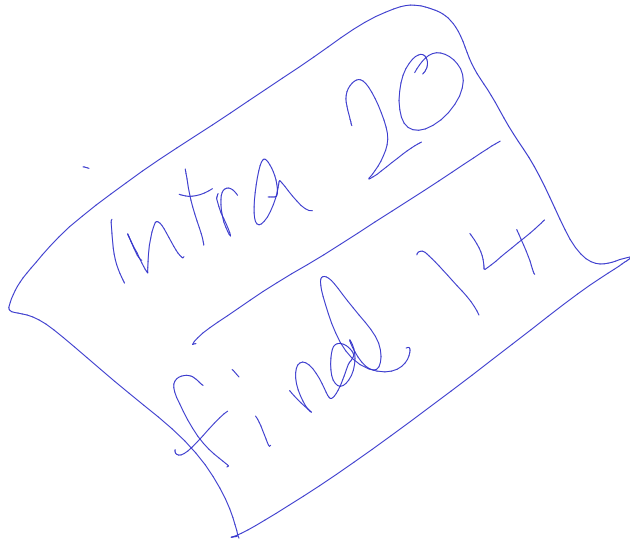


Communication inter-processus



INF3173 – Principes des systèmes d'exploitation
Automne 2024

Francis Giraldeau
giraldeau.francis@uqam.ca

Université du Québec à Montréal



Agenda

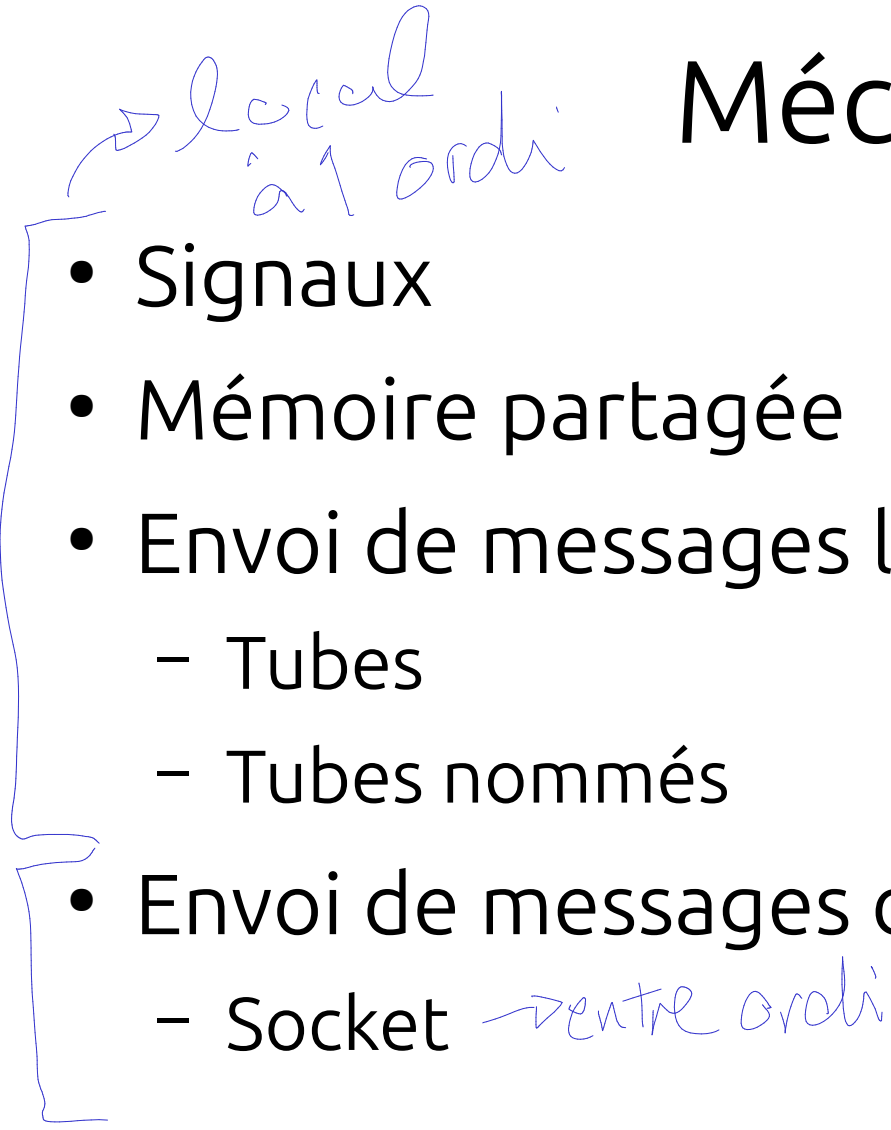
- Introduction
- Survol des méthodes de communication
- Études de cas, exemples, exercices

.

