // pour 1.6
               else while(wait(NULL)>0);
               return 0;
}
```

4. Complétez le code pour que les sorties standards du processus principal et de ses descendants soient associées au fichier « data.txt » que vous devez ouvrir avant la boucle for.
5. Complétez le code obtenu en 4 pour que les processus sans enfants se transforment pour exécuter le code « *foo.exe* », juste avant l’instruction « *return 0;* ». Cette transformation est réalisée par la fonction « *execl* ». Le fichier exécutable « *foo.exe* » n’a aucun paramètre. Il est supposé existant dans le répertoire courant.
6. Complétez le code obtenu en 5 pour que les processus parents attendent la fin de leurs fils avant de se terminer.

Exercice 3 – Synchronisation

Considérez les 3 processus A, B et C concurrents suivants :



A	B	C
a1; a2;	b1; b2;	c1; c2;


Où a_i , b_i et c_i , pour $i=1,2$, sont des actions atomiques.

- 1) Synchronisez, en utilisant les sémaphores, les processus A, B et C de manière à empêcher l'entrelacement des actions de A, B et C.

Semaphore S=1;		
A	B	C
P(S); a1; a2; V(S);	P(S); b1; b2; V(S);	P(S); c1; c2; V(S);

Exercice 3 – Synchronisation

Considérez les 3 processus A, B et C concurrents suivants :



A	B	C
a1; a2;	b1; b2;	c1; c2;


Où a_i , b_i et c_i , pour $i=1,2$, sont des actions atomiques.

2) Utilisez les sémaphores pour forcer l'exécution des actions du processus C après celles de A et B. Les actions de A et B s'exécutent en concurrence (possibilité d'entrelacements des actions de A et B).

Semaphore S=0;		
A	B	C
a1; a2; V(S);	b1; b2; V(S);	P(S); P(S); c1; c2;

Exercice 3 – Synchronisation

Considérez les 3 processus A, B et C concurrents suivants :



A	B	C
a1; a2;	b1; b2;	c1; c2;


Où a_i , b_i et c_i , pour $i=1,2$, sont des actions atomiques.

3) Utilisez les sémaphores pour forcer l'exécution des actions du processus A avant celles de B et C. Les actions de B et C s'exécutent en concurrence (possibilité d'entrelacements des actions de B et C).

Semaphore S=0;		
A	B	C
a1; a2; V(S); V(S);	P(S); b1; b2;	P(S); c1; c2;

Exercice 3 – Synchronisation

Considérez les 3 processus A, B et C concurrents suivants :



A	B	C
a1; a2;	b1; b2;	c1; c2;

Où a_i , b_i et c_i , pour $i=1,2$, sont des actions atomiques.

4) Utilisez les sémaphores pour forcer l'exécution de l'action a_1 du processus A avant celles de B et C et l'action a_2 après celles de B et C. Les actions de B et C s'exécutent en concurrence (possibilité d'entrelacements des actions de B et C).

Semaphore $S_1=0, S_2=0$;		
A	B	C
a1; V(S1);V(S1); P(S2);P(S2); a2;	P(S1); b1; b2; V(S2);	P(S1); c1; c2; V(S2);

Exercice 4 - Redirections

Considérez le programme suivant qui a en entrée trois paramètres : deux fichiers exécutables et un nom de fichier. Ce programme crée deux processus pour exécuter les deux fichiers exécutables.

`$/prog exec1 exec2 fichier`

Complétez le code de manière à exécuter, l'un après l'autre, les deux fichiers exécutables et à rediriger les sorties standards des deux exécutables vers le fichier spécifié comme troisième paramètre. On récupérera ainsi dans ce fichier les résultats du premier exécutable suivis de ceux du deuxième.

Chaque processus doit attendre la fin de ses fils avant de se terminer.

Ne traitez pas les cas d'erreur.

