# Rappel Compilation (avec gcc)

# Compilation d'un fichier

toujours mettre -Wall -Wextra -g
le code doit compilé sans warning

hello.c

```c
#include <stdio.h>

int main(void)
{
    puts("\t Hello World!!");
    return 0;
}
```

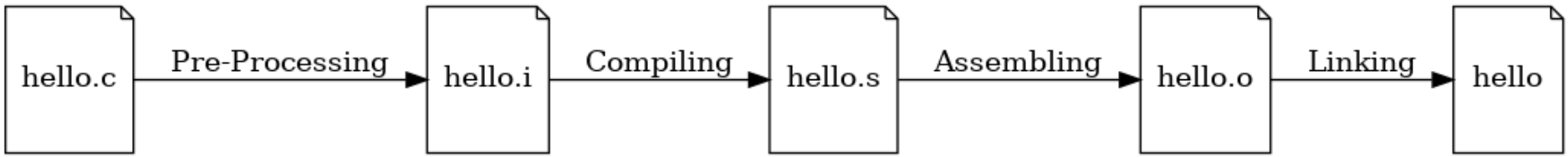
Terminal

```
#Compilation
gcc -Wall -Wextra -g hello.c
-o hello
#Exécution
./hello
    Hello World!!
```

- -Wall: Ajout de warning classique
- -Wextra: Ajout de warning supplémentaire
- -g: Ajout des informations de debug au sein de l'exécutable
- -o: Nom de l'exécutable

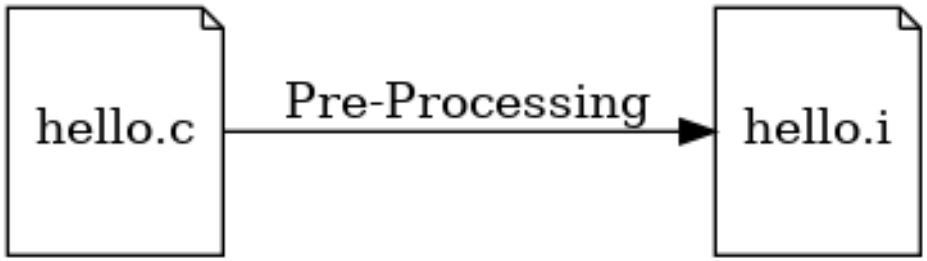
# Les principales étapes de compilation

- Pré-processeur
- Compilation
- Assemblage
- Édition de lien



# Pre-Processing

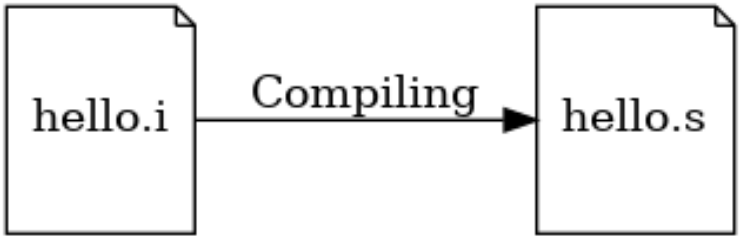
```
gcc -E hello.c > hello.i
```




- Suppression des commentaires (// et /* */)
- Inclusion des fichiers .h dans le fichier .c (#include)
- Traitement des directives qui commencent par un caractère #
  - define
  - ifdef
  - ifndef
  - endif
  - if
  - elif
  - else

# Compiling

```
gcc -S hello.i
```

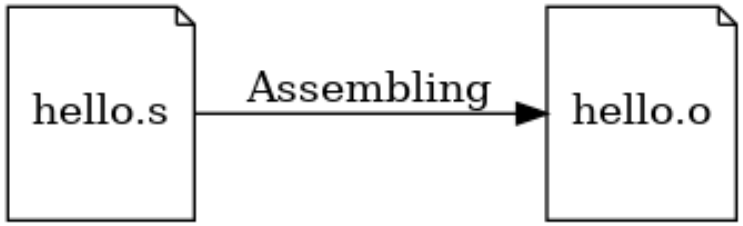



- Vérification de la syntaxe
- Compilation en langage assembleur

```
        .file   "hello.c"
        .text
        .section        .rodata
.LC0:
        .string "\t Hello World!!"
        .text
        .globl  main
        .type   main, @function
main:
.LFB0:
        .cfi_startproc
        pushq   %rbp
        .cfi_def_cfa_offset 16
        .cfi_offset 6, -16
        movq    %rsp, %rbp
        .cfi_def_cfa_register 6
        leaq    .LC0(%rip), %rax
        movq    %rax, %rdi
        call    puts@PLT
        movl    $0, %eax
        popq    %rbp
        .cfi_def_cfa 7, 8
        ret
        .cfi_endproc
.LFE0:
        .size   main, .-main
        .ident  "GCC: (Debian 12.2.0-14) 12.2.0"
        .section        .note.GNU-stack,"",@progbits
```

# Assembling

```
gcc -c hello.s
```

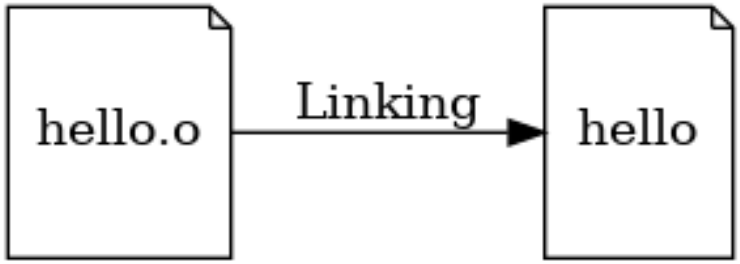



- Transformation en code machine

```
od -x hello.o

0000000 457f 464c 0102 0001 0000 0000 0000 0000
0000020 0001 003e 0001 0000 0000 0000 0000 0000
0000040 0000 0000 0000 0000 0220 0000 0000 0000
0000060 0000 0000 0040 0000 0000 0040 000d 000c
0000100 4855 e589 8d48 0005 0000 4800 c789 00e8
0000120 0000 b800 0000 0000 c35d 2009 6548 6c6c
0000140 206f 6f57 6c72 2164 0021 4700 4343 203a
0000160 4428 6265 6169 206e 3231 322e 302e 312d
0000200 2934 3120 2e32 2e32 0030 0000 0000 0000
0000220 0014 0000 0000 0000 7a01 0052 7801 0110
0000240 0c1b 0807 0190 0000 001c 0000 001c 0000
0000260 0000 0000 001a 0000 4100 100e 0286 0d43
0000300 5506 070c 0008 0000 0000 0000 0000 0000
0000320 0000 0000 0000 0000 0000 0000 0000 0000
0000340 0001 0000 0004 fff1 0000 0000 0000 0000
0000360 0000 0000 0000 0000 0000 0000 0003 0001
0000400 0000 0000 0000 0000 0000 0000 0000 0000
0000420 0000 0000 0003 0005 0000 0000 0000 0000
0000440 0000 0000 0000 0000 0009 0000 0012 0001
0000460 0000 0000 0000 0000 001a 0000 0000 0000
0000500 000e 0000 0010 0000 0000 0000 0000 0000
0000520 0000 0000 0000 0000 6800 6c65 6f6c 632e
0000540 6d00 6961 006e 7570 7374 0000 0000 0000
0000560 0007 0000 0000 0000 0002 0000 0003 0000
0000600 fffc ffff ffff ffff 000f 0000 0000 0000
0000620 0004 0000 0005 0000 fffc ffff ffff ffff
...
```

# Linking

```
gcc hello.o -o hello
```

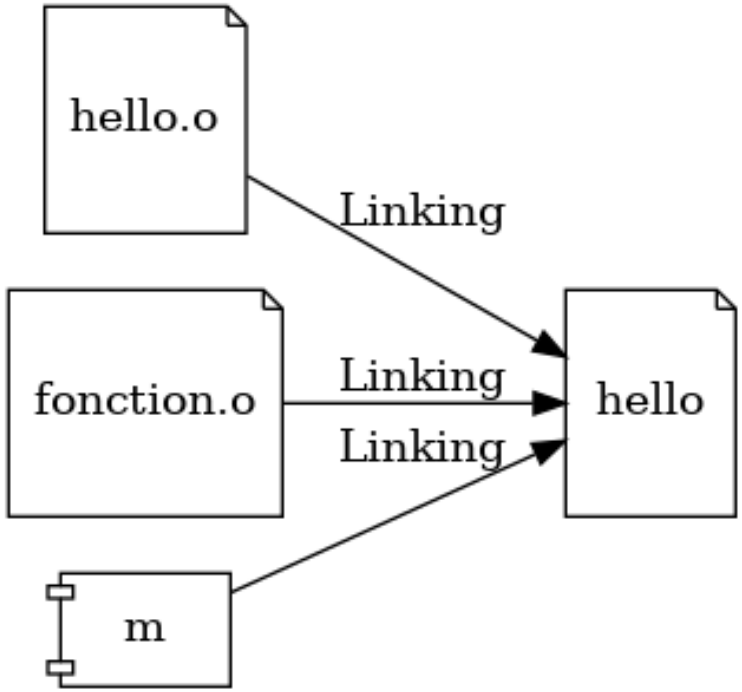



- Fait le lien entre les différents fichiers objet
- Fait le lien avec la bibliothèque C
- Fait le lien avec les bibliothèques (-l)
- Crée l'executable

```
ldd hello
    linux-vdso.so.1 (0x00007fff1c11a000)
    libc.so.6 => /lib/x86_64-linux-gnu/libc.so.6 (0x00007f8b9cba8000)
    /lib64/ld-linux-x86-64.so.2 (0x00007f8b9ce8f000)
```

# Linking avec plusieurs fichiers et bibliothèque

```
gcc hello.o fonction.o  -o hello -lm
```




```
ldd hello
    linux-vdso.so.1 (0x00007fff1c11a000)
    libm.so.6 => /lib/x86_64-linux-gnu/libm.so.6 (0x00007f8b9cd89000)
    libc.so.6 => /lib/x86_64-linux-gnu/libc.so.6 (0x00007f8b9cba8000)
    /lib64/ld-linux-x86-64.so.2 (0x00007f8b9ce8f000)
```

# Arguments d'un programme

# La fonction main

Observez bien le prototype de la fonction main.

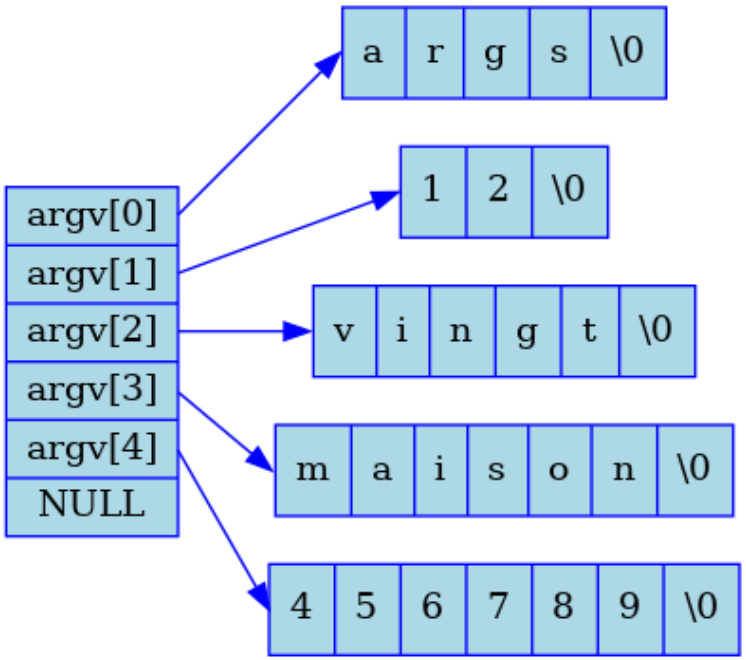
```c
#include <stdio.h>

int main(int argc, char* argv[])
{
    printf("\n%s a reçu %d argument(s):\n", argv[0], argc);
    for (int i = 1; i < argc; ++i)
        puts(argv[i]);
    return 0;
}
```

Si on l'exécute:

```
user@machine $ ./args 12 vingt maison 456789

./args a reçu 5 argument(s):
12
vingt
maison
456789
```

# argc et argv

- argc correspond au nombre d'argument reçu par le programme
- argc a une valeur minimale de 1
- argv contient les pointeurs vers chacun des arguments reçu par le programme
- argv a une taille de argc + 1.
- Le premier élément de argv (argv[0]) est le nom du programme
- Le dernier élément de argv (argv[argc]) contient NULL



# Processus

# Processus/Programme

Qu'est ce qui définit un Programme ?

Qu'est ce qui définit un Processus ?

