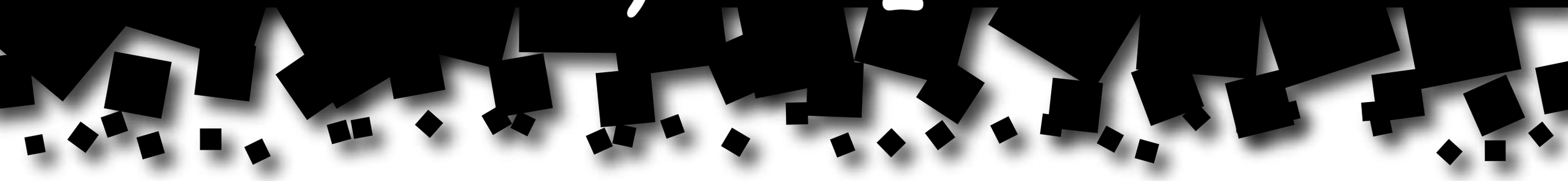
M2101 - Architecture 2

Cours n°6
Communication entre processus

Victor Poupet

Fork, et après ?



À l'aide de la fonction **`fork`** on peut créer de nouveaux processus

- mais chaque processus a ses propres variables, et sa propre mémoire
- comment les faire communiquer pour effectuer une tâche en commun ?

Exemples de communication :

- une application réseau communique avec le serveur distant
- un serveur web démarre des processus fils pour traîter les requêtes
- le shell ordonne l'interruption d'un processus (**`kill`**)
- éviter les conflits lorsque plusieurs processus veulent accéder à un fichier
- deux commandes sont exécutées avec un *pipe* (ex : **`ls | grep toto`**)

Techniques IPC

Il existe de nombreuses méthodes de communication *inter-processus* (IPC en anglais) :

- signaux
- tubes (*pipes*)
- FIFO
- Verrous
- Files de messages
- Sémaphores
- Segments de mémoire partagée
- Fichiers sur mémoire
- Sockets

Signaux

Un processus peut envoyer un "signal" à un autre

- Il existe une liste de signaux qui peuvent être envoyés
- Le destinataire est désigné par son PID
- Chaque processus peut réagir aux signaux à l'aide de fonctions (*handlers*)
- Un processus peut ignorer un signal
- Il existe des handlers par défaut (ex : **SIGINT** provoque par défaut l'arrêt du processus)
- Certains handlers ne peuvent pas être modifiés (ex : **SIGKILL**, **SIGSTOP**)

```
#include <signal.h>

int kill(pid_t pid, int sig);

int raise(int sig);

int sigaction(int sig, const struct
sigaction *act, struct sigaction *oact);
```

Signaux

```
#include <signal.h>

int kill(pid_t pid, int sig);

int raise(int sig);

int sigaction(int sig, const struct
sigaction *act, struct sigaction *oact);
```

Exemples de signaux :

- **SIGINT** (^C)
- **SIGSTOP**
- **SIGCONT**
- **SIGTERM** (kill)
- **SIGKILL** (kill -9)
- **SIGQUIT** (fermeture du terminal)
- **SIGCHLD** (fils terminé ou stoppé)
- **SIGUSR1**
- **SIGUSR2** (pas de comportement par défaut)

Signaux

```c
#include <signal.h>

int kill(pid_t pid, int sig);

int raise(int sig);

int sigaction(int sig, const struct
sigaction *act, struct sigaction *oact);
```

`struct sigaction` est un type contenant plusieurs champs, parmi lesquels :

- **`sa_handler`** : la fonction *handler* (ou **`SIG_IGN`** pour ignorer)
- **`sa_mask`** : liste de signaux à ignorer pendant le traîtement
- **`sa_flags`** : des options