```
int main(int argc, char* argv[])
{
    /*0*/
    if (fork()==0)
    {
        /*1*/
        execlp(argv[1], argv[1],NULL);
        _exit(1);
    }
    /*2*/
    if (fork()==0)
    {
        /*3*/
        execlp(argv[2], argv[2],NULL);
        _exit(1);
    }
    /*4*/
    return 0;
}
```


Exercice 4 - Redirections

Complétez le code de manière à exécuter, l'un après l'autre, les deux fichiers exécutables et à rediriger les sorties standards des deux exécutables vers le fichier spécifié comme troisième paramètre. On récupérera ainsi dans ce fichier les résultats du premier exécutable suivis de ceux du deuxième. Chaque processus doit attendre la fin de ses fils avant de se terminer. Ne traitez pas les cas d'erreur.

```
$/prog exec1 exec2 fichier  
→ argc=4  
→ argv[0] = "./prog"  
→ argv[1] = "exec1"  
→ argv[2] = "exec2"  
→ argv[3] = "fichier"
```

```
int main(int argc, char* argv[])  
{    /*0*/  
    int fd = open(argv[3], O_WRONLY|O_CREAT);  
    if (fork()==0)  
    {    /*1*/  
        dup2(fd,1); close(fd);  
        execlp(argv[1], argv[1], NULL);  
        _exit(1);  
    }  
    /*2*/  
    wait(NULL);  
    if (fork()==0)  
    {    /*3*/  
        dup2(fd,1); close(fd);  
        execlp(argv[2], argv[2], NULL);  
        _exit(1);  
    }  
    /*4*/  
    close(fd); wait(NULL);  
    return 0;  
}
```

Exercice 5 : Synchronisation

Considérez le pseudo-code de la solution au problème des producteurs / consommateurs :

```
const int N = 10;
int tampon [N];
int ip=0, ic=0;
Semaphore libre=N, occupe=0, mutex=1;
```

```
Producteur (int pid) // pid = 0,1, ...
{ while(1)
  {
    P(libre);
    P(mutex);
    tampon[ip] = pid;
    ip = (ip +1)%N;
    V(mutex);
    V(occupe);
  }
}
```

```
Consommateur (int cid) // cid = 0,1, ...
{ int item;
  while(1)
  {
    P(occupe);
    P(mutex);
    item = tampon[ic];
    ic= (ic+1)%N;
    V(mutex);
    V(libre);
  }
}
```

Exercice 5 : Synchronisation

Supposez qu'il y a exactement 3 producteurs (numérotés de 0 à 2) et 3 consommateurs (numérotés 0 à 2). On veut que les producteurs produisent tour à tour (producteur 0, producteur 1, producteur 2, producteur 0, etc.) et que les consommateurs consomment aussi tour à tour (consommateur 0, consommateur 1, consommateur 2, consommateur 0, etc.).

Utilisez les sémaphores pour forcer les productions tour à tour et les consommations tour à tour.

Peut-on éliminer le sémaphore mutex ? Justifiez votre réponse.

Exercice 5 : Synchronisation - Solution

```
const int N = 10;
int tampon [N];
int ip=0, ic=0;
Semaphore libre=N, occupe=0, mutex=1;
Semaphore prod[3]={1,0,0}, cons[3]={1,0,0};
Producteur (int pid) // pid = 0,1, ...
{ while(1) {
    P(prod[pid]);
    P(libre);
    P(mutex);
    tampon[ip] = pid;
    ip = (ip +1)%N;
    V(mutex);
    V(occupe);
    V(prod[(pid+1)%3]);
}
}
```

```
Consommateur (int cid) // cid = 0,1, ...
{ int item;
  while(1) {
    P(cons[cid]);
    P(occupe);
    P(mutex);
    item = tampon[ic];
    ic= (ic+1)%N;
    V(mutex);
    V(libre);
    V(cons[(cid+1)%3]);
  }
}
```

Le sémaphore mutex n'est plus nécessaire car il n'y a ni plus d'accès concurrents aux variables partagées ip et ic.

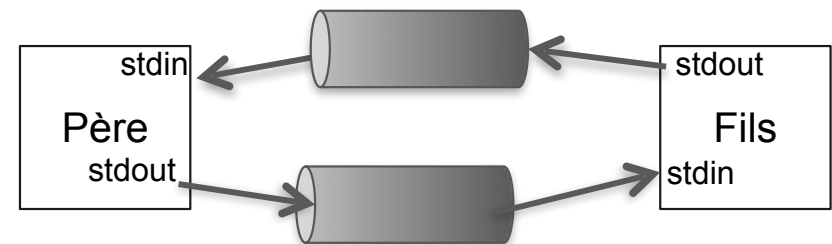
Exercice 6 - tube de communication

Considérez le code suivant :

