

# Linux Embarqué

## Séance 4 : Communication inter-processus

**Laurent Fiack**

**Bureau D212 – [laurent.fiack@ensea.fr](mailto:laurent.fiack@ensea.fr)**

# Menu du jour

Les signaux

Les pipes

Le reste

# Communication entre processus

- Entre un père et son fils
  - Zone de data après `fork()` (mais à sens unique)
  - Paramètres après `exec()`
  - Pipe
- Entre deux processus sans lien
  - Pipe nommé
  - Fichiers
- **Problème** : solutions très lentes

# Les signaux

# Signaux

- Moyen simple de communication entre processus
- Générés soit
  - Exception au cours de l'exécution du processus
  - D'origine matérielle : erreur d'adressage, virgule flottante
  - D'origine logicielle : CTRL-C, SIGCHLD, sur commande
- Lorsqu'un processus reçoit un signal
  - Il est détruit (par défaut)
  - Une routine spéciale est exécutée

# Type de signaux

```

#define SIGHUP          1      /* Hangup (POSIX).  */
#define SIGINT          2      /* Interrupt (ANSI). */
#define SIGQUIT         3      /* Quit (POSIX).   */
#define SIGILL          4      /* Illegal instruction (ANSI). */
#define SIGTRAP         5      /* Trace trap (POSIX). */
#define SIGABRT         6      /* Abort (ANSI).   */
#define SIGIOT          6      /* IOT trap (4.2 BSD). */
#define SIGBUS          7      /* BUS error (4.2 BSD). */
#define SIGFPE          8      /* Floating-point exception (ANSI). */
#define SIGKILL         9      /* Kill, unblockable (POSIX). */
#define SIGUSR1         10     /* User-defined signal 1 (POSIX). */
#define SIGSEGV         11     /* Segmentation violation (ANSI). */
#define SIGUSR2         12     /* User-defined signal 2 (POSIX). */
#define SIGPIPE        13     /* Broken pipe (POSIX). */
#define SIGALRM         14     /* Alarm clock (POSIX). */
#define SIGTERM         15     /* Termination (ANSI). */
#define SIGSTKFLT       16     /* Stack fault. */
#define SIGCHLD         SIGCHLD /* Same as SIGCHLD (System V). */
#define SIGCHLD         17     /* Child status has changed (POSIX). */
#define SIGCONT         18     /* Continue (POSIX). */
#define SIGSTOP         19     /* Stop, unblockable (POSIX). */
#define SIGTSTP         20     /* Keyboard stop (POSIX). */
#define SIGTTIN         21     /* Background read from tty (POSIX). */
#define SIGTTOU         22     /* Background write to tty (POSIX). */
#define SIGURG         23     /* Urgent condition on socket (4.2 BSD). */
#define SIGXCPU         24     /* CPU limit exceeded (4.2 BSD). */
#define SIGXFSZ         25     /* File size limit exceeded (4.2 BSD). */
#define SIGVTALRM       26     /* Virtual alarm clock (4.2 BSD). */
#define SIGPROF         27     /* Profiling alarm clock (4.2 BSD). */
#define SIGWINCH        28     /* Window size change (4.3 BSD, Sun). */
#define SIGPOLL         SIGIO  /* Pollable event occurred (System V). */
#define SIGIO           29     /* I/O now possible (4.2 BSD). */
#define SIGPWR          30     /* Power failure restart (System V). */
#define SIGSYS          31     /* Bad system call. */

```

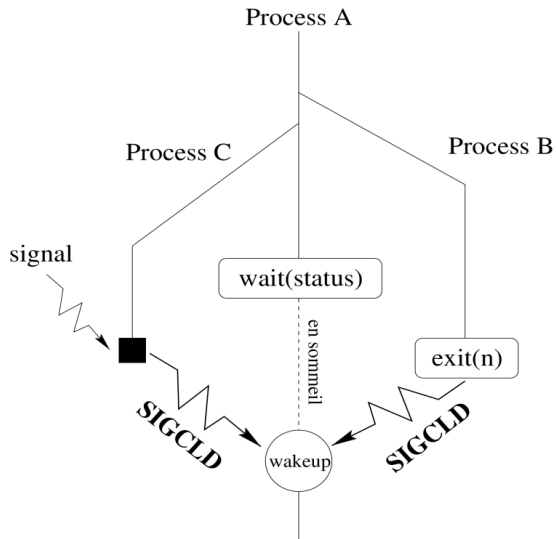
# Signal vs Interruption

## Attention

Les signaux ne sont pas des interruptions !