Signaux

```
void sigint_handler(int sig) {
    write(0, "Recu SIGINT!\n", 13);
}

int main(void) {
    void sigint_handler(int sig);
    char s[200];

    struct sigaction sa;
    sa.sa_handler = sigint_handler;
    sa.sa_flags = 0;
    sigemptyset(&sa.sa_mask);
    sigaction(SIGINT, &sa, NULL);

    printf("Entrez du texte: ");
    if (fgets(s, sizeof s, stdin)==NULL)
        perror("fgets");
    else
        printf("message: %s\n", s);
    return 0;
}
```

```
Entrez du texte: bonjour^CReçu SIGINT!
fgets: Interrupted system call
```

`struct sigaction` est un type contenant plusieurs champs, parmi lesquels :

- **`sa_handler`** : la fonction *handler* (ou **`SIG_IGN`** pour ignorer)
- **`sa_mask`** : liste de signaux à ignorer pendant le traîtement
- **`sa_flags`** : des options

Signaux

```
void sigint_handler(int sig) {
    write(0, "Recu SIGINT!\n", 13);
}

int main(void) {
    void sigint_handler(int sig);
    char s[200];

    struct sigaction sa;
    sa.sa_handler = sigint_handler;
    sa.sa_flags = SA_RESTART;
    sigemptyset(&sa.sa_mask);
    sigaction(SIGINT, &sa, NULL);

    printf("Entrez du texte: ");
    if (fgets(s, sizeof s, stdin)==NULL)
        perror("fgets");
    else
        printf("message: %s\n", s);
    return 0;
}
```

```
Entrez du texte: bonjour^CRecu SIGINT!
coucou^CRecu SIGINT!
salut
message: salut
```

`struct sigaction` est un type contenant plusieurs champs, parmi lesquels :

- **`sa_handler`** : la fonction *handler* (ou **`SIG_IGN`** pour ignorer)
- **`sa_mask`** : liste de signaux à ignorer pendant le traîtement
- **`sa_flags`** : des options

Tubes (pipes)

```
#include <unistd.h>
int pipe(int fildes[2]);
```

```
int main(void) {
    int pfds[2];
    char buf[30];

    if (pipe(pfds) == -1) {
        perror("pipe");
        exit(1);
    }

    printf("ecriture (%d)\n", pfds[1]);
    write(pfds[1], "test", 5);
    printf(""lecture (%d)\n", pfds[0]);
    read(pfds[0], buf, 5);
    printf("recu: %s\n", buf);

    return 0;
}
```

Structure abstraite dans laquelle on peut lire et écrire des données

- un appel à la fonction **`pipe`** créé deux descripteurs de fichiers (deux bouts)
- on lit dans le premier descripteur et on écrit dans le second
- les données lues sont supprimées (capacité limitée à environ 10ko)
- c'est le mécanisme utilisé par le shell (ex : **`cat a.txt|head -n 10`**)
- le tube n'est pas accessible par un processus extérieur, mais les descripteurs de fichiers sont copiés par **`fork`**
- les tubes sont unidirectionnels

Tubes (pipes)

```c
int main(void) {
    int pfds[2];
    char buf[30];

    pipe(pfds);
    if (!fork()) {
        printf("(FILS) ecriture\n");
        write(pfds[1], "test", 5);
        printf("(FILS) fin\n");
        exit(0);
    } else {
        printf("(PERE) lecture\n");
        read(pfds[0], buf, 5);
        printf("(PERE) lu: %s\n", buf);
        wait(NULL);
    }
    return 0;
}
```

```
(PERE) lecture
(FILS) ecriture
(FILS) fin
(PERE) lu: test
```

Structure abstraite dans laquelle on peut lire et écrire des données

- un appel à la fonction **`pipe`** créé deux descripteurs de fichiers (deux bouts)
- on lit dans le premier descripteur et on écrit dans le second
- les données lues sont supprimées (capacité limitée à environ 10ko)
- c'est le mécanisme utilisé par le shell (ex : **`cat a.txt|head -n 10`**)
- le tube n'est pas accessible par un processus extérieur, mais les descripteurs de fichiers sont copiés par **`fork`**
- les tubes sont unidirectionnels