```
int main ( ) {  
    if(fork()==0) {  
        write(1, "message du fils\n", 16);  
        char A[13]; read(0, A, 13); _exit(0);  
    }  
    char B[16]; read(0, B, 16);  
    write(1, "message reçu\n", 13);  
    wait(NULL);  
    return 0;  
}
```

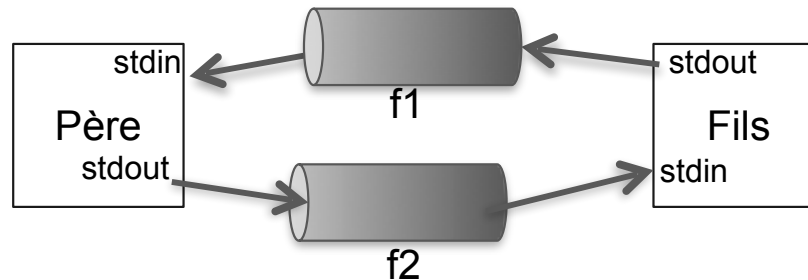
Complétez le code pour créer deux tubes anonymes et faire les ouvertures et redirections nécessaires pour que :

- le père récupère le message *"message du fils\n"* de l'un des deux tubes, et
- le fils récupère le message *"message reçu\n"* de l'autre tube.



Exercice 6 - tube de communication - Solution

```
int main ( ) {  
    int f1[2], f2[2];  
    pipe(f1); pipe(f2);  
    if(fork()==0) { // Fils  
        dup2(f1[1],1); dup2(f2[0],0); close(f1[0]); close(f1[1]); close(f2[0]); close(f2[1]);  
        write(1, "message du fils\n", 16);  
        char A[13]; read(0, A, 13); _exit(0);  
    } // Père  
    dup2(f2[1],1); dup2(f1[0],0); close(f1[0]); close(f1[1]); close(f2[0]); close(f2[1]);  
    char B[16]; read(0, B, 16);  
    write(1, "message reçu\n", 13);  
    wait(NULL);  
    return 0;  
}
```



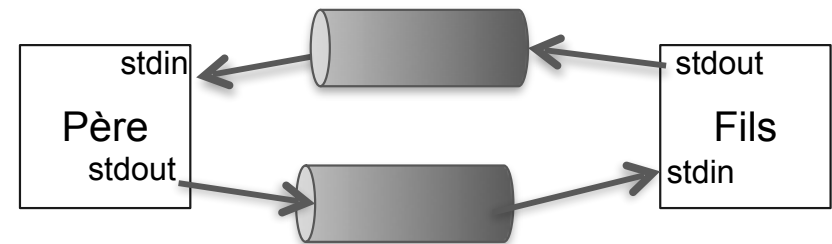
Exercice 7 - tube de communication

Considérez le code suivant :

```
int main ( ) {  
    if(fork()==0) {  
        write(1, "message du fils\n", 16);  
        char A[13]; read(0, A, 13); _exit(0);  
    }  
    char B[16]; read(0, B, 16);  
    write(1, "message reçu\n", 13);  
    wait(NULL);  
    return 0;  
}
```

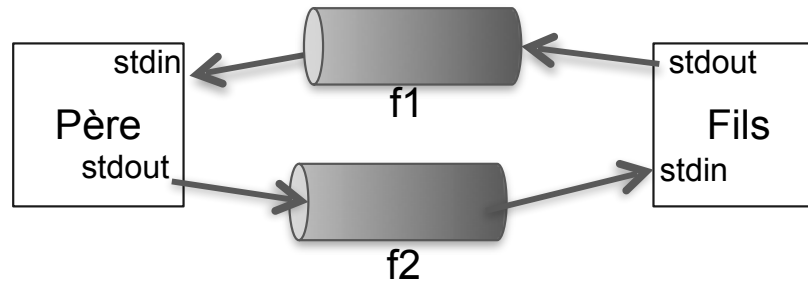
Complétez le code pour créer deux tubes nommés et faire les ouvertures et redirections nécessaires pour que :

- le père récupère le message *"message du fils\n"* de l'un des deux tubes, et
- le fils récupère le message *"message reçu\n"* de l'autre tube.



Exercice 8 - tube de communication - Solution