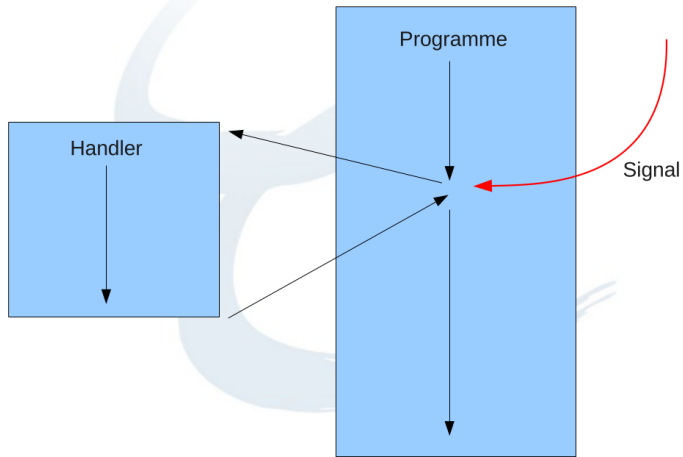
- Lors d'une interruption, la routine associée est exécutée immédiatement
- Lors de la réception d'un signal, le code associé sera exécuté lorsque le processus aura du temps d'exécution disponible sur un CPU
- Les signaux ne sont **pas temps réels** !

# Exemple





# Mise en place d'un handler



# signal()

```
#include <signal.h>

void spint(int sig)
{
    printf("Recu signal d'interruption %d\n",sig);
}

int main()
{
    signal(SIGINT,spint);
    pause();

    exit(0);
}
```

# kill()

```
#include <stdio.h>

void spsig(int sig) {
    printf("Signal %d reçu\n",sig);
}

int main() {
    int idfils, status;
    if (idfils=fork()) {    /* pere */
        sleep(5);
        kill(idfils,SIGUSR1);
        wait(&status);
        exit(0);
    }
    else {                  /* fils */
        signal(SIGUSR1,spsig);
        pause();
        exit(1);
    }
}
```

# Signaux temps-réel

- La norme posix 1.b impose au moins 8 signaux dits "temps-réels"
- Linux en supporte 32 numérotés de 32 (SIGRTMIN) à 63 (SIGRTMAX)
- Les signaux temps-réel n'ont pas de signification prédéfinie
- L'action par défaut est de terminer le processus
- Si des signaux standards et des signaux temps-réel sont simultanément en attente pour un processus Linux donne la priorité aux signaux temps-réel

# Mensonges !

- Le qualificatif "temps-réel" est trompeur !
- **Aucune contrainte temporelle** n'est imposée au niveau de l'ordonnanceur
- Comme pour les signaux standards le signal sera délivré à un processus **uniquement quand celui-ci sera ordonnancé**

# Propriétés

- Un signal temps-réel peut être envoyé par `sigqueue()` et accompagné d'un paramètre.
- Un gestionnaire utilisant l'appel `sigaction()` peut accéder à cette valeur
- Si plusieurs signaux temps-réel sont envoyés ils sont délivrés en commençant par le signal de numéro le moins élevé (le signal de plus fort numéro est celui de priorité la plus faible).
- Les signaux temps-réel du même type sont délivrés dans l'ordre où ils ont été émis.

# Handler

```
void handler(int sig, siginfo_t *info, void *inutile) {
    char *origine;
    printf("Reception du signal n %d ", sig);
    switch (info->si_code) {
        case SI_USER:
            printf("envoye au moyen de kill() ou raise() par le processus %ld\n", info->si_pid);
            break;
        case SI_QUEUE:
            printf("envoyé au moyen de sigqueue() par le processus %ld avec la valeur %d\n",
                (long)info->si_pid, info->si_value.sival_int);
            break;
        default:
            printf("\n");
            break;
    }
}
```

# Handler

```
int main()
{
    pid_t pid_fils;
    switch (pid_fils = (long)fork()) {
        case -1:
            perror("Erreur lors de fork");
            return EXIT_FAILURE;
        case 0:           // Fils
            sigset_t masque; struct sigaction action;
            sigemptyset(&masque);
            sigaddset(&masque, SIGUSR1);
            sigprocmask(SIG_BLOCK, &masque, NULL);
            action.sa_sigaction = handler;
            action.sa_mask = masque;
            action.sa_flags = SA_SIGINFO;
            sigaction(SIGUSR1, &action, NULL);
            sigprocmask(SIG_UNBLOCK, &masque, NULL);
            while(1);

            default:       // Père
                union sigval val;
                val.sival_int = 123;
                sigqueue(pid_fils, SIGUSR1, val);
                wait(NULL);
    }
    return EXIT_SUCCESS;
}
```