Introduction

- Les processus sont **isolés** par défaut
- Mécanisme pour communiquer nécessaire
- Exemple: processus parent et enfant TP1
 - Il faut indiquer au parent l'échec de exec()
 - Le code de retour est très limité

Mécanismes

- 
- Signaux
 - Mémoire partagée
 - Envoi de messages locaux
 - Tubes
 - Tubes nommés
 - Envoi de messages distribués
 - Socket *entre ordi*

Signaux

- Forme d'interruption logicielle
- Peut survenir à n'importe quel moment
 - S'imbrique par dessus le contexte d'exécution principal!
- L'application peut définir une fonction de rappel par signal
- L'application peut masquer (ignorer) certains signaux temporairement

Fonctionnement signaux

- Le système d'exploitation sauvegarde l'état du programme (registres et instruction courante)
- Ajoute sur la pile les informations suivantes:
 - Les signaux masqués
 - L'adresse de l'instruction courante
 - Information à propos du signal
- Le programme reprend l'exécution dans le gestionnaire de signal (fonction de rappel)
- Au retour de cette fonction, le contexte d'origine est restauré.

Exemple: SIGINT

```
void sigint_handler(int sig) {  
    printf("DEBUG: sigint_handler %d\n", sig);  
}  
  
int main() {  
    // Définir la fonction de rappel pour SIGINT  
    signal(SIGINT, sigint_handler);  
  
    printf("Faire CTRL-C pour signaler SIGINT\n");  
  
    // Traitement...  
    struct timespec ts = {.tv_sec = 1, .tv_nsec = 0};  
    int count = 0;  
    while (1) {  
        printf("count %d\n", count++);  
        nanosleep(&ts, NULL);  
    }  
  
    printf("Terminé\n");  
    return 0;  
}
```

On affiche un message, mais on ne quitte pas!

Surcharge de la fonction de rappel

Le signal est envoyé si le programme est en exécution ou s'il est bloqué

kill -SIGKILL 67690 man 7 signal

Pgrep nom Kill -s SIGSTOP → Blocker (SIGCONT) SIGCONT

Exemple: SIGSEGV

```
// Structure de donnée pour setjump
jmp_buf jmpbuffer;
```

```
void segfault_handler(int sig) {
    printf("Signal SIGSEGV %d\n", sig);
    longjmp(jmpbuffer, 1);
}
```

Récupérer de l'erreur avec longjmp()

STOP et kill est pas ignorable
یعنی راجع به سیگنال است
ولی این درآوردن سیگنال

Si on a pas ça, → boucle infinie

```
int main() {
    printf("Démarrage...\n");
```

```
// Surcharger la fonction de rappel pour SIGSEGV
signal(SIGSEGV, segfault_handler);
```

```
// Initialiser jump buffer
```

```
if (setjmp(jmpbuffer) == 0) {
```

```
// Exécution normale
```

```
// Causer SIGSEGV en accédant un pointeur NULL
```

```
printf("KABOUM!\n");
```

```
int* ptr = NULL;
```

```
*ptr = 42;
```

```
} else {
```

```
// setjmp(jmpbuffer) retourne 1 à cause de longjmp
```

```
printf("On a survécu à l'erreur de segmentation!\n");
```

```
}
```

```
printf("Fin normale du program\n");
```

```
return 0;
```

```
}
```

Déréférencement d'un pointeur NULL

L'exécution se poursuit ici

Exemple minuterie

→ forcer accès à la variable

```
volatile int run = 1;
```

```
void alarm_handler(int sig) {  
    printf("DEBUG: alarm_handler %d\n", sig);  
    run = 0;  
}
```

```
int main() {  
    // Surcharger le gestionnaire de signal SIGALRM  
    signal(SIGALRM, alarm_handler);  
    // Demander un signal dans 1 seconde. Contrairement à nanosleep,  
    // Le programme continue à s'exécuter  
    alarm(1);  
  
    printf("Travail...\n");  
    volatile unsigned long count = 0;  
    while (run) {  
        count++;  
    }  
  
    printf("Fin du programme count=%ld\n", count);  
    return 0;  
}
```