# Un Processus Unix

- Un Programme en cours d'exécution
- Un identifiant ⇒ PID
- Des entrées/sorties ⇒ Fichier
- Un Parent
- Zero ou Plusieurs Enfants

# Fork

# Prototype de Fork

```
pid_t fork(void);
```

- le processus est cloné
- le nouveau processus (fils) est identique à son père au moment du fork
- le nouveau processus a un PID différent de celui de son père
- Les fichiers ouverts par le père le sont aussi dans le fils.
- Les deux processus continue l'exécution après l'appel à Fork
- la valeur retournée par fork est :
  - le PID du processus fils dans le processus père
  - 0 dans le processus fils
  - -1 si fork n'a pas pu cloner le process (ex: Maximum de process atteint)

il démarre son éxecution juste après l'execution du fork

# Exemple très simple

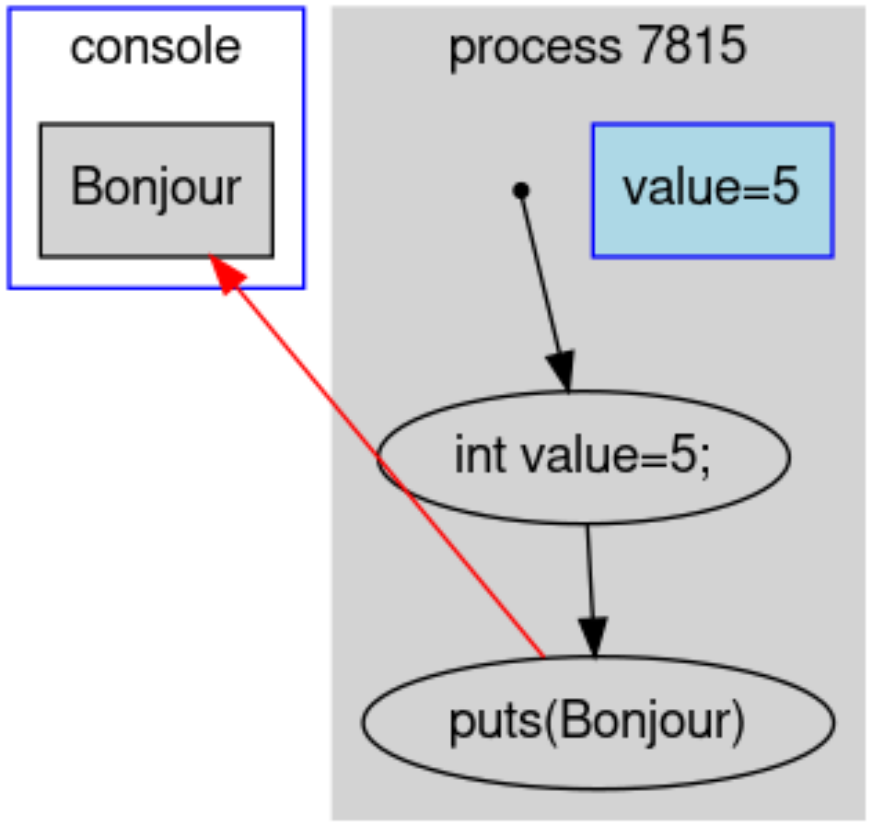
```c
#include <stdio.h>
#include <unistd.h>

int main(void)
{

    int value = 5;
    puts("Bonjour");
    pid_t pid_fils = fork();
    if (pid_fils == 0)
    {
        value+=5;
    }
    printf("%d\n",value);
    return 0;
}
```

Une Sortie possible

```
Bonjour
10
5
```

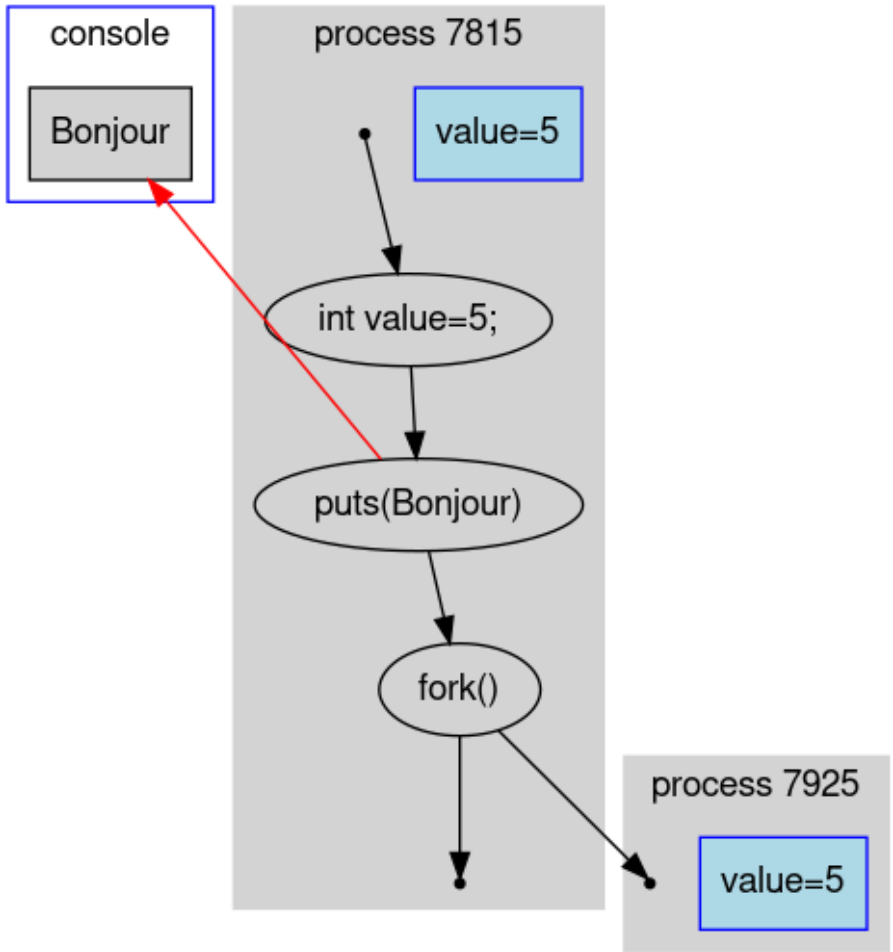
# Exemple très simple: pas à pas

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

int main(void)
{

    int value = 5;
    puts("Bonjour");                    <=
    pid_t pid_fils = fork();
    if (pid_fils == 0)
    {
        value+=5;
    }
    printf("%d\n",value);
    return 0;
}
```



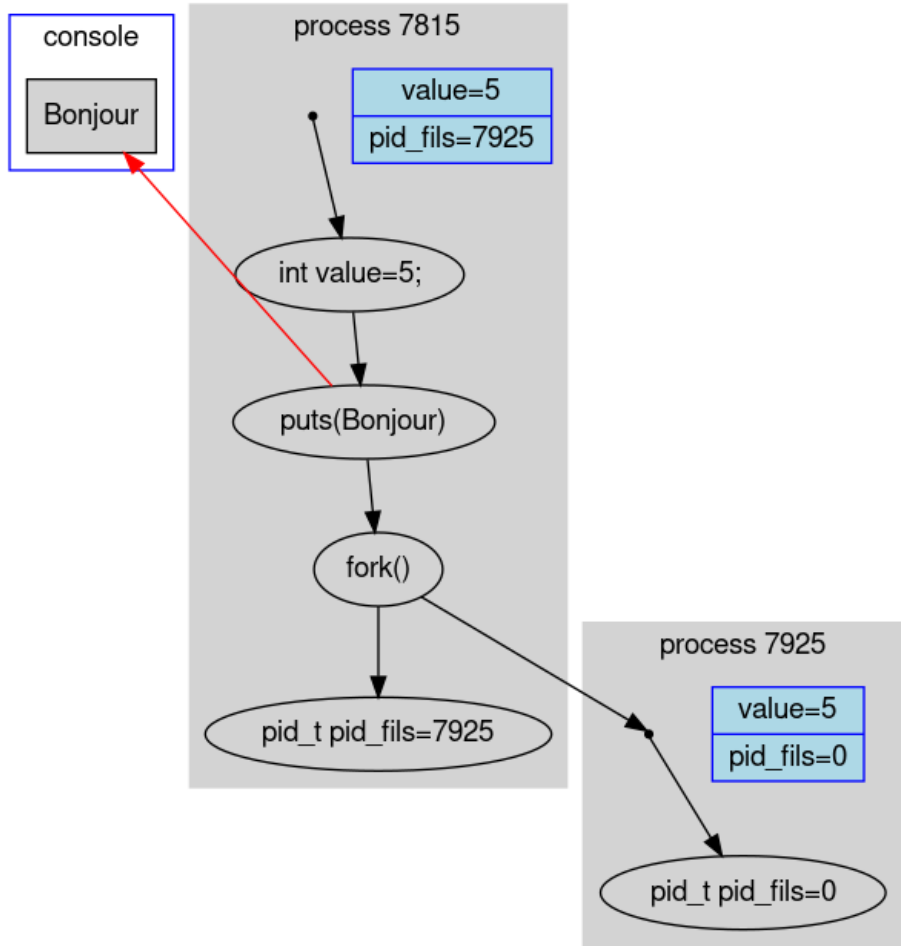
# Exemple très simple: pas à pas

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

int main(void)
{

    int value = 5;
    puts("Bonjour");
    pid_t pid_fils = fork();          <=
    if (pid_fils == 0)
    {
        value+=5;
    }
    printf("%d\n",value);
    return 0;
}
```



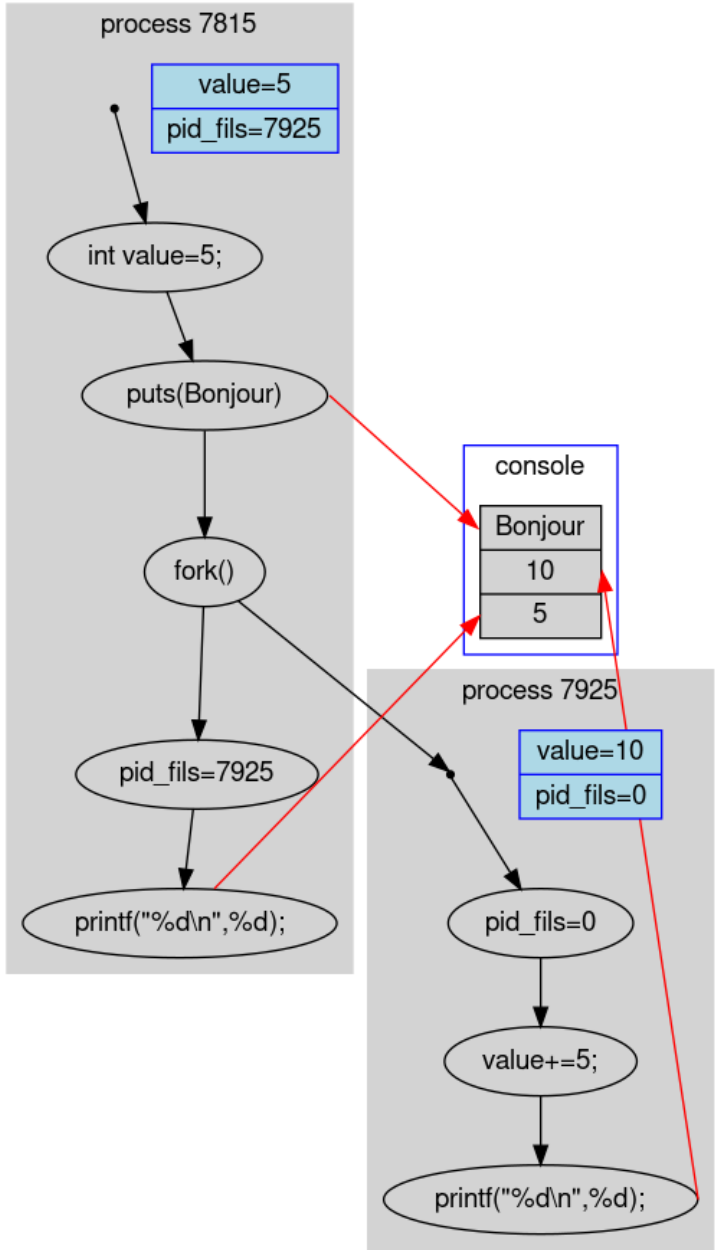
# Exemple très simple: pas à pas

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

int main(void)
{

    int value = 5;
    puts("Bonjour");
    pid_t pid_fils = fork();          <=
    if (pid_fils == 0)
    {
        value+=5;
    }
    printf("%d\n",value);
    return 0;
}
```



# Exemple très simple: pas à pas

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

int main(void)
{

    int value = 5;
    puts("Bonjour");
    pid_t pid_fils = fork();
    if (pid_fils == 0)
    {
        value+=5;
    }
    printf("%d\n",value);                <=
    return 0;
}
```



# Un autre exemple de fork

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

int main(int argc, char* argv[])
{
        int n = 0;
        if (argc != 2)
                exit(-1);
        int success = sscanf(argv[1],"%d",&n);
        if(!success)
                exit(-2);
        for(int i=0; i<n;i++)
        {
                pid_t fils = fork();
                pid_t pere = getppid();
                pid_t moi = getpid();
                printf("%d - Je suis %d, mon père est %d, fork à retourner %d\n",i,moi,pere,fils);
        }
        return 0;
}
```

# Exit et Wait

# Exit et return

- Un Processus se termine lorsque la fonction main se termine
- Un Processus se termine lors de l'exécution de la fonction `void exit(int status)`
- return 0 (dans le main) ou exit(0) (dans n'importe quelle fonction) indique qu'un processus s'est terminé normalement
- Tout autre valeur que 0 représente une erreur.
- Chaque programme est responsable de ses erreurs et de fournir ou non une documentation explicite des erreurs.

# Zombies et Wait

- Un processus terminé est à l'état de zombie tant que son père n'a pas regardé dans quel état il s'est terminé
- L'appel système `pid_t wait(int* status)` récupère l'état d'un processus fils terminé et le stock dans la variable status, wait renvoi le pid du fils terminé.
- On a donc avec wait l'identité du processus terminé et son status.
- Seul le parent direct d'un processus est a même d'utiliser l'appel système wait.
- Si le processus n'a plus aucun enfant à attendre wait renvoie -1.

on ne peut attendre que un fils

# Exemple Wait et Exit

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

int main(void)
{
        int status_fils = 0;
        pid_t fils = fork();
        if (fils == 0)
        {
                srand(getpid());
                int time = rand() % 5 + 1;
                printf("%d %d\n",getpid(), time);
                sleep(time);
                exit(time);
        }
        pid_t fils_termine = wait(&status_fils);
        printf("fils: %d, status: %d\n",fils_termine, WEXITSTATUS(status_fils));
        return 0;
}
```