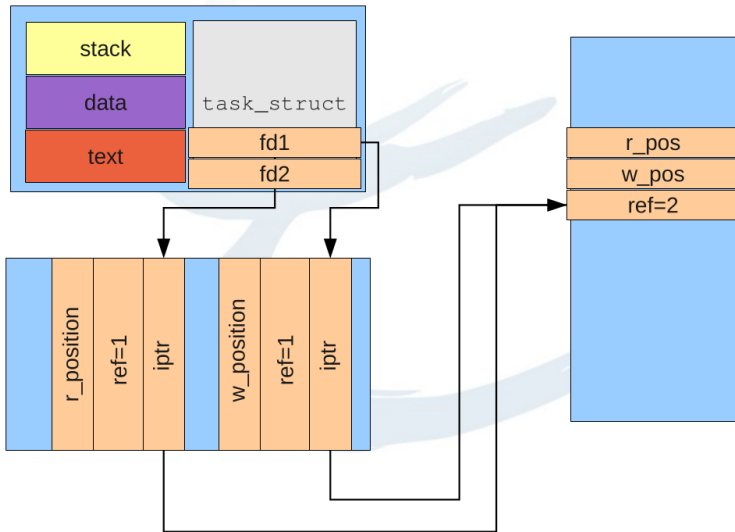


Les pipes

# Pipes

- Un pipe est un fichier virtuel (pas stocké sur un disque)
- Il possède deux descripteurs : un en lecture et un en écriture
- Les descripteurs de fichiers étant partagés par un processus père et son fils, le pipe permet la communication de données entre processus
- Il fonctionne avec un buffer circulaire

# pipe()



# pipe()



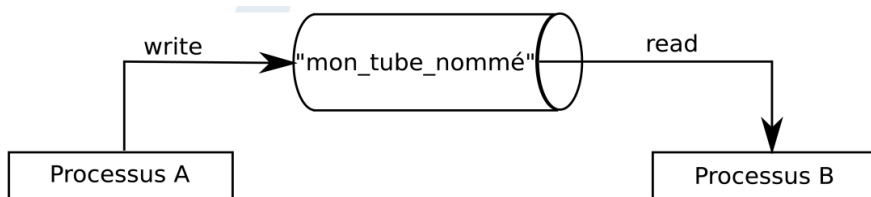
- Le père écrit dans le pipe
- Le fils lit depuis le pipe
- Ou l'inverse...

## Exemple de code

```
int main()
{
    int status, pipedes[2];
    char buf[100];
    int len;
    char msg[]="Salut Fred !";
    if (pipe(pipedes))
        exit(1);
    if (fork()) { /* Code du pere */
        printf("père\n");
        write(pipedes[1],msg,strlen(msg));
        printf("ok, envoyé\n");
        wait(&status);
        return 0;
    }
    else { /* Code du fils */
        printf("fils\n");
        read(pipedes[0],buf,100);
        printf("Mon pere adore l'art de decaler les sons : %s\n",buf);
        return 0;
    }
}
```

# Pipes nommés

- Un fichier spécial est créé (p)
- Un inode est réservé
- N'est pas utilisé comme un fichier mais comme un pipe



# Exemple

```
#define NAME_PIPE "./my_pipe"
```

```
int main(int ac, char **av)
{
    char DATA[32];
    int fd;
    int pid;
    mkfifo(NAME_PIPE, 0666);
    fd = open(NAME_PIPE, O_WRONLY);
    if (fd == -1)
        exit(1);
    pid = getpid();
    sprintf(DATA, "%i dit hello", pid);
    write(fd, DATA, strlen(DATA) + 1);
    sleep(5);
    return 0;
}
```

```
#define NAME_PIPE "./my_pipe"
```

```
#define BUFFSIZE 1024
```

```
int main(int ac, char **av)
{
    char sbBuf[BUFFSIZE];
    int fd;
    int pid;
    fd = open(NAME_PIPE, O_RDONLY);
    pid = getpid();
    read(fd, sbBuf, BUFFSIZE);
    printf("[%i] %s\n", pid, sbBuf);
    return 0;
}
```

Le reste



# Autres types d'IPC

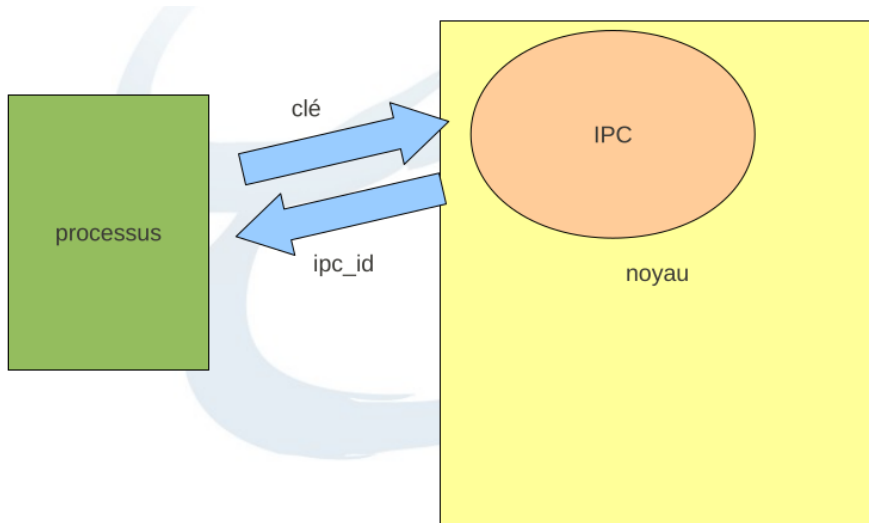
3 types de communication inter processus :