Tubes (pipes)

```
ls | wc -l
```

```c
int main(void)
{
    int pfds[2];
    pipe(pfds);

    if (!fork()) {
        close(1); // fermer stdout
        dup(pfds[1]); //stdout = pfds[1]
        execlp("ls", "ls", NULL);
    } else {
        close(0); // fermer stdin
        dup(pfds[0]); // stdin = pfds[0]
        execlp("wc", "wc", "-l", NULL);
    }
    return 0;
}
```

Structure abstraite dans laquelle on peut lire et écrire des données

- un appel à la fonction **`pipe`** créé deux descripteurs de fichiers (deux bouts)
- on lit dans le premier descripteur et on écrit dans le second
- les données lues sont supprimées (capacité limitée à environ 10ko)
- c'est le mécanisme utilisé par le shell (ex : **`cat a.txt|head -n 10`**)
- le tube n'est pas accessible par un processus extérieur, mais les descripteurs de fichiers sont copiés par **`fork`**
- les tubes sont unidirectionnels

Tubes nommés (FIFO)

```c
int main(void) {
    char s[300];
    int num, fd;

    mknod("my_fifo", S_IFIFO | 0666, 0);
    printf("attente...\n");
    fd = open("my_fifo", O_WRONLY);
    printf("connecté !\n");

    while (gets(s), !feof(stdin)) {
        write(fd, s, strlen(s));
    }

    return 0;
}
```

File (*first in, first out*)

- Fonctionne comme un *pipe*, mais a un nom dans le système de fichiers
- Accessible depuis l'extérieur
- Se manipule comme un fichier (**`open`**, **`read`**, **`write`**, etc.)
- **`open()`** bloque jusqu'à ce que les deux extrêmités soient ouvertes
- Possibilité de connecter plusieurs processus à chaque extrêmité

Tubes nommés (FIFO)

```
int main(void) {
    char s[300];
    int num, fd;

    mknod("my_fifo", S_IFIFO | 0666, 0);

    printf("attente...\n");
    fd = open("my_fifo", O_RDONLY);
    printf("connecté !\n");

    do {
        num = read(fd, s, 300);
        s[num] = '\0';
        printf("lu: %s\n", s);
    } while (num > 0);

    return 0;
}
```

File (*first in, first out*)

- Fonctionne comme un *pipe*, mais a un nom dans le système de fichiers
- Accessible depuis l'extérieur
- Se manipule comme un fichier (**`open`**, **`read`**, **`write`**, etc.)
- **`open()`** bloque jusqu'à ce que les deux extrêmités soient ouvertes
- Possibilité de connecter plusieurs processus à chaque extrêmité

Verrous de fichiers

```
struct flock fl;
int fd;

fl.l_type = F_WRLCK;
    // autres types : F_RDLCK, F_UNLCK
fl.l_whence = SEEK_SET;
    // autres : SEEK_CUR, SEEK_END
fl.l_start = 0;
fl.l_len = 0; // tout le fichier
fl.l_pid = getpid();

fd = open("filename", O_WRONLY);

fcntl(fd, F_SETLKW, &fl);
```

On peut verrouiller une portion de fichier

- verrous en lecture ou écriture
  - un seul accès à la fois en écriture
  - plusieurs accès possibles en lecture
- On précise la zone du fichier qui est bloquée (offset de départ, taille)

On peut obtenir la même chose avec des sémaphores

Sémaphores

```
#include <sys/ipc.h>
#include <sys/sem.h>

int semget(key_t key, int nsems, int
semflg);
int semctl(int semid, int semnum, int
cmd, ... /*arg*/);
int semop(int semid, struct sembuf *sops,
unsigned int nsops);
```