# Exemple Wait et Exit

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

int main(void)
{
        int status_fils = 0;
        for (int i = 0; i < 3; i++)
        {
                pid_t fils = fork();
                if (fils == 0)
                {
                        srand(getpid());
                        int time = rand() % 5 + 1;
                        printf("%d %d\n",getpid(), time);
                        sleep(time);
                        exit(i);
                }
        }
        for (int i = 0; i < 3; i++)
        {
                pid_t fils_termine = wait(&status_fils);
                printf("fils: %d, status: %d\n",fils_termine, WEXITSTATUS(status_fils));
        }
        return 0;
}
```

# Exemple Zombie

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

int main(void)
{
        int status_fils = 0;
        pid_t fils = fork();
        if (fils == 0)
        {
                srand(getpid());
                int time = rand() % 5 + 1;
                sleep(time);
                exit(time);
        }
        sleep(30);
        return 0;
}
```

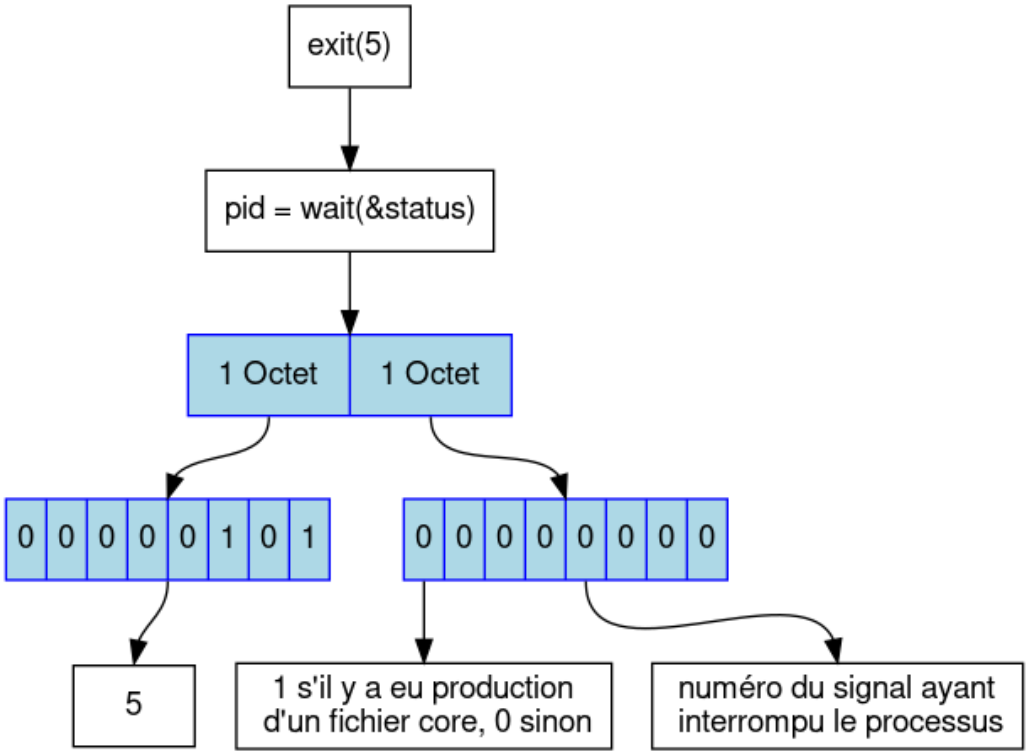
Exécution:

```
user@machine:~$ ./zombie &
[1] 511
user@machine:~$ ps -f
UID          PID  PPID  C STIME TTY          TIME CMD
user         511 14922  0 19:42 pts/12   00:00:00 ./zombie
user         515   511  0 19:42 pts/12   00:00:00 [zombie] <defunct>
```

# Exemple Wait et Exit

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

int main(void)
{
        int n = 5, status_fils = 0, status = 0;
        char c='a';
        for(int i=0; i<n;i++)
        {
                pid_t fils = fork();
                if (fils != 0)
                {
                        c+=i;
                        wait(&status_fils);
                        break;
                }
                status = i;
        }
        sleep(c-'a');
        printf("%c %d %d\n",c,WEXITSTATUS(status_fils));
        return status;
}
```

# Exit Code




- WIFEXITED(status) ⇒ vrai si le processus s'est terminé normalement utilisation d'exit ou return dans main
  - WEXITSTATUS(status) ⇒ renvoi la valeur de retour du processus
- WIFSIGNALED(status) ⇒ vrai si le processus a été terminé par un signal
  - WTERMSIG(status) ⇒ numéro du signal ayant interrompu le processus
  - WCOREDUMP ⇒ Vrai si le processus s'est terminé en produisant une image mémoire (core dump)

# Orphelins

- Si le père d'un processus meurt avant lui. Le processus est orphelin
- Aucun processus hormis init (1) ne peut vivre sans parent
- init adopte donc tout processus orphelin.
- Si on regarde le parent d'un processus orphelin, on retrouve 1

# Exemple d'un orphelin

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

int main(void)
{
        pid_t fils = fork();
        if (fils == 0)
        {
                printf("avant: père:%d pid:%d\n", getppid(), getpid());
                sleep(5);
                printf("après: père:%d pid:%d\n", getppid(), getpid());
        }
        else
        {
                printf("père:%d pid:%d\n", getppid(), getpid());
        }
        sleep(1);
        return 0;
}
```

# Makefile

# Avec un seul fichier: règle implicite

Makefile ou makefile

```
#seul les flags sont définis
CFLAGS=-Wall -Wextra -g
```

La commande make en précisant ce qu'on veut obtenir

```
make hello
cc -Wall -Wextra -g    hello.c   -o hello
```

Automatiquement il utilise les FLAGS défini, il cherche un fichier hello.c et génère hello

```
make
make: *** Pas de cible spécifiée et aucun makefile n'a été trouvé. Arrêt.
```

# Avec un seul fichier: règle un peu plus explicite

Une règle c'est une cible : une ou plusieurs dépendances

```
#On défini toujours les flags
CFLAGS=-Wall -Wextra -g

#définition cible: hello, dépendance: hello.c
hello:hello.c
```

make utilise la première règle du Makefile ici hello si on l'utilise sans argument.

On peut également utiliser un appel a une règle spécifique: make règle

```
make #ou make hello
cc -Wall -Wextra -g    hello.c   -o hello
```

⚠ Pour que l'on reste avec des règles encore très implicites, le fichier c contenant la fonction main et le nom de la cible doivent être le même. hello.c ⇒ hello

# Avec plusieurs fichiers: Mauvaise version

On arrive très rapidement à avoir besoin de séparer le code en plusieurs fichiers. Fichier Header: fonction.h Fichier Source: fonction.c hello.c

```
ls
fonction.c  fonction.h  hello.c  Makefile
```

Chaque fichier source est compilé indépendamment en fichier objet: fonction.o hello.o Ensuite les fichiers sont liés ensemble pour former un exécutable hello

```
CFLAGS=-Wall -Wextra -g

hello:hello.o fonction.o
fonction.o:fonction.c
hello.o:hello.c
```

⚠ fonction.h n'est pas indiqué comme dépendance nulle part, si l'on le modifie sans modifier les autres, le programme ne sera pas recompilé or on a fait des changements

# Avec plusieurs fichiers: Bonne version

On ajoute en dépendance fonction.h pour chaque fichier qui l'utilise

```
CFLAGS=-Wall -Wextra -g

hello:hello.o fonction.o
fonction.o:fonction.c fonction.h
hello.o:hello.c fonction.h

clean:
    rm -rf *.o hello
```

# Plusieurs Exécutables

Ajout d'une règle permettant de générer les différents exécutables Ajout des règles spécifiques à l'exécutable

```
CFLAGS=-Wall -Wextra -g

all: hello fork

hello:hello.o fonction.o
hello.o:hello.c fonction.h

fork: fork.o fonction.o
fork.o:fork.c fonction.h

fonction.o:fonction.c fonction.h

clean:
    rm -rf *.o hello fork
```

# Règles explicites

- clean est une règle explicite
- Une règle c'est cible:dépendances
  - Sur les lignes suivantes des actions précédé d'une tabulation

```
CFLAGS=-Wall -Wextra -g

hello:hello.o fonction.o
    $(CC) $^ -o $@ -lm
fonction.o:fonction.c fonction.h
    $(CC) -c $< -o $@ $(CFLAGS)
hello.o:hello.c fonction.h

clean:
    rm -rf *.o hello
```

- $@ ⇒ cible
- $< ⇒ le premier fichier indiqué en dépendance
- $^ ⇒ l'ensembles des dépendances

# Exec

# Une famille de fonction

```
int execl(const char* path, const char* arg,…, NULL);
int execlp(const char* file, const char* arg,…, NULL);
int execle(const char* path, const char* arg,…,
           NULL,char* const envp[]);


int execv(const char* path, char* const argv[]);
int execvp(const char* file, char* const argv[]);
int execvpe(const char* file, char* const argv[],
           char* const envp[]);
```

# Que font ces fonctions?

- Exécute le programme dont le nom/chemin complet est fourni dans le processus courant
- Le code du processus courant n'existe plus et est remplacé par le programme fourni en argument a exec
- Les fichiers ouverts avant l'appel a exec sont toujours ouverts pour le processus remplaçant
- Elle est toujours utilisée en combinaison avec fork

# Exemple simple execlp

```c
#include <stdio.h>
#include <unistd.h>
#include <sys/wait.h>

int main(void)
{
        pid_t fils = fork();
        if (fils == 0)
        {
                int error = execlp("ls","ls","-l",NULL);
                if (error == -1)
                {
                        perror("exec fail");
                        exit(-1);
                }

        }
        wait(NULL);
        return 0;
}
```

# Exemple simple execvp

```c
#include <stdio.h>
#include <unistd.h>
#include <sys/wait.h>

int main(void)
{
        pid_t fils = fork();
        if (fils == 0)
        {
                char* argv[3] = {"ls","-l",NULL}
                int error = execvp("ls",argv);
                if (error == -1)
                {
                        perror("exec fail");
                        exit(-1);
                }

        }
        wait(NULL);
        return 0;
}
```

# execv* ou execl*

```c
int execl(const char* path, const char* arg,…, NULL);
int execlp(const char* file, const char* arg,…, NULL);
int execle(const char* path, const char* arg,…, NULL,char* const envp[]);

int execv(const char* path, char* const argv[]);
int execvp(const char* file, char* const argv[]);
int execvpe(const char* file, char* const argv[], char* const envp[]);
```

- Deux familles:
  - execl* ⇒ arguments sous forme de liste terminée par NULL
  - execv* ⇒ arguments sous forme de tableau dont la dernière valeur est NULL
- Des variantes:
  - *p ⇒ si le premier argument de la fonction ne contient pas de / l'exécutable est cherché dans la variable d'environnement PATH
  - *e ⇒ l'argument envp contient les variables d'environnement qui seront fournis au programme exécuté