- Message queue
  - Boîtes aux lettres où les processus peuvent déposer ou récupérer des messages
- Mémoire partagée
  - Zone de mémoire dans laquelle les processus peuvent lire et écrire des données
- Sémaphores
  - Structures de données utilisées par les processus pour gérer l'accès aux ressources partagées

# Principe général

- Une clé publique permet d'accéder à un IPC
- Le processus qui crée l'IPC définit son mode d'accès
- La gestion des IPC est un mécanisme du noyau indépendant, ce qui permet d'accéder ou de modifier de manière externe les caractéristiques de l'IPC
- La gestion des IPC est l'un des rôles importants du noyau (présent même dans les micro-noyaux)

# Principe général



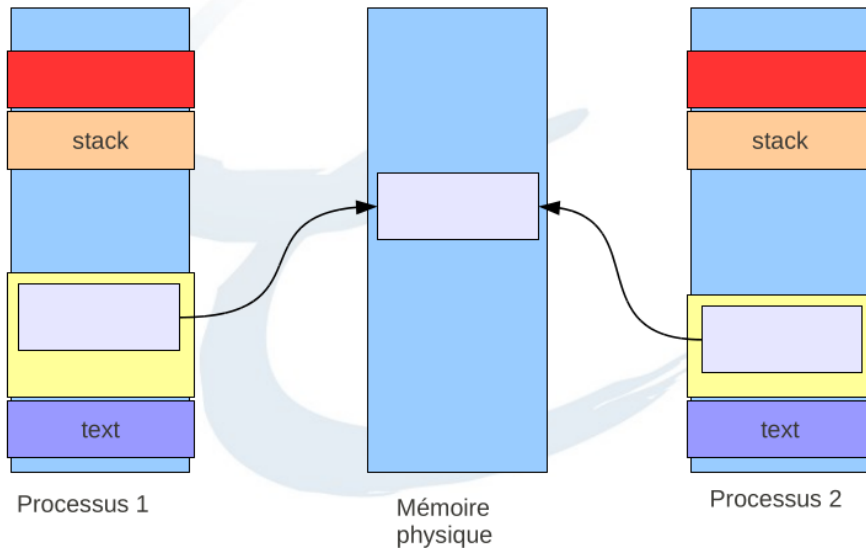
# Mémoire partagée

- Permet d'attacher un bloc de mémoire à un ou plusieurs processus
- Une fois le bloc attaché, il s'utilise comme n'importe quel bloc de mémoire alloué

L'attachement s'effectue en deux temps :

- Création de la mémoire partagée à partir de l'identifiant
  - `shmget()`
- Récupération de l'adresse de la mémoire
  - `shmat()`

# Mémoire partagée



# Création d'une shared memory

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/shm.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

int main(int argc, char * argv[])
{
    int shmid;
    if ((shmid=shmget((key_t)1,100,IPC_CREAT+0666))== -1)
    {
        perror("shmget");
        exit(1);
    }
    printf("%s%d\n", "L'identifiant memoire est ", shmid);
}
```

# Écriture dans une shared memory

```
int main(int argc, char * argv[])
{
    int shmid;
    char *buf;
    char *msg="Message mémoire commune";
    if ((shmid=shmget((key_t)1,0,0))==-1) {
        perror("shmget");
        exit(1);
    }
    if ((buf=shmat(shmid,0,0))== (char *)-1) {
        perror("shmat");
        exit(1);
    }
    *buf=0;                /* segment libre */
    strcpy(buf+1,msg);     /* transmission msg */
    *buf=1;                /* message disponible */
    while(*buf);           /* attend liberation segment */
    shmdt(buf);            /* detachment */
}
```

# Lecture dans une shared memory

```
int main(int argc, char * argv[])
{
    int shmid;
    char *buf;
    char msg[100];
    if ((shmid=shmget((key_t)1,0,0))==-1) {
        perror("shmget");
        exit(1);
    }
    if ((buf=shmat(shmid,0,0))== (char *)-1) {
        perror("shmat");
        exit(1);
    }
    while(*buf!=1);           /* attente msg disponible */
    strcpy(msg,buf+1);        /* lecture message */
    *buf=0;                   /* segment libre */
    printf("%s\n",msg);        /* affichage message */
    shmdt(buf);               /* detachement */
}
```