```
int main ( ) {  
    mkfifo("f1", 0600);  
    mkfifo("f2", 0600);  
    if(fork()==0) {  
        int f1 = open("f1" , O_WRONLY); dup2(f1,1); close(f1);  
        int f2 = open("f2" , O_RDONLY); dup2(f2,0); close(f2);  
        write(1, "message du fils\n", 16);  
        char A[13]; read(0, A, 13); _exit(0);  
    }  
    int f1 = open("f1" , O_RDONLY); dup2(f1,0); close(f1);  
    int f2 = open("f2" , O_WRONLY); dup2(f2,1); close(f2);  
    char B[16]; read(0, B, 16); write(1, "message reçu\n", 13); wait(NULL); return 0;  
}
```




Exercice 8 – Processus

Considérez le code suivant :

```
int gen_alea( int LOW, int HIGH ) //génère aléatoirement un nombre entre LOW et HIGH
{
    srand((unsigned int) clock());
    return rand() % (HIGH - LOW + 1) + LOW;
}
int main ( ) { /*0*/
    return 0;
}
```

1. Complétez le code de la fonction « main » pour créer 5 processus fils. Chaque processus fils exécute la fonction « gen_alea(1,20) » puis se termine. Le processus père se termine après la création de tous ses fils. Vous ne devez pas traiter les cas d'erreur.
2. Est-il possible au processus principal de récupérer le nombre aléatoire généré par chacun de ses fils ? Si oui, complétez/modifiez le code de la fonction « main » afin de récupérer et d'afficher ces nombres à l'écran.

Exercice 8 - Processus - Solution 6.1



```
int gen_alea( int LOW, int HIGH ) //génère aléatoirement un nombre entre LOW et HIGH
{ srand((unsigned int) clock());
  return rand() % (HIGH - LOW + 1) + LOW; }
int main ( ) { /*0*/
    for(int i=0; i<5;i++)
        if(fork()==0) { gen_alea(1,20); _exit(0); }

    return 0;
}
```


1. Complétez le code de la fonction « main » pour créer 5 processus fils. Chaque processus fils exécute la fonction « gen_alea(1,20) » puis se termine. Le processus père se termine après la création de tous ses fils. Vous ne devez pas traiter les cas d'erreur.

Exercice 8 - Processus - Solution 6.2