La boucle se termine
après 1 seconde

SIGUSR1 → *Servicio personalizado*

Signaux d'intérêt

- SIGINT : interruption du clavier
- SIGALRM : expiration d'une minuterie
- SIGSEGV: erreur de segmentation
- SIGFPE: erreur de point flottant
- SIGCHLD: processus enfant a quitté → *à la place wait*
- SIGTRAP : point d'arrêt (débugueur)
- SIGTERM : demande de quitter
↳ *quitter proprement*
- SIGSTOP : mise en pause du processus
- SIGKILL : Sashay away
↳ *force quitter immédiatement*

SIGSTOP et SIGKILL
ne peuvent pas être
bloqués

Envoyer un signal

```
int result = kill(pid, SIGUSR1);
```

- Appel système `kill()` : peut envoyer n'importe quel signal (pas seulement SIGKILL)
- Un processus peut envoyer un signal à un autre processus
- Si `pid > 0` le signal `sig` est envoyé à ce processus
- Si `pid = 0` le signal est envoyé à tous les processus du groupe de l'appelant
- Les deux processus doivent appartenir au même utilisateur
- Seul le super-utilisateur (root) peut envoyer un signal au processus d'un autre utilisateur
 - Sinon, un utilisateur pourrait par exemple terminer les processus d'un autre utilisateur!

Intercepter un signal

- L'appel système `sigaction()` permet de redéfinir le gestionnaire associé à un signal.
- La structure `sigaction` :
 - La fonction de rappel : `void (*sa_handler)(int)`
 - L'ensemble des signaux à bloquer pendant l'exécution de la fonction de rappel: `sigset_t sa_mask`
 - Options : `int sa_flags`
- On peut associer un même gestionnaire à des signaux différents.