Semblable aux verrous, mais plus versatiles

- permettent de gérer des sections critiques (problèmes de concurrence)
- peuvent être utilisés pour verrouiller des fichiers
- sont créés à l'échelle de l'OS (donc accessibles par tous les processus)
- souvent plus rapide que les verrous de fichiers

Attention : les sémaphores systèmes ne sont pas les mêmes que les sémaphores POSIX (vus en TD)

Sémaphores

```
#include <sys/ipc.h>
#include <sys/sem.h>

int semget(key_t key, int nsems, int
semflg);
int semctl(int semid, int semnum, int
cmd, ...);
int semop(int semid, struct sembuf *sops,
unsigned int nsops);
```

Un sémaphore a une valeur (entier relatif)

- la valeur correspond à des passages autorisés
- on peut augmenter la valeur
- on peut demander à diminuer la valeur : on doit attendre que la valeur soit supérieure à ce qu'on veut retirer

Mémoire partagée

```
int shmget(key_t key, size_t size, int
shmflg);
void *shmat(int shmid, void *shmaddr, int
shmflg);
int shmdt(void *shmaddr);
int shmctl(int shmid, int cmd, struct
shmid_ds *buf);
```

```
key_t key;
int shmid;
char *data;

key = ftok("/home/toto/fichier", 'R');
shmid = shmget(key, 1024, 0644 | IPC_
CREAT);
data = shmat(shmid, (void *)0, 0);
if (data == (char *)(-1)) {
    perror("shmat");
}

gets(data);
printf("partagé: %s\n", data);

shmctl(shmid, IPC_RMID, NULL);
```

Bloc de mémoire commun à deux processus

- on obtient un pointeur vers le bloc de mémoire
- toute modification est visible par les autres processus
- le processus peut se détacher du bloc partagé à l'aide de la commande **`shmdt`**
- le bloc doit être libéré explicitement à l'aide de la commande **`shmctl`**

Fichiers sur mémoire

```
void *mmap(void *addr, size_t len, int
prot, int flags, int fildes, off_t off);
int munmap(caddr_t addr, size_t len);
```

```
int fd, pagesize;
char *data;

fd = open("toto", O_RDONLY);
pagesize = getpagesize();
data = mmap((caddr_t)0, pagesize, PROT_
READ, MAP_SHARED, fd, pagesize);
```

Si on veut partager un fichier entre deux processus, il est parfois plus simple de mettre le fichier dans la mémoire et de manipuler des adresses

- on ouvre le fichier avec **`open`**
- on place le fichier en mémoire avec **`mmap`**
  - **`addr`** : adresse où mettre le fichier (en général on laisse l'OS choisir)
  - **`len`** : taille du bloc à mettre en mémoire
  - **`prot`** : permissions du bloc (doit être compatible avec les paramètre de **`open`**)
  - **`flags`** : options
  - **`filedes`** : descripteur obtenu par **`open`**
  - **`off`** : position du début du bloc à mettre en mémoire

Fichiers sur mémoire

```
void *mmap(void *addr, size_t len, int
prot, int flags, int fildes, off_t off);
int munmap(caddr_t addr, size_t len);
```

```
int fd, pagesize;
char *data;

fd = open("toto", O_RDONLY);
pagesize = getpagesize();
data = mmap((caddr_t)0, pagesize, PROT_
READ, MAP_SHARED, fd, pagesize);
```

- Les fichiers en mémoire sont beaucoup plus rapides d'accès (mémoire rapide et pas d'appel système)
- Chaque processus doit placer le fichier en mémoire
- L'espace mémoire est libéré automatiquement lorsque le processus termine

Sockets

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

int socket(int domain, int type, int
protocol);
int bind(int socket, const struct
sockaddr *address, socklen_t address_
len);
ssize_t recv(int socket, void *buffer,
size_t length, int flags);
ssize_t send(int socket, const void
*buffer, size_t length, int flags);
```

Tubes bi-directionnels à travers un réseau

- ne sont pas limitées à une machine
- cf. suite du cours...