# Un interpréteur de commande très simple et très limité

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

int main(void)
{
        char cmd[100] = {0};
        do {
                printf("interp: $ ");
                scanf("%99s",cmd);
                if (strcmp("exit",cmd) != 0 && fork() == 0)
                {
                        char* argv[2] = {cmd,NULL};
                        execvp(cmd,argv);
                        perror("exec");
                        exit(-1);
                }
                wait(NULL);
        } while(strcmp("exit",cmd));
        return 0;
}
```

# Fichiers

# Fichiers Standards

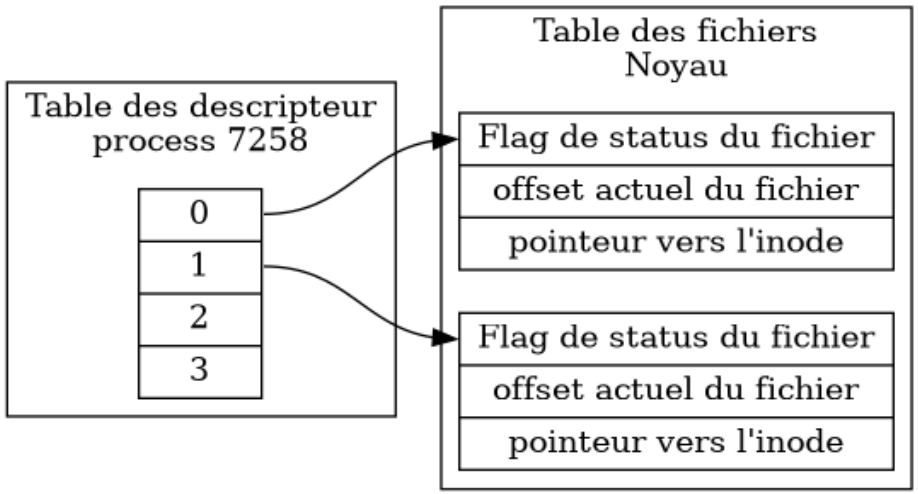
Tout processus a au moins trois fichiers ouverts lors de son lancement

| File Descriptor | FILE* | Description |
|---|---|---|
| 0 | stdin | Entrée Standard |
| 1 | stdout | Sortie Standard |
| 2 | stderr | Sortie d'Erreur |

Les appels système read et write utilise les descripteurs de fichier (0,1,2)
Les fonctions fread et fwrite utilise les FILE* (stdin, stdout, stderr)

# Table des descripteurs de fichier

- À chaque fichier ouvert correspond un descripteur de fichier
- Les descripteurs de fichier appartiennent aux processus
- La table des fichiers est géré par le noyau
- Pour chaque fichier le noyau gère un certain nombre d'information
  - status d'ouverture du fichier
  - position de lecture/écriture
  - pointeur vers l'inode
- Si le fichier est ouvert deux fois, il y a deux entrées



# Fork

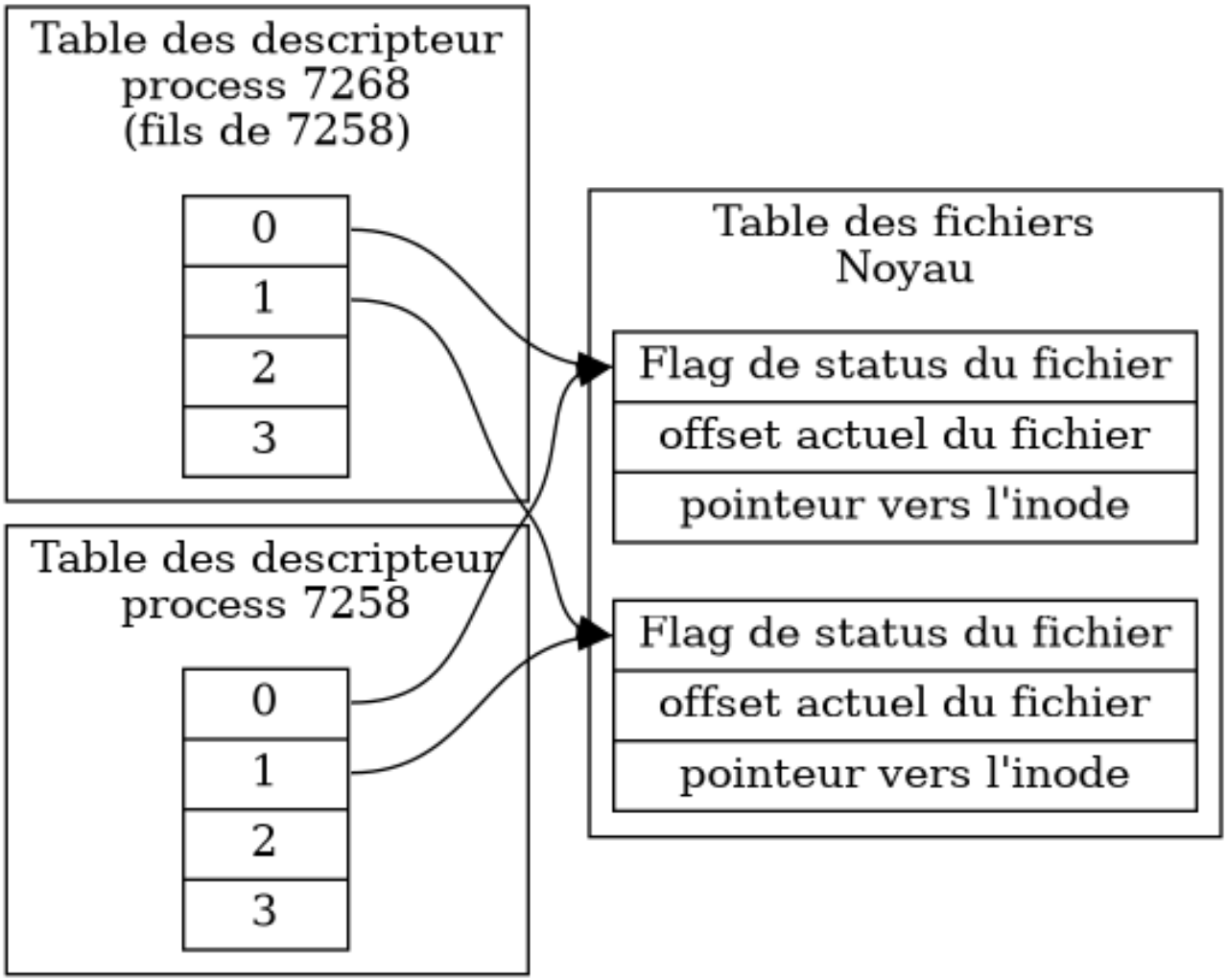


Table des descripteur
process 7268
(fils de 7258)
0
1
2
3
Table des descripteur
process 7258
0
1
2
3
Table des fichiers
Noyau
Flag de status du fichier
offset actuel du fichier
pointeur vers l'inode
Flag de status du fichier
offset actuel du fichier
pointeur vers l'inode

# open/close

```
int open(const char* pathname, int flags);
int open(const char* pathname, int flags, mode_t mode);
int close(int fd);
```

- open renvoie un descripteur de fichier
- open prends des flags pour savoir comment ouvrir le fichier (read/write/...)
- Avec certains flags, il est nécessaire de préciser les droits sur le fichier (mode)
- close permet de fermer un descripteur de fichier
- Quelques flags:
  - O_RDONLY
  - O_WRONLY
  - O_RDWR
  - O_CREAT
  - O_APPEND
  - O_TRUNC

# read/write

```
ssize_t read(int fd, void buf[.count], size_t count);
ssize_t write(int fd, const void buf[.count], size_t count);
```

- read lit au maximum count octet dans le fichier associé au descripteur de fichier
- buf doit avoir une taille de count octet
- buf est un tableau void (il peut donc s'agir de n'importe quel type de tableau)
- read/write renvoi la taille lue/écrite qui peut être plus petite que count

# Exemple

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

#define MAX_SIZE 512

int main()
{
    int input = open("fichier",O_RDONLY);
    int output = open("output",O_CREAT|O_WRONLY,0644);
    size_t size_read = 0;
    char data[MAX_SIZE]={0};
    do {
        size_read = read(input,data,MAX_SIZE);
        size_t size_write = write(output,data,size_read);
        if (size_read != size_write) {
            perror("read write error");
            exit(-1);
        }
    }
    while(size_read > 0);
    close(input);
    close(output);

    return 0;
}
```

# Exemple

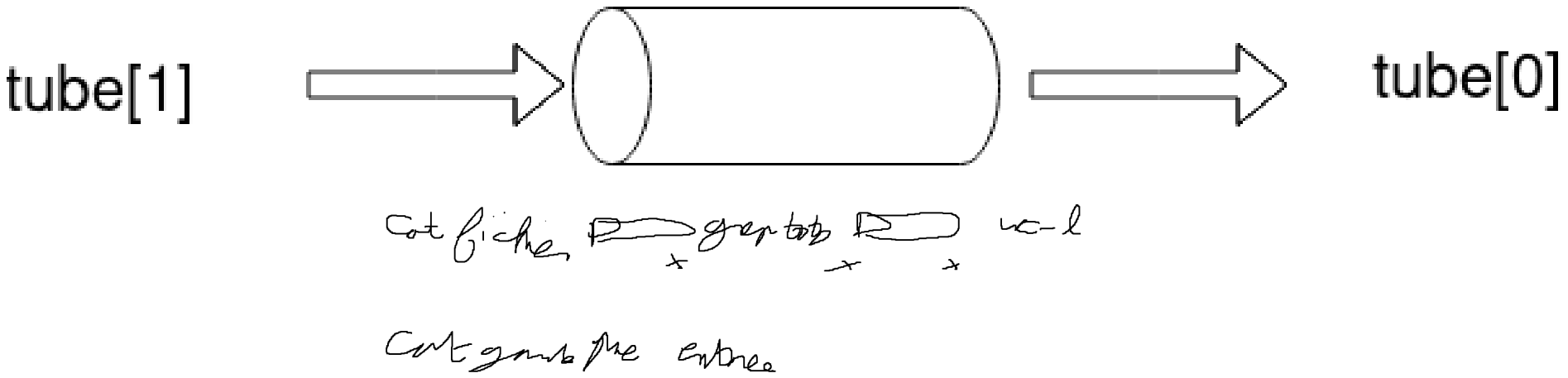
```c
#include <stdio.h>
#include <unistd.h>
#include <fcntl.h>
#include <stdlib.h>

int main()
{
    int input = open("fichier",O_RDONLY);
    int nb_float = 0;
    size_t size_read = read(input,&nb_floar,sizeof(int));
    float tab[nb_float] = {0};
    size_t size_read = read(input,tab,nb_float * sizeof(float));
    close(input);

    return 0;
}
```

# Tubes anonymes (Pipe)

# Pipe

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

- Crée un tube
- Un tube a une entrée tube[1] et une sortie tube[0]
- pipefd est un tableau qui contient les descripteurs de fichiers de la sortie et de l'entrée du tube
- Le tube fonctionne sur un système de FIFO (First In Firt Out)
- ⚠ Il faut absolument fermer les extrémités que l'on utilise pas!



# Exemple: Échange d'entier

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

int main()
{
    int tube[2];
    int res = pipe(tube);
    if (res == -1){
        perror("Création Pipe"); exit(-1);
    }

    if (fork() == 0) {
        int random = 0;
        close(tube[1]);
        puts("Waiting data"); //On est bloqué jusqu'à ce qu'il y ai quelque chose d'écrit dans le tube
        read(tube[0],&random,sizeof(int)); //Lecture depuis le tube
        close(tube[0]);
        printf("Receive: %d\n",random);
    } else  {
        sleep(8);
        srand(getpid());
        close(tube[0]);
        int random = rand()%50;
        printf("Send: %d\n",random);
        write(tube[1],&random,sizeof(int)); //Écriture dans le tube
        close(tube[1]);
    }
    return 0;
}
```

# Exemple; Protocole

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

int main()
{
    int tube[2];
    int res = pipe(tube);
    if (res == -1) {
        perror("Création Pipe"); exit(-1);
    }

    if (fork() == 0) {
        int size = 0;
        close(tube[1]);
        read(tube[0],&size,sizeof(int));
        char chaine[size];
        read(tube[0], chaine, size*sizeof(char));
        close(tube[0]);
        printf("Receive(%d): %s\n",size,chaine);
    } else {
        close(tube[0]);
        const char* chaine = "Hello World!";
        int size = strlen(chaine)+1;
        printf("Send(%d): %s\n",size,chaine);
        write(tube[1],&size,sizeof(int));
        write(tube[1],chaine,sizeof(char)*size);
        close(tube[1]);
    }
    return 0;
}
```

# Redirection

# Redirection

```
int dup2(int oldfd, int newfd);
```

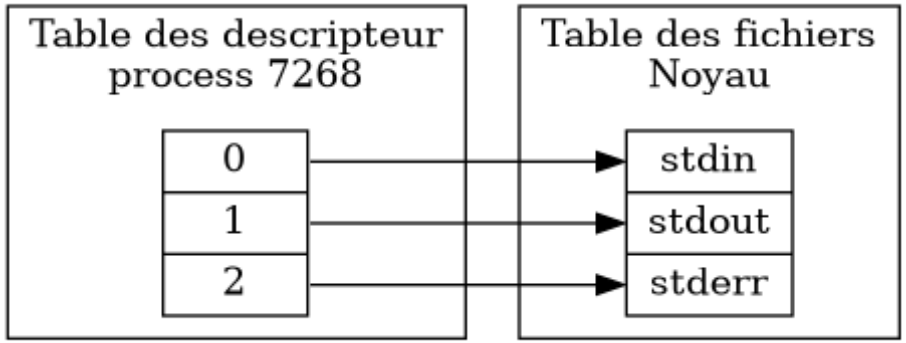
- oldfd est le descripteur de fichier que l'on veut dupliquer
- newfd est le descripteur de fichier sur lequel on veut faire la duplication
- oldfd et newfd pointent sur le même fichier.
- Utilisé autant pour les tubes (|) que pour les redirections entrée/sortie (>, »,«,<,2>...)