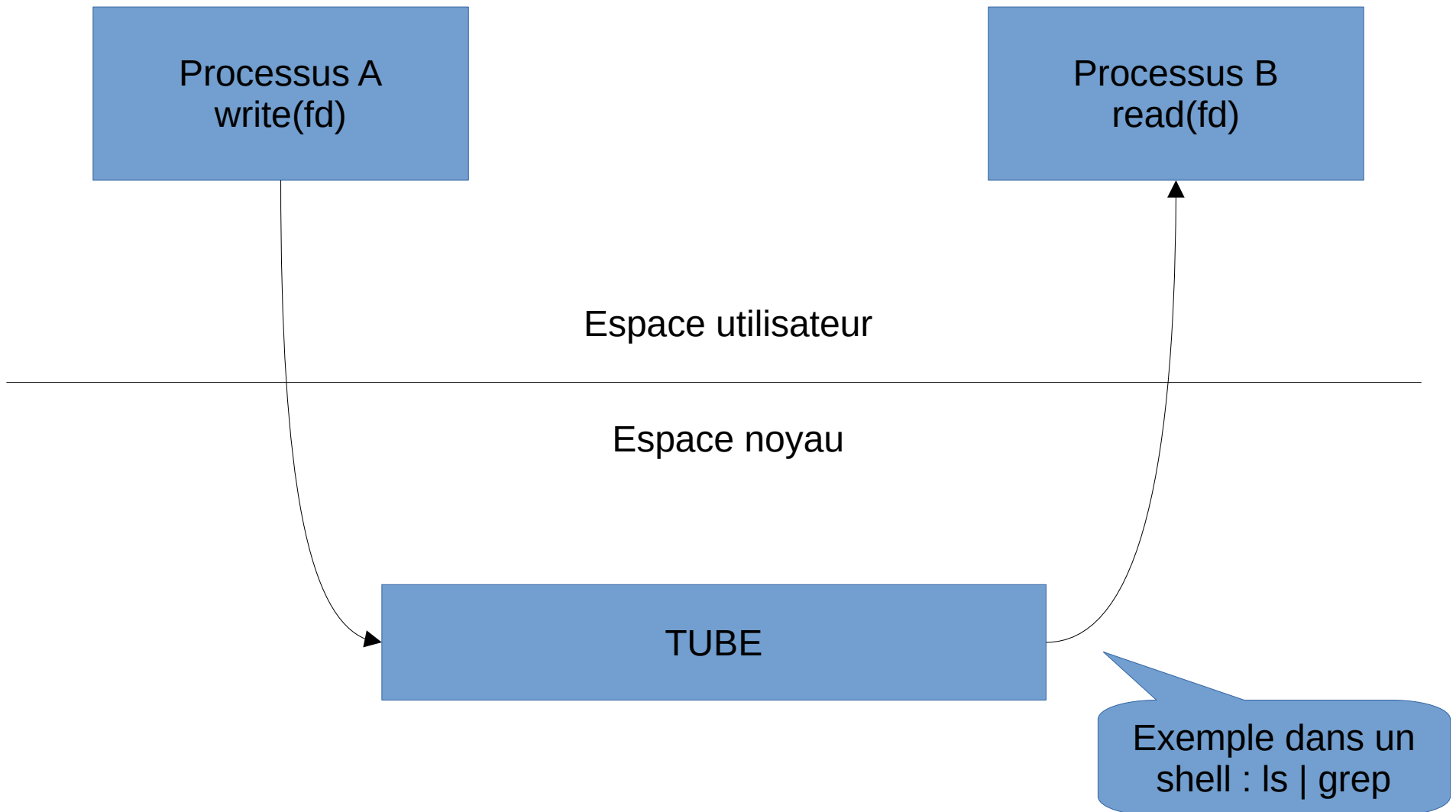
Bloquer et attendre un signal

- `sigprocmask()` : empêcher certains signaux de survenir (sauf `SIGKILL` et `SIGSTOP`)
 - Par exemple, si on ne veut pas être dérangé
- `pause()` : attend jusqu'au prochain signal
- `sigsuspend(mask)` : combinaison de `sigprocmask()` suivi de `pause()`