```
int main ( ) { /*0*/
    for(int i=0; i<5;i++)
        if(fork()==0) { gen_alea(1,20); _exit(0);
            _exit(gen_alea(1,20));
        }
    int s;
    while (wait(&s)>0)
    {   printf(“%d\n”,WEXITSTATUS(s)); }
    return 0;
}
```

2. Est-il possible au processus principal de récupérer le nombre aléatoire généré par chacun de ses fils ? Si oui, complétez/modifiez le code de la fonction « main » afin de récupérer et d’afficher ces nombres à l’écran.

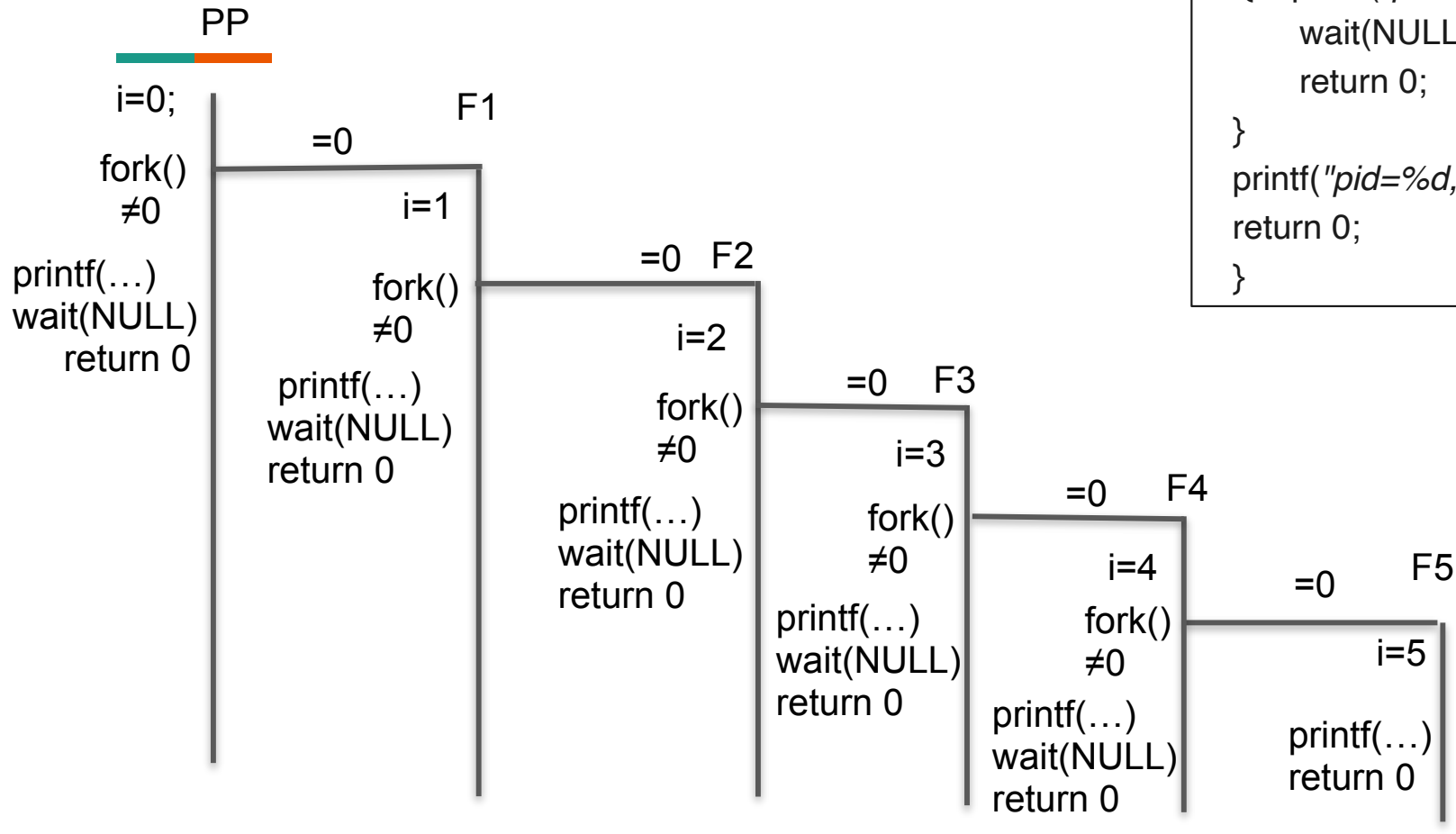
Exercice 9 - Processus



```
int main ( ) { int i;
for(i=0; i<5;i++)
    if(fork()!=0)
    {    printf("pid=%d, i=%d\n", getpid(), i);
        wait(NULL);
        return 0;
    }
printf("pid=%d, i=%d\n", getpid(), i);
return 0;
}
```

Donnez l'arborescence des processus créés par le code ci-dessus.

Exercise 9 - Processus (solution)



```
int main ( ) { int i;
for(i=0; i<5;i++)
    if(fork()!=0)
    { printf("pid=%d, i=%d\n", getpid(),i);
      wait(NULL);
      return 0;
    }
printf("pid=%d, i=%d\n", getpid(),i);
return 0;
}
```

PP
|
F1
|
F2
|
F3
|
F4
|
F5