# Exemple

```
cat fichier | wc -l
```

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

int main()
{
    int tube[2];
    int res = pipe(tube);
    if (res == -1)  {
        perror("Création Pipe");
        exit(-1);
    }

    if (fork() == 0) {
        close(tube[1]);
        dup2(tube[0],0);
        close(tube[0]);
        execlp("wc","wc","-l",NULL);
    } else {
        close(tube[0]);
        dup2(tube[1],1);
        close(tube[1]);
        execlp("cat","cat","fichier",NULL);
    }
    return 0;
}
```

<=

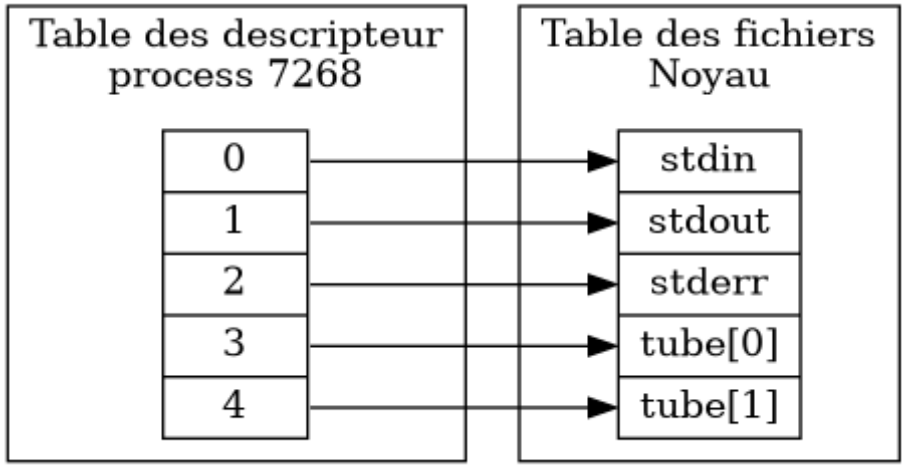


# Exemple

```
cat fichier | wc -l
```

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

int main()
{
    int tube[2];
    int res = pipe(tube);
    if (res == -1)  {
        perror("Création Pipe");
        exit(-1);
    }

    if (fork() == 0) {
        close(tube[1]);
        dup2(tube[0],0);
        close(tube[0]);
        execlp("wc","wc","-l",NULL);
    } else {
        close(tube[0]);
        dup2(tube[1],1);
        close(tube[1]);
        execlp("cat","cat","fichier",NULL);
    }
    return 0;
}
```

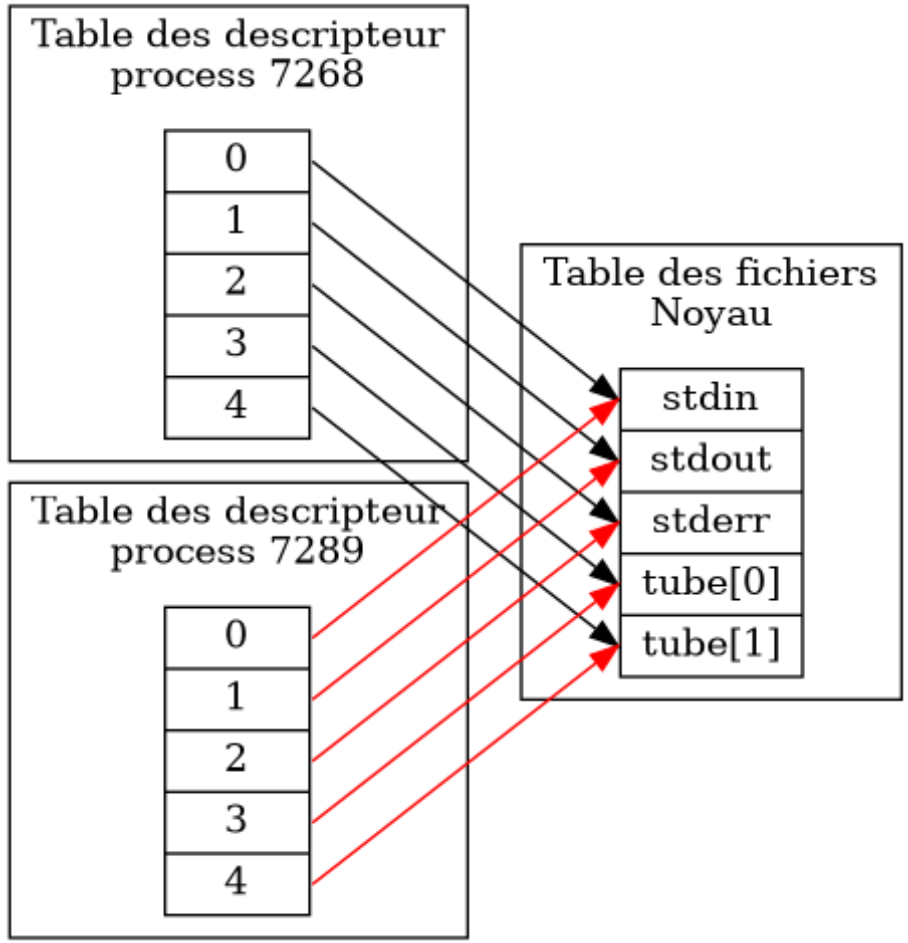
<=



# Exemple

```
cat fichier | wc -l
```

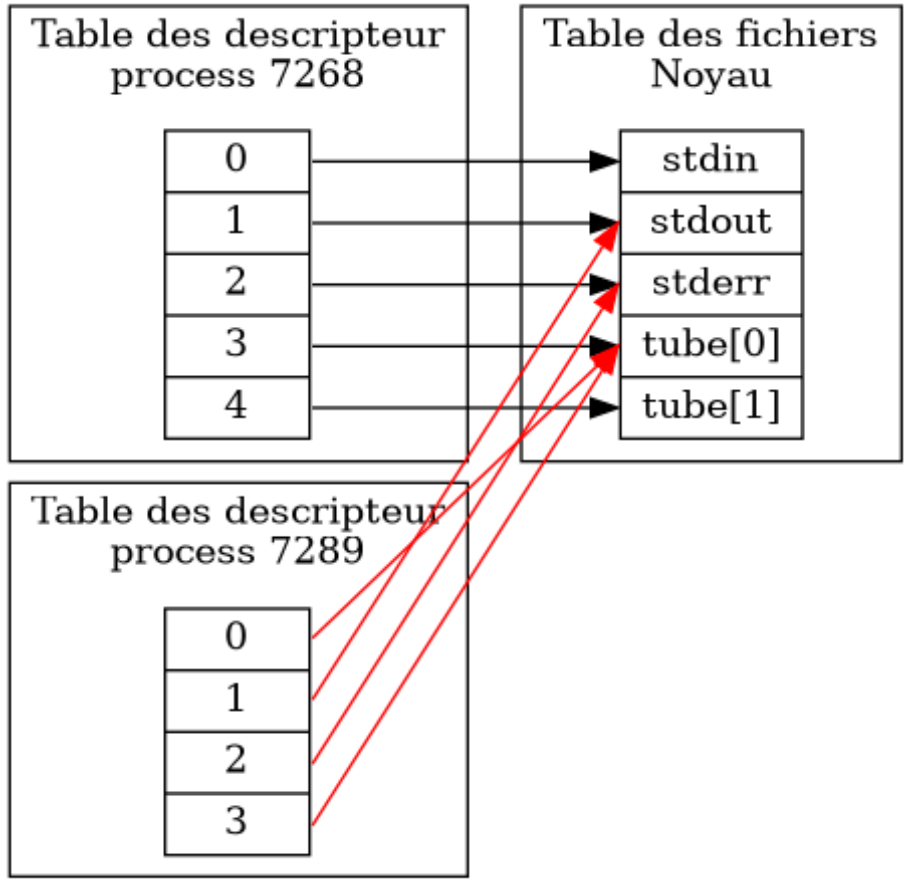
```c
#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>

int main()
{
    int tube[2];
    int res = pipe(tube);
    if (res == -1)  {
        perror("Création Pipe");
        exit(-1);
    }

    if (fork() == 0) {                                    <=
        close(tube[1]);
        dup2(tube[0],0);
        close(tube[0]);
        execlp("wc","wc","-l",NULL);
    } else {
        close(tube[0]);
        dup2(tube[1],1);
        close(tube[1]);
        execlp("cat","cat","fichier",NULL);
    }
    return 0;
}
```



# Exemple

```
cat fichier | wc -l
```

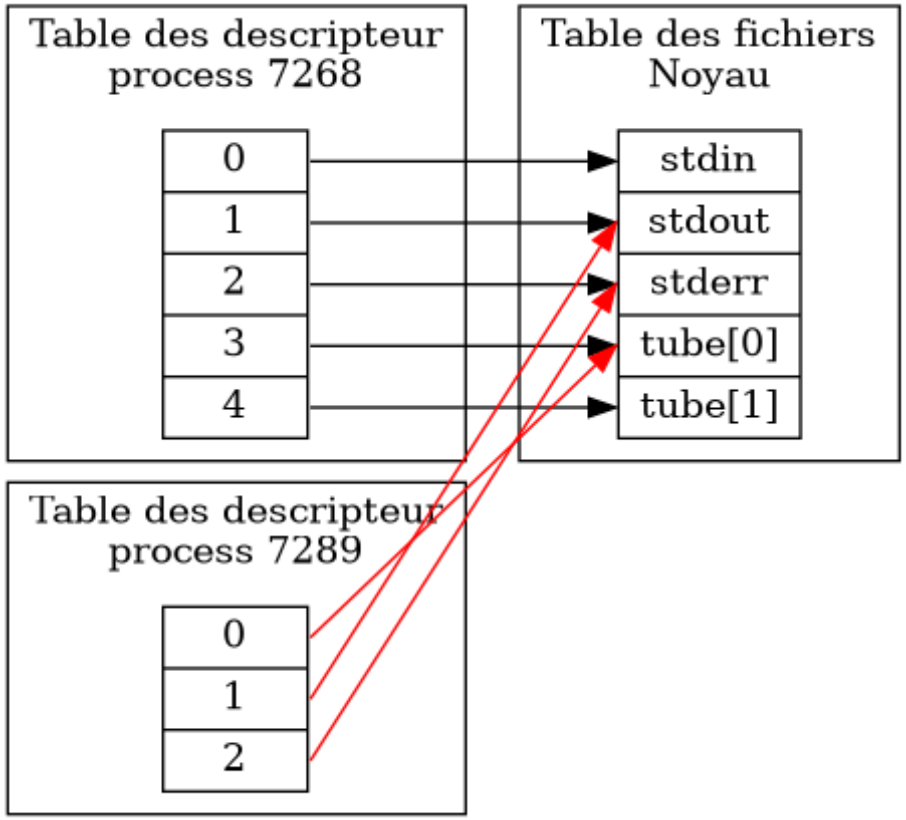
```c
#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>

int main()
{
    int tube[2];
    int res = pipe(tube);
    if (res == -1)  {
        perror("Création Pipe");
        exit(-1);
    }

    if (fork() == 0) {
        close(tube[1]);
        dup2(tube[0],0);                    <=
        close(tube[0]);
        execlp("wc","wc","-l",NULL);
    } else {
        close(tube[0]);
        dup2(tube[1],1);
        close(tube[1]);
        execlp("cat","cat","fichier",NULL);
    }
    return 0;
}
```



# Exemple

```
cat fichier | wc -l
```

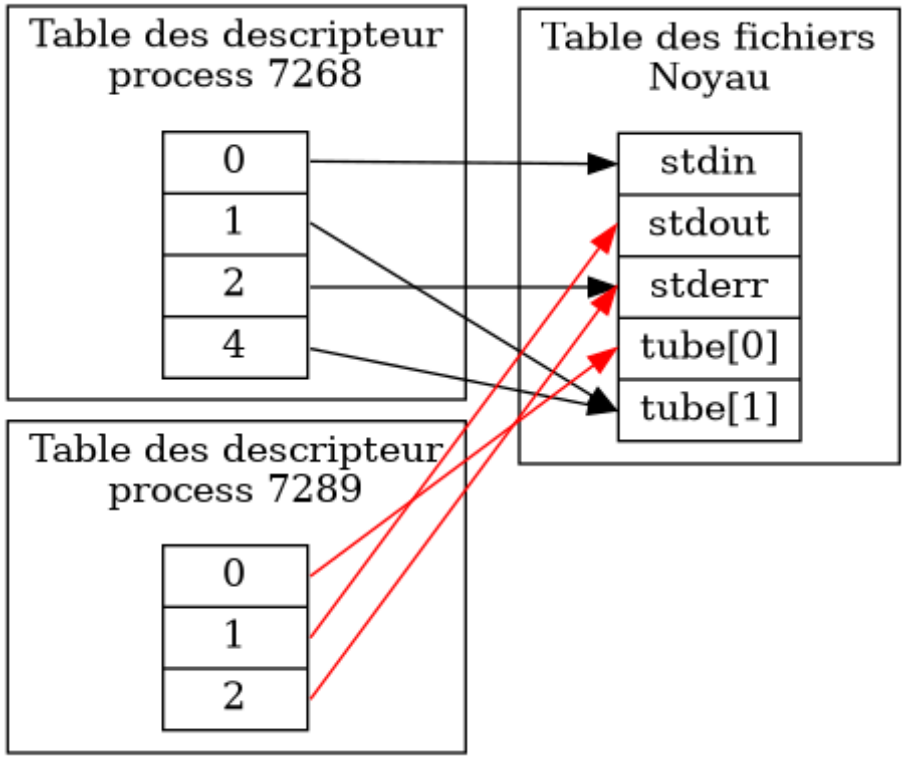
```c
#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>

int main()
{
    int tube[2];
    int res = pipe(tube);
    if (res == -1)  {
        perror("Création Pipe");
        exit(-1);
    }

    if (fork() == 0) {
        close(tube[1]);
        dup2(tube[0],0);
        close(tube[0]);                          <=
        execlp("wc","wc","-l",NULL);
    } else {
        close(tube[0]);
        dup2(tube[1],1);
        close(tube[1]);
        execlp("cat","cat","fichier",NULL);
    }
    return 0;
}
```