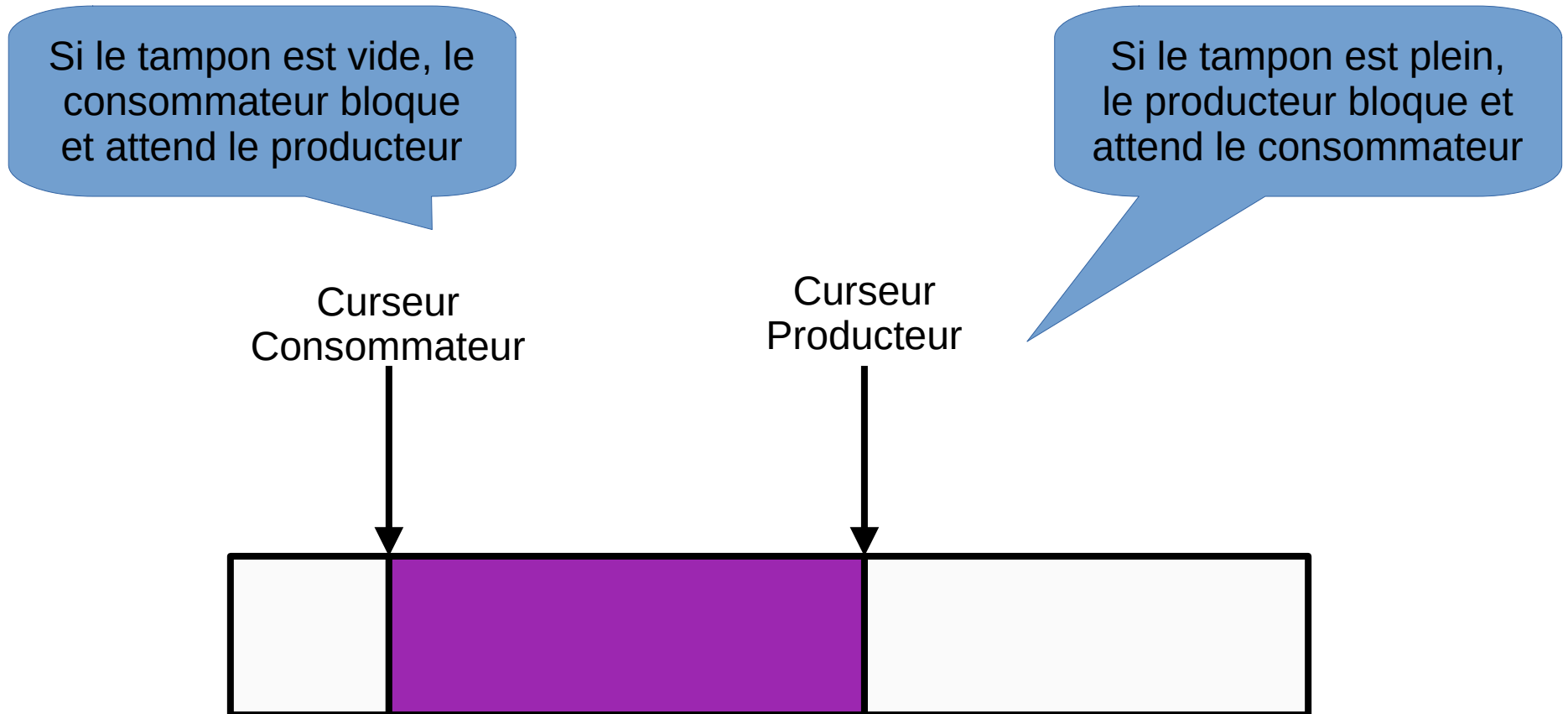
Mémoire partagée

- Principe : référencer les mêmes pages mémoires dans deux processus différents
- `shm_open()` : ouvrir un descripteur de fichier sous `/dev/shm`
- `ftruncate()` et `mmap()` requis ensuite pour accéder à l'espace
→ enlever la lien
- `shm_unlink()` : supprime l'espace partagé lorsque le dernier utilisateur a terminé
*shm open → 1
" " → 2
1 → shm_unlink*
- Commande `ipcs` : lister les ressources partagées
*0 4 4
l'espace
memoire
supprime!*
- Attention : conditions critiques possibles sur des accès concurrents à la mémoire. Nécessite une synchronisation (prochain cours)

Tubes



Fonctionnement



Exemple tube (1)

Début : vide



write(10)



read(5)



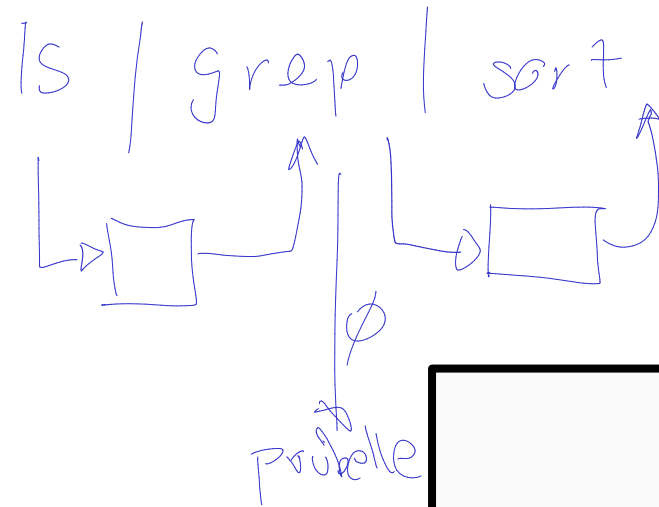
write(10)



read(5)