# Exemple

```
cat fichier | wc -l
```

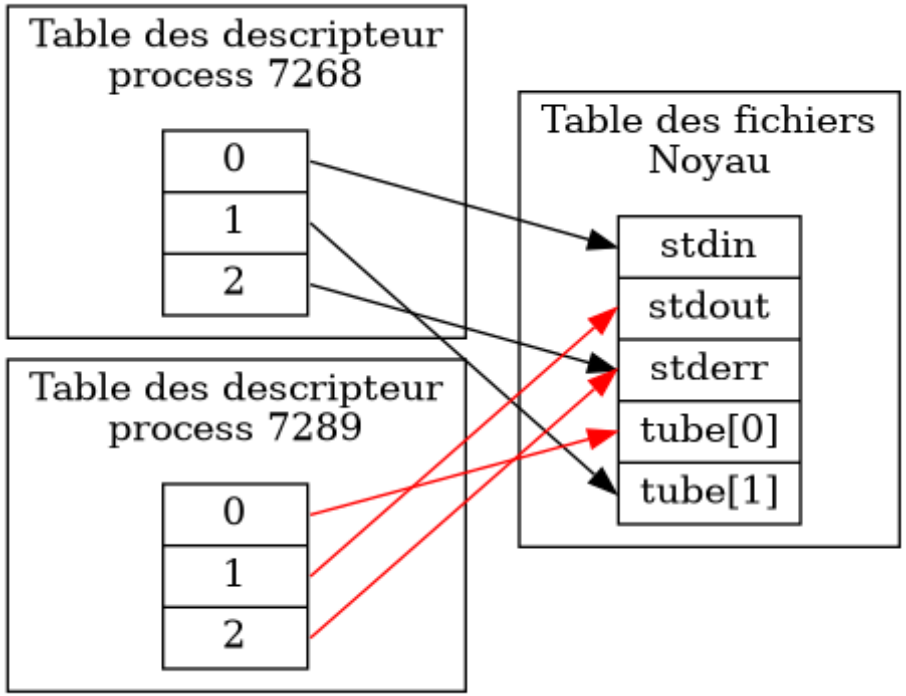
```c
#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>

int main()
{
    int tube[2];
    int res = pipe(tube);
    if (res == -1)  {
        perror("Création Pipe");
        exit(-1);
    }

    if (fork() == 0) {
        close(tube[1]);
        dup2(tube[0],0);
        close(tube[0]);
        execlp("wc","wc","-l",NULL);
    } else {
        close(tube[0]);
        dup2(tube[1],1);                    <=
        close(tube[1]);
        execlp("cat","cat","fichier",NULL);
    }
    return 0;
}
```



# Exemple

```
cat fichier | wc -l
```

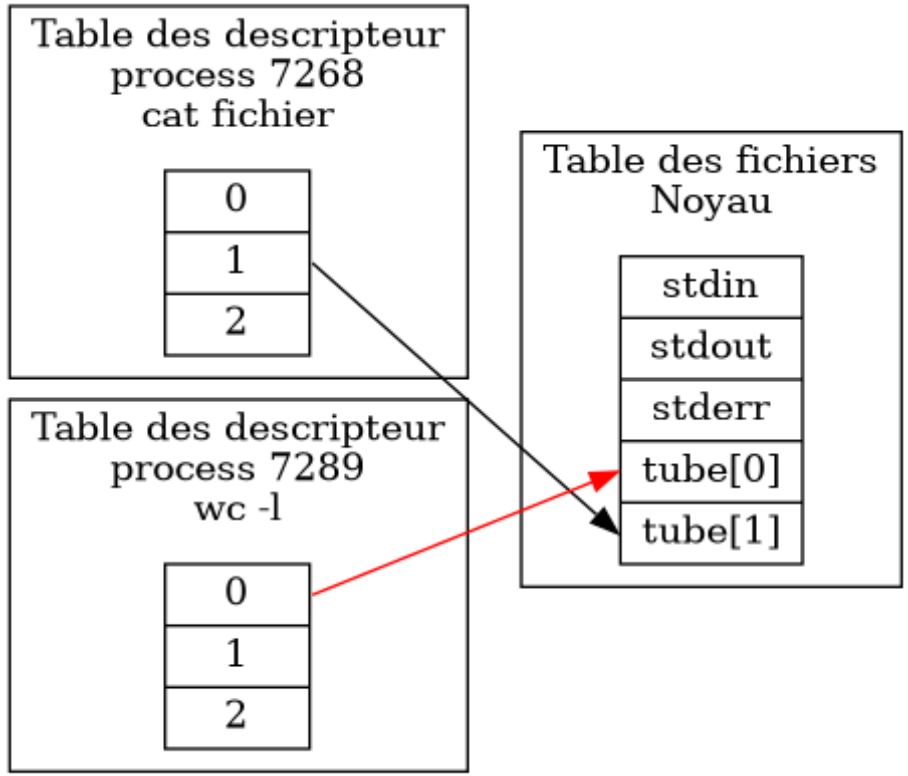
```c
#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>

int main()
{
    int tube[2];
    int res = pipe(tube);
    if (res == -1)  {
        perror("Création Pipe");
        exit(-1);
    }

    if (fork() == 0) {
        close(tube[1]);
        dup2(tube[0],0);
        close(tube[0]);
        execlp("wc","wc","-l",NULL);
    } else {
        close(tube[0]);
        dup2(tube[1],1);
        close(tube[1]);                          <=
        execlp("cat","cat","fichier",NULL);
    }
    return 0;
}
```



# Exemple

```
cat fichier | wc -l
```

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

int main()
{
    int tube[2];
    int res = pipe(tube);
    if (res == -1)  {
        perror("Création Pipe");
        exit(-1);
    }

    if (fork() == 0) {
        close(tube[1]);
        dup2(tube[0],0);
        close(tube[0]);
        execlp("wc","wc","-l",NULL);
    } else {
        close(tube[0]);
        dup2(tube[1],1);
        close(tube[1]);
        execlp("cat","cat","fichier",NULL);
    }
    return 0;
}
```



# Tubes Nommés

# mkfifo

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

- Crée un fichier de type tube sur le système de fichier
- mode correspond aux permissions d'accès du fichier
- Deux processus indépendants peuvent ainsi communiquer
- Les deux extrémités du tube doivent être ouvertes pour que les processus puissent communiquer
  - open ⇒ RDONLY ⇒ Sortie du tube
  - open ⇒ WRONLY ⇒ Entrée du tube
- read/write pour écrire dans le tube

```
int unlink(const char *pathname);
```

- Supprime un nom de fichier, si c'est le dernier, le fichier est supprimé

# Exemple Client

```c
#define SERVEUR "Serveur"
#define CLIENT "Client"

void error(char* file) {
    fprintf(stderr,"Can't open %s\nLaunch Server before client\n",file); exit(1);
}

int main(void)
{
    int dfr, dfw;
    dfw = open(SERVEUR,O_WRONLY);
    if(dfw == -1){ error(SERVEUR); }
    dfr = open(CLIENT,O_RDONLY);
    if(dfw == -1){  error(CLIENT); }
    srand(getpid());
    for(int i = 0; i < 10; i++) {
        int val = rand()%10;
        char op = '+';
        int val2 = rand()%10;
        write(dfw,&val,sizeof(int));
        write(dfw,&op,sizeof(char));
        write(dfw,&val2,sizeof(int));
        int result;
        read(dfr,&result,sizeof(int));
        printf("%d %c %d = %d\n",val,op,val2,result);
    }
    close(dfr);
    close(dfw);
    return 0;
}
```

# Exemple Serveur

```c
#define SERVEUR "Serveur"
#define CLIENT "Client"

void error(char* file) {
    fprintf(stderr,"Can't create %s\n",file); exit(1);
}

int main(void) {
    int dfr, dfw;
    unlink(SERVEUR);
    unlink(CLIENT);
    if(mkfifo(SERVEUR,0644) == -1){ error(SERVEUR); }
    if(mkfifo(CLIENT,0644) == -1){ error(CLIENT); }
    dfr = open(SERVEUR,O_RDONLY);
    dfw = open(CLIENT,O_WRONLY);
    int end = 1;
    while(end) {
        int val,val2;
        char op;
        end = read(dfr,&val,sizeof(int));
        if (end == 0)   continue;
        read(dfr,&op,sizeof(char));
        read(dfr,&val2,sizeof(int));
        int resultat = 0;
        if( op == '+')
            resultat = val + val2;
        write(dfw,&resultat,sizeof(int));
    }
    close(dfr); close(dfw);
    unlink(CLIENT); unlink(SERVEUR);
    return 0;
}
```

# Principe

# Origine

- Introduits par Edsger Dijkstra en 1965
  - Mathématicien et Informaticien néerlandais. Lauréat du Prix Turing en 1972 pour ses contributions à la science et l'art des langages de programmation et au langage Algol.
  - Connu pour l'algorithme de Dijkstra (Chemin le plus court) et pour les Semaphores

# Concept

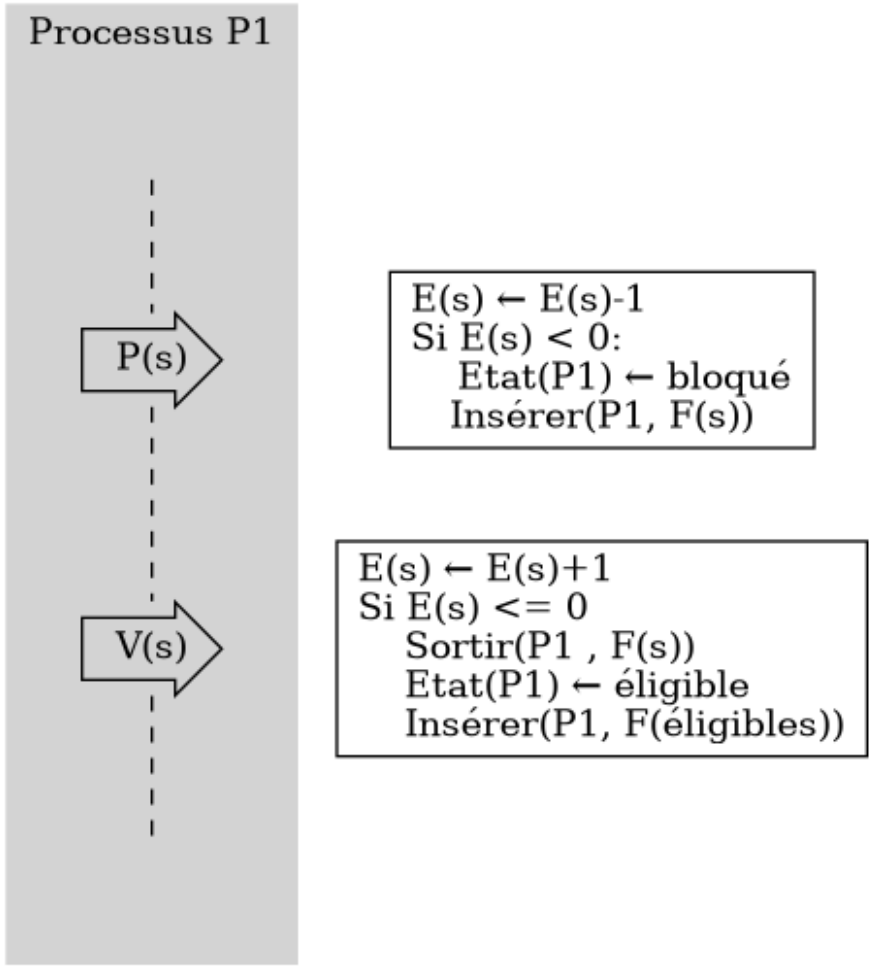
- Outil élémentaire de synchronisation multi-processus/multi-thread évitant l'attente active

# Concrètement

Un Sémaphore c'est:

- Un entier E(s)
- Une file d'attente F(s)
- Deux Opérations:
  - P(s)
  - V(s)
- P et V sont des opérations atomiques

# P et V


Processus P1
P(s)
V(s)
E(s) ← E(s)-1
Si E(s) < 0:
Etat(P1) ← bloqué
Insérer(P1, F(s))
E(s) ← E(s)+1
Si E(s) <= 0
Sortir(P1 , F(s))
Etat(P1) ← éligible
Insérer(P1, F(éligibles))

# Cas particulier: Mutex

# Objectifs

- Protéger une ressource critique
  - ⇒ Un seul processus a le droit d'accéder à la ressource pendant autant de temps que nécessaire
- La protéger le moins longtemps possible pour éviter de bloquer longtemps d'autres processus
  - ⇒ Éviter de bloquer tous les processus alors que le processus ayant fait le verrouillage n'a plus besoin de la ressource

# Comment faire?

## Un seul processus a le droit d'accéder à la ressource

- Un sémaphore possédant un jeton lors de son initialisation
- Pour chaque opération P, il y a une opération V faites par le même processus
- Un V est forcément précédé d'un P

## Éviter de bloquer tous les processus trop longtemps

- Copier la ressource localement pour travailler avec dans son coin?
- Structurer le code pour avoir l'usage de cette donnée de manière la plus réduite possible
  - ⇒ Est-ce que je bloque la ressource avant la boucle ? Lors de chaque itération de la boucle ?

# Exemple d'utilisation

# Exemple d'utilisation

- Gestion d'un parking avec barrière d'accès
- Rendez-vous (à partir de 2 et jusqu'à N personne)
- Gestion d'accès à une Base de Données (Modèle Lecteur/Rédacteur)
- Gestion d'accès à des ressources (Exemple des philosophes)
- Producteur/Consommateur
- ...

# Cas du parking avec barrière d'accès

- Un nombre de place limité.
- Une barrière à l'entrée
  - ⇒ s'il n'y a plus de place, on attend
- Une barrière à la sortie
  - ⇒ permet à la prochaine personne de rentrer

# Gestion d'accès à une Base de Données (Modèle Lecteur/Rédacteur)

Lecteur

```
//...
DebutLecture();
Lire();
FinLecture();
```

Redacteur

```
//...
DebutEcriture();
Lire();
FinEcriture();
```

# Base de donnée à Priorité Rédacteur

- R: Semaphore de protection de la BDD (Mutex)
- Lec: Semaphore d'exclusion entre Lecteur (1 Jeton)
- Compt: Semaphore de Protection des compteurs partagé (Mutex)
- PR: Sémaphore de gestion de la priorité Rédacteur (1 Jeton)

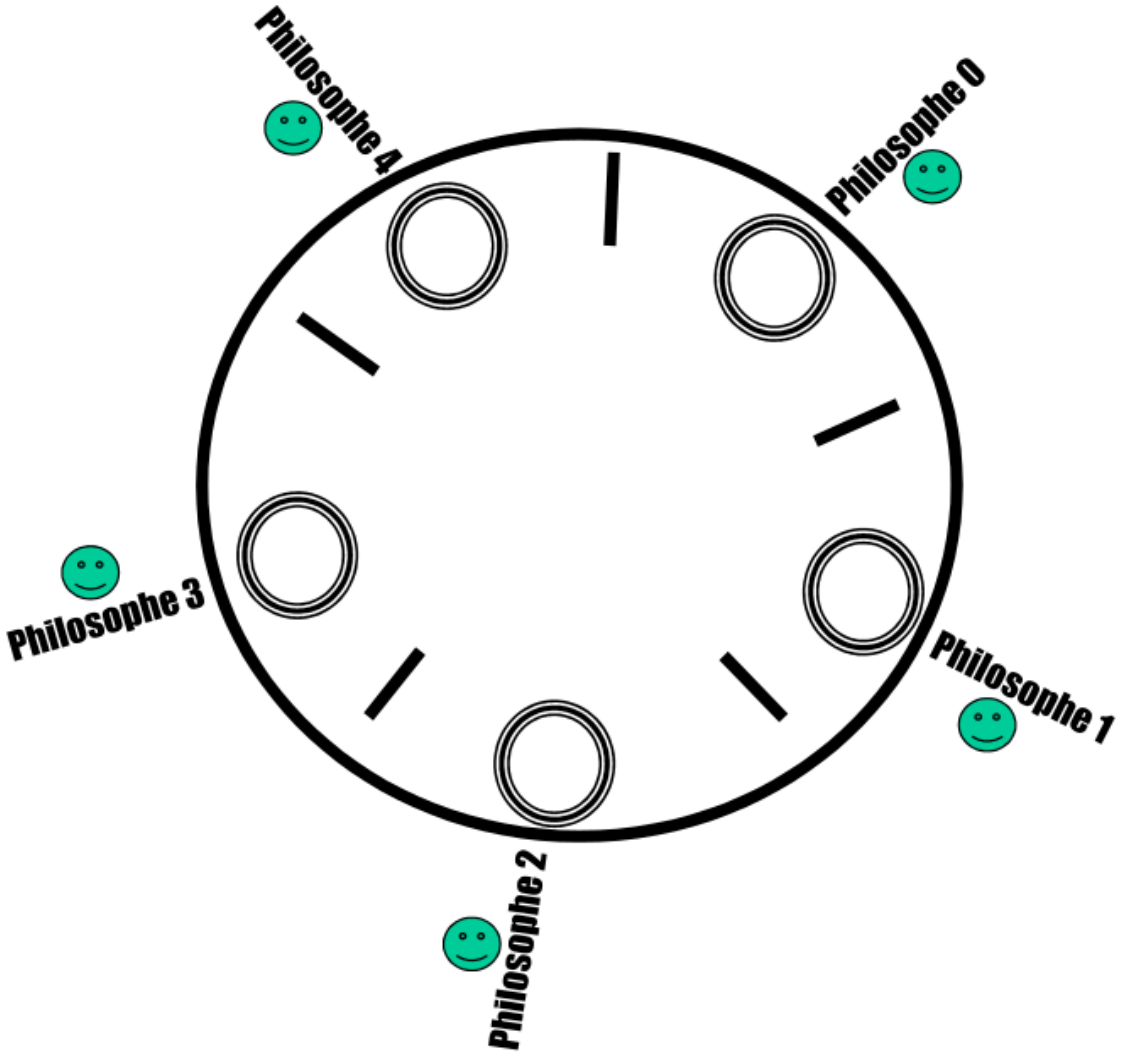
```
void DebutLecture() {
    P(Lec);
    P(PR);
    P(Compt);
    NbLecteur++;
    if (NBLecteurs == 1)
        P(R);
    V(Compt);
    V(PR);
    V(Lec);
}
```

```
void FinLecture() {
    P(Compt);
    NbLecteur--;
    if (NbLecteur == 0)
        V(R);
    V(Compt);
}
```

```
void DebutEcriture() {
    P(Compt);
    NbRedacteur++;
    if (NbRedacteur == 1)
        P(PR);
    V(Compt);
    P(R);
}
```

```
void FinEcriture() {
    V(R);
    P(Compt);
    NbRedacteur--;
    if (NbRedacteur == 0)
        V(PR);
    V(Compt);

}
```

# Gestion d'accès à des ressources (Exemple des philosophes)



# Gestion d'accès à des ressources (Exemple des philosophes)

Sémaphore Fourchette[5] initialisés à 1

```
void Philosophe(int i)
{
    while(1)
    {
        Penser()
        P(Fourchette(i])
        P(Fourchette( (i+1) % 5])
        Manger()
        V(Fourchette( (i+1) % 5])
        V(Fourchette(i])
    }
}
```

# En C

# Création/Obtention du semaphore

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

- key: est une clef unique permettant d'identifier un semaphore
- nsems: doit toujours valoir 1
- semflg: droit sur le sémaphore
  - 644 par exemple
  - Peut être couplé avec IPC_CREAT et IPC_EXCL

# Initialisation et Destruction

```
union semun {
    int              val;    /* Valeur pour SETVAL */
    struct semid_ds *buf;    /* Tampon pour IPC_STAT, IPC_SET */
    unsigned short  *array;  /* Tableau pour GETALL, SETALL */
    struct seminfo  *__buf;  /* Tampon pour IPC_INFO
                                (spécifique à Linux) */
};

int semctl(int semid, int semnum, int op, arg);
```

- semid: identifiant du semaphore obtenu via semget
- op ⇒ SETVAL (interdiction d'utiliser GETVAL) ou IPC_RMID
- semnum: Toujours 0
- arg: Élément de l'union semun ou 0 dans le cas de IPC_RMID
  - ⇒ pour SETVAL arg.val sera utilisé avec la valeur initiale du semaphore

# Opération sur le semaphore

```
struc sembuf {
    unsigned short sem_num;   /* Numéro du sémaphore          */
    short          sem_op;    /* Opération sur le sémaphore */
    short          sem_flg;   /* Options pour l'opération    */
}

int semop(int semid, struct sembuf *sops, size_t nsops);
```

- semid: identifiant du semaphore obtenu via semget
- sops: Pointeur vers une structure sembuf
  - semnum ⇒ 0
  - semop ⇒ 1(V) ou -1(P)
  - sem_flg ⇒ 0
- nsops ⇒ 1

# Implémentation de Dijkstra

# Initialisation

```
int sem_create(key_t cle, int initval) {
    int semid;
    union semun { int val; struct semid_ds *buf; ushort *array; } arg_ctl;

    semid = semget(cle, 1 , IPC_CREAT|IPC_EXCL|0600);
    if (semid == -1)
    {
        return -1;
    }
    else
    {
        arg_ctl.val = initval;
        if (semctl(semid, 0, SETVAL, arg_ctl) == -1)
        {
            perror("Erreur initialisation sémaphore");
            exit(1);
        }
    }
    return(semid);
}
```

# Récupération du semaphore

```c
int sem_get(key_t cle)
{
    int semid = semget(cle,1, 0600);
    if (semid == -1)
    {
        perror("Erreur semget()");
        exit(1);
    }
    return semid;
}
```

# P et V

```c
void P(int semid)
{
    struct sembuf sempar;
    sempar.sem_num = 0;
    sempar.sem_op = -1;
    sempar.sem_flg = 0;
    if (semop(semid, &sempar, 1) == -1)
        perror("Erreur operation P");
}

void V(int semid)
{
    struct sembuf sempar;
    sempar.sem_num = 0;
    sempar.sem_op = 1;
    sempar.sem_flg = 0;
    if (semop(semid, &sempar, 1) == -1)
        perror("Erreur opération V");
}
```

# Suppression du semaphore

```c
void sem_delete(int semid)
{
    if (semctl(semid,0,IPC_RMID,0) == -1)
        perror("Erreur dans destruction sémaphore");
}
```

# Mémoire Partagée

# Principe

- Permettre le partage de mémoire entre plusieurs processus.
- Moyen simple d'échange de donnée
- ⚠ Espace Critique ⇒ Utilisation de sémaphore (Mutex)

# Création/Récupération

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

- Une clé permettant d'identifier le segment
- La taille souhaitée
- Les droits sur le segment de mémoire partagée
- Un identifiant pour les opérations

# Attachement/Détachement

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

- shmat
  - identifiant
  - NULL
  - 0
  - ⇒ Pointeur vers la mémoire partagée
- shmdt
  - pointeur vers la mémoire à détacher

# Suppression

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

- Identifiant
- IPC_RMID
- NULL

# Exemple

```c
//init.c
#define KEY 5942
#define KEY_2 5943
#define SIZE 4

int main(void)
{
    sem_create(KEY,1);
    sem_create(KEY_2,0);
    int shmid = shmget(KEY,SIZE * sizeof(int),
                IPC_CREAT|IPC_EXCL|0600);
    if (shmid == -1) {
        perror("Shmget Memory");
    }

    return 0;
}
```

```c
//clean.c
#define KEY 5942
#define KEY_2 5943
#define SIZE 4

int main(void)
{
    int shmid = shmget(KEY,SIZE * sizeof(int),0);
    int res = shmctl(shmid,IPC_RMID,NULL);
    if (res == -1) {
        perror("Delete Shm");
    }
    int mutexId = sem_create(KEY,1);
    sem_delete(mutexId);
    int semId = sem_create(KEY_2,0);
    sem_delete(semId);


    return 0;
}
```

# Exemple

```c
//write_memory.c
#define KEY 5942
#define KEY_2 5943
#define SIZE 4

int main()
{
    srand(getpid());
    int shmid = shmget(KEY,SIZE * sizeof(int),0);
    int mutexId = sem_create(KEY,1);
    int semId = sem_create(KEY_2,0);

    int* mem = shmat(shmid,NULL,0);

    P(mutexId);
    for (int i = 0; i < SIZE; i++) {
        mem[i] = rand()%5;
        printf("gen: %d\n",mem[i]);
        sleep(1);
    }
    V(mutexId);
    V(semId);
    shmdt(mem);

    return 0;
}
```

```c
//read_memory.c
#define KEY 5942
#define KEY_2 5943
#define SIZE 4

int main()
{
    int shmid = shmget(KEY,SIZE * sizeof(int),0);
    if (shmid == -1) {
        perror("Shmget Memory");
        exit(-1);
    }
    int* mem = shmat(shmid,NULL,0);
    if ((int64_t)mem == -1) {
        perror("Shmat Memory");
        exit(-1);
    }

    int mutexId = sem_create(KEY,1);
    int semId = sem_create(KEY_2,0);
    P(semId);
    P(mutexId);
    for (int i = 0; i < SIZE; i++) {
        int var = mem[i];
        printf("read: %d\n",var);
    }
    V(mutexId);
    shmdt(mem);

    return 0;
}
```

# Buffer Circulaire

# Objectif

- Échanger beaucoup de données
- Utiliser un minimum de mémoire

# Problème

- Mono-Process ⇒ facile
- Multi-Process
  - Le process qui lit a-t-il des données à lire?
  - le process qui écrit ne va-t-il pas écraser les données avant qu'elles ne soient lues?
  - Comment gérer les positions?
  - Comment faire en sorte que les processus ne se marchent pas dessus

## Comment on règle les problèmes

# Thread

# Principe

- Un processus est décomposé en thread (processus légers)
- Un thread correspond à une tâche qui s'exécute plus ou moins indépendants des autres
- Thread ⇒ Fil d'exécution
  - Une application a donc de fait toujours au moins un thread
- Mémoire automatiquement partagé
- Changement de contexte plus rapide

# Création

equivalent à un fork

```
int pthread_create(pthread_t *restrict thread,
                   const pthread_attr_t *restrict attr,
                   void *(*start_routine)(void *),
                   void *restrict arg);
```

- un pointeur vers une variable pthread_t qui sera modifié par la fonction permettant les opérations sur le thread
- NULL
- Un pointeur vers une fonction dont le prototype est:
  - `void* fonction(void* arg)`
- Un pointeur vers les arguments que recevra la fonction quand elle sera exécutée
- retourne 0 en cas de succès, tout autre valeur signale une erreur

# Terminaison

```c
int pthread_join(pthread_t thread, void **retval);
```

- Attends le thread identifié par la variable thread
- Récupère le résultat du thread
  - valeur issue du return du thread
  - valeur issue de l'appel à `pthread_exit(void *retval)`
- Comme wait pour les processus, une partie des informations du thread est conservé jusqu'à l'utilisation de pthread_join

equivalent à un wait

# Exemple 1

Compilation et Édition de lien à réaliser avec l'option -pthread

```c
#include <stdio.h>
#include <pthread.h>

void* hello(void* arg)
{
    const char* name = arg;
    printf("Hello %s\n",name);

    return NULL;
}

int main(void)
{
    pthread_t thread;
    char* arg = "World!";
    pthread_create(&thread,NULL,hello,arg);

    return 0;
}
```

# Exemple 2

```c
struct threadArg {
    char* name; int compteur;
};

struct threadRet {
    int retour;
};

void* hello(void* arg) {
    const struct threadArg* arguments = arg;
    for(int i =0; i < arguments->compteur; i++) {
        printf("Hello %s\n",arguments->name); sleep(1);
    }
    struct threadRet* status = malloc(sizeof(struct threadRet));
    status->retour = arguments->compteur;
    return status;
}

int main(void) {
    pthread_t thread;
    struct threadArg arg;
    arg.name = "World!";
    arg.compteur = 3;
    pthread_create(&thread,NULL,hello,&arg);

    struct threadRet* status;
    pthread_join(thread,(void**)&status);
    printf("retour: %d\n",status->retour);
    free(status);

    return 0;
}
```

# Semaphore

# Création

```
#include <semaphore.h>

// Déclaration du sémaphore
sem_t semaphore;

// Initialisation du sémaphore (valeur initiale 1)
sem_init(&semaphore, 0, 1);
```

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

- un pointeur vers une variable sem_t qui sera modifié par la fonction permettant les opérations sur le semaphore
- 0
- Valeur initial du sémaphore

```
sem_wait(&semaphore);
```

# Opération

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

- sem_wait ⇒ Opération P
- sem_post ⇒ Opération V

# Destruction

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

Supprime le semaphore

# Exemple

```c
void *task1(void* arg) {
    sem_t* sem = arg;
    puts("start task1");
    sleep(5);
    puts("wake task2");
    sem_post(sem);
    return NULL;
}

void *task2(void* arg) {
    sem_t* sem = arg;
    puts("start task2");
    sem_wait(sem);
    puts("restart task2");
    return NULL;
}

int main(void)
{
    sem_t sem;
    sem_init(&sem,0,0);
    pthread_t thread, pthread_t thread2;
    pthread_create(&thread,NULL,task1,&sem);
    pthread_create(&thread2,NULL,task2,&sem);
    pthread_join(thread,NULL);
    pthread_join(thread2,NULL);
    sem_destroy(&sem);
    return 0;
}
```

# Mutex

# Création

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

- un pointeur vers une variable pthread_mutex_t qui sera modifié par la fonction permettant les opérations sur le mutex
- NULL

# Opération

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

- lock ⇒ Essaye de prendre le mutex sinon est bloqué (P)
- unlock ⇒ Libère le mutex (V)

# Destruction

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

Supprime le mutex

# Exemple

```c
#define N 4

struct structArg {
    pthread_mutex_t* mutex;
    int count;
};

void *task1(void* arg)
{
    struct structArg* args = arg;
    int counts = 0;
    for (int i =0; i < 1000000; i++)
        counts += 1;
    pthread_mutex_lock(args->mutex);
    args->count += counts;
    pthread_mutex_unlock(args->mutex);
    return NULL;
}

int main(void)
{
    pthread_mutex_t mutex;
    pthread_mutex_init(&mutex,NULL);
    pthread_t thread[N];
    struct structArg args;
    args.mutex = &mutex;
    args.count=0;
    for (int i =0; i < N; i++)
        pthread_create(&thread[i],NULL,task1,&args);

    for (int i =0; i < N; i++)
        pthread_join(thread[i],NULL);
    pthread_mutex_destroy(&mutex);

    printf("Count %d\n",args.count);
```

# Barrière

# Création

```
int pthread_barrier_init(pthread_barrier_t *restrict barrier,
      const pthread_barrierattr_t *restrict attr,
      unsigned count);
```

- Un pointeur vers une variable pthread_barrier_t qui sera modifié par la fonction permettant les opérations sur la barrière
- NULL
- Nombre de thread à attendre à la barrière

# Opération

```c
int pthread_barrier_wait(pthread_barrier_t *barrier);
```

- Bloque les threads jusqu'à ce qu'il y ait count thread

# Destruction

```c
int pthread_barrier_destroy(pthread_barrier_t *barrier);
```

Supprime la barrière

# Exemple

```c
#define N 4

struct structArg {
    pthread_barrier_t* barrier; int id;
};

void *task1(void* arg) {
    struct structArg* args = arg;   srand(getpid()+args->id);
    int random = (rand()%3)*args->id;
    printf("Thread %d sleeping %d\n",args->id, random); sleep(random);
    printf("Thread %d wait others\n", args->id);
    pthread_barrier_wait(args->barrier);
    printf("Thread %d awake\n", args->id);
    return NULL;
}

int main(void)
{
    pthread_barrier_t barrier;
    pthread_barrier_init(&barrier,NULL, 4);
    pthread_t thread[N];
    struct structArg args[N];
    for (int i =0; i < N; i++) {
        args[i].barrier = &barrier; args[i].id = i+1;
        pthread_create(&thread[i],NULL,task1,&args[i]);
    }

    for (int i =0; i < N; i++) {
        pthread_join(thread[i],NULL);
    }
    pthread_barrier_destroy(&barrier);

    return 0;
}
```

# 1 Signal

# Signal

# Principe

- Mécanisme fondamental de communication inter-processus (IPC)
- L'OS communique avec les processus dans différents cas:
  - Erreur (SIGSEGV, SIGPIPE...)
  - A la suite d'action utilisateur (SIGINT, SIGSTP,SIGCONT...)

# Les signaux 1/2

| Signal | Action | Commentaire |
|---|---|---|
| SIGABRT | Core | Signal d'arrêt d'abort(3) |
| SIGALRM | Term | Signal de temporisation d'alarm(2) |
| SIGBUS | Core | Erreur de bus (mauvais accès mémoire) |
| SIGCHLD | Ign | Enfant arrêté ou terminé |
| SIGCLD | Ign | Synonyme pour SIGCHLD |
| SIGCONT | Cont | Continuer si arrêté |
| SIGEMT | Term | Interception (trap) d'émulateur |
| SIGFPE | Core | Exception de virgule flottante |
| SIGHUP | Term | Déconnexion détectée sur le terminal de contrôle ou mort du processus de contrôle |
| SIGILL | Core | Instruction illégale |
| SIGINFO | | Synonyme pour SIGPWR |
| SIGINT | Term | Interruption depuis le clavier |
| SIGIO | Term | E/S maintenant possible (4.2BSD) |
| SIGIOT | Core | Interception IOT – synonyme pour SIGABRT |
| SIGKILL | Term | Signal d'arrêt |
| SIGLOST | Term | Perte de verrou de fichier (inutilisé) |
| SIGPIPE | Term | Tube brisé : écriture dans un tube sans lecteur – voir pipe(7) |
| SIGPOLL | Term | Événement scrutable (System V) – synonyme pour SIGIO |
| SIGPROF | Term | Fin d'une temporisation de profilage |
| SIGPWR | Term | Panne d'alimentation (System V) |
| SIGQUIT | Core | Quitter depuis le clavier |
| SIGSEGV | Core | Référence mémoire non valable |
| SIGSTKFLT | Term | Erreur de pile sur coprocesseur (inutilisé) |
| SIGSTOP | Stop | Processus d'arrêt |
| SIGTSTP | Stop | Stop saisi sur le terminal |
| SIGSYS | Core | Mauvais appel système (SVr4) – voir aussi seccomp(2) |

# Les signaux 2/2

| Signal | Action | Commentaire |
|---|---|---|
| SIGTERM | Term | Signal de fin |
| SIGTRAP | Core | Interception pour trace ou pour point d'arrêt |
| SIGTTIN | Stop | Entrée du terminal pour processus en arrière-plan |
| SIGTTOU | Stop | Sortie du terminal pour processus en arrière-plan |
| SIGUNUSED | Core | Synonyme pour SIGSYS |
| SIGURG | Ign | Condition urgente sur un socket (4.2BSD) |
| SIGUSR1 | Term | Signal utilisateur 1 |
| SIGUSR2 | Term | Signal utilisateur 2 |
| SIGVTALRM | Term | Horloge virtuelle d'alarme (4.2BSD) |
| SIGXCPU | Core | Limite de tempsCPU dépassée (4.2BSD) Consultez setrlimit(2) |
| SIGXFSZ | Core | Taille de fichier excessive (4.2BSD) Consultez setrlimit(2) |
| SIGWINCH | Ign | Fenêtre redimensionnée (4.3BSD, Sun) |

# Kill

```c
int kill(pid_t pid, int sig);
```

- Emet le signal de numéro sig à destination du processus pid
- pid > 0 ⇒ signal envoyé aux processus correspondant
- pid == 0 ⇒ signal envoyé à tous les processus appartenant au même groupe que l'appellant
- 0 ⇒ signal envoyé
- -1 ⇒ erreur

# Exemple

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

int main(void)
{
    pid_t son = fork();
    if(son == 0) {
        puts("start");
        sleep(20);
        puts("after sleep");
    }
    else {
        sleep(5);
        kill(son,SIGTERM);
    }
    puts("end");
    return 0;
}
```

# Interception de signaux

```
struct sigaction {
    void     (*sa_handler)(int);
    void     (*sa_sigaction)(int, siginfo_t *, void *);
    sigset_t   sa_mask;
    int        sa_flags;
    void     (*sa_restorer)(void);
};

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

- Numéro du signal a rediriger
- structure utilisé pour gérer la redirection ou SIG_IGN ou SIG_DFL
- structure pour récupérer l'ancienne redirection (utile pour restaurer l'état plus tard)
- sa_handler est une fonction void fonction(int signum)
- sa_flags permet de gérer le comportement de la redirection

# Exemple: SIGINT

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

int main(void)
{
    struct sigaction act;
    memset(&act,0,sizeof(act));
    act.sa_handler = SIG_IGN;
    sigaction(SIGINT,&act,NULL);
    sleep(15);
    puts("end");
    return 0;
}
```

# Exemple: SIGPIPE

```c
#include <signal.h>

void redirect(int signum) {
    if (signum == SIGPIPE)
        printf("SIGPIPE(%d) signal received\n",signum);
}

int main(void)
{
    struct sigaction act;
    memset(&act,0,sizeof(act));
    act.sa_handler = redirect;
    sigaction(SIGPIPE,&act,NULL);
    int tube[2];
    pipe(tube);
    int value = 5;
    close(tube[0]);
    write(tube[1],&value,sizeof(value));
    value = 0;
    read(tube[0],&value,sizeof(value));
    close(tube[1]);
    printf("value = %d\n",value);
    return 0;
}
```

# Alarm

```c
unsigned int alarm(unsigned int nb_sec);
```

- Envoi un signal SIGALRM au processus au bout de nb_sec secondes
- Si nb_sec == 0 ⇒ annule l'alarme en cours
- Renvoi 0 si aucune alarme n'avait été placé avant
- Renvoi le temps qu'il restait avant le déclenchement de l'alarme si une avait été précédemment placé

# Exemple

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

void redirect(int signum)
{
    if (signum == SIGALRM)
        printf("SIGALRM(%d) signal received\n",signum);
}

int main(void)
{
    struct sigaction act;
    memset(&act,0,sizeof(act));
    act.sa_handler = redirect;
    sigaction(SIGALRM,&act,NULL);
    alarm(5);
    sleep(10);
    puts("end");
    return 0;
}
```

# Python

# Base

- argc
- argv

```
import sys

if __name__ == "__main__":
    argc = len(sys.argv)
    print("Programme: " + sys.argv[0])
    for i in range(1,argc):
        print(sys.argv[i])
    sys.exit(0)
```

# Equivalence Process

- fork
- wait
- exec

```
import sys
import os

if __name__ == "__main__":
    pid = os.fork()
    if pid == 0:
        os.execlp("ls","ls","-l")
    (pid,status) = os.wait()
```

# Equivalence Pipe

- pipe
- dup2
- read
- write

```
import sys
import os

if __name__ == "__main__":
    tube = os.pipe()
    pid = os.fork()
    if pid == 0:
        os.close(tube[0])
        os.dup2(tube[1],1)
        os.close(tube[1])
        os.execlp("ls","ls","-l")
    size = 1
    os.close(tube[1])
    while size != 0:
        data = os.read(tube[0],1)
        size = len(data)
        if size > 0:
            print(data.decode(),end="")
```

# Multiprocessing

# Process

```
import multiprocessing as mp

def fonction(id, iter, tab):
    somme = 0
    for i in range(iter):
        somme += tab[i]
    print("Id", id, "Somme",somme)

if __name__ == "__main__":
    processList = []
    for i in range(4):
        tab = [val for val in range(1+10*i,10*(i+1)+1)]
        process = mp.Process(target=fonction, args=(i,10,tab,))
        processList.append(process)
        process.start()

    for i in range(4):
        processList[i].join()
```

# Semaphore

```
sem = mp.Semaphore(0) # Semaphore avec 0 "jeton"
sem.release() # V(sem)
sem.acquire() # P(sem)
```

# Mutex

```
sem = mp.Lock()
sem.acquire()
sem.release()
```

- Python garanti que le l'on ne peut faire plusieurs release sans avoir eu de acquire entre.

# Memoire Partagée

```
variable = mp.Value('i',5) # crée une variable partagée de type entier contenant 5
variable.value = 8

tab = mp.Array('i',7)  # crée un tableau partagé de 7 entier
tab[0] = 12
```

- Le premier paramètre défini le type utilisé
  - C'est nécessaire pour créer l'espace mémoire à la bonne dimension

# Les types

- ‘c’: ctypes.c_char
- ‘u’: ctypes.c_wchar,
- ‘b’: ctypes.c_byte
- ‘B’: ctypes.c_ubyte,
- ‘h’: ctypes.c_short
- ‘H’: ctypes.c_ushort,
- ‘i’: ctypes.c_int
- ‘I’: ctypes.c_uint,
- ‘l’: ctypes.c_long
- ‘L’: ctypes.c_ulong,
- ‘q’: ctypes.c_longlong
- ‘Q’: ctypes.c_ulonglong,
- ‘f’: ctypes.c_float
- ‘d’: ctypes.c_double

# Pipes

```
import multiprocessing as mp

output,input = mp.Pipe()
input.send("Data")
data = output.recv()
```