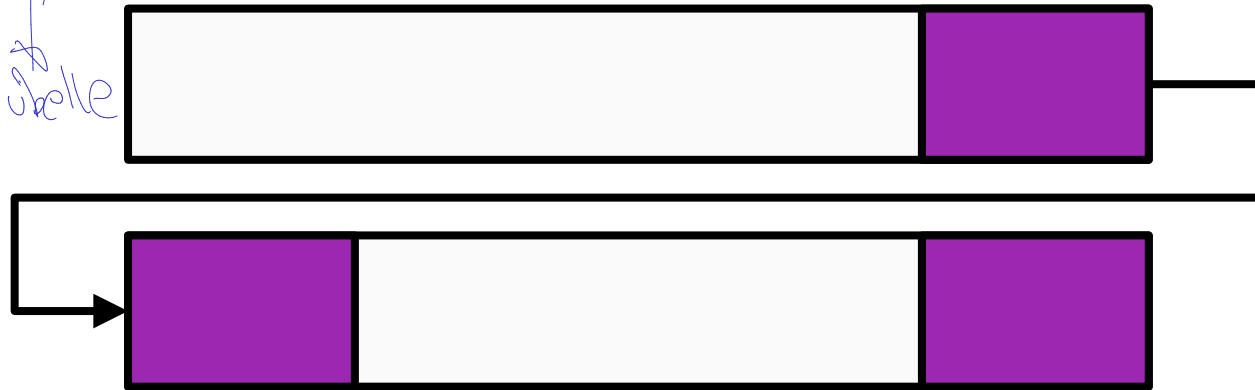


Exemple tube (3)

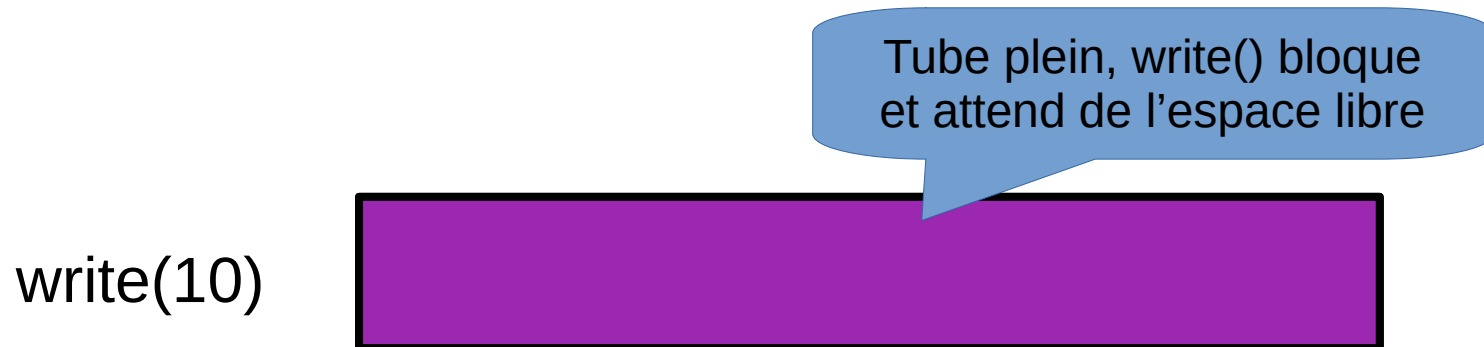
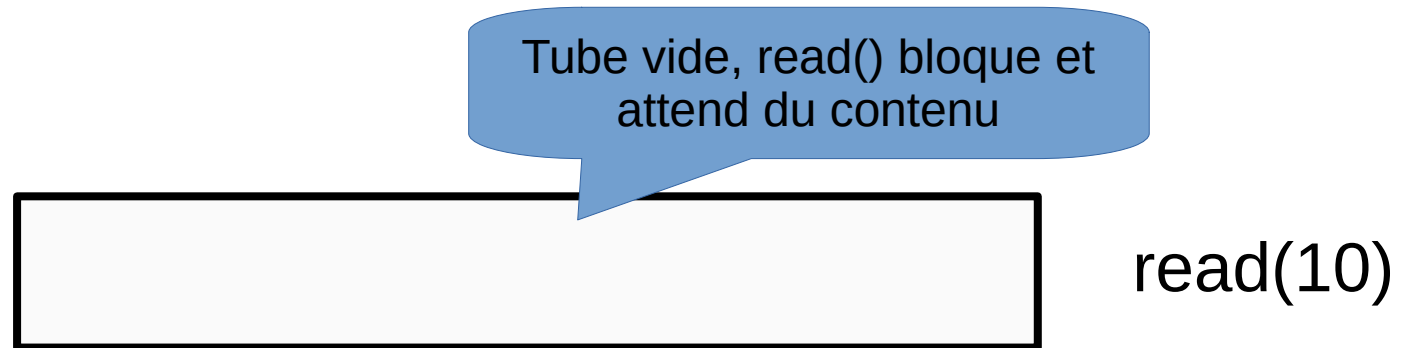


Tampon circulaire :
Quand on est arrivé à la fin,
on recommence au début

write(10)



Exemple tube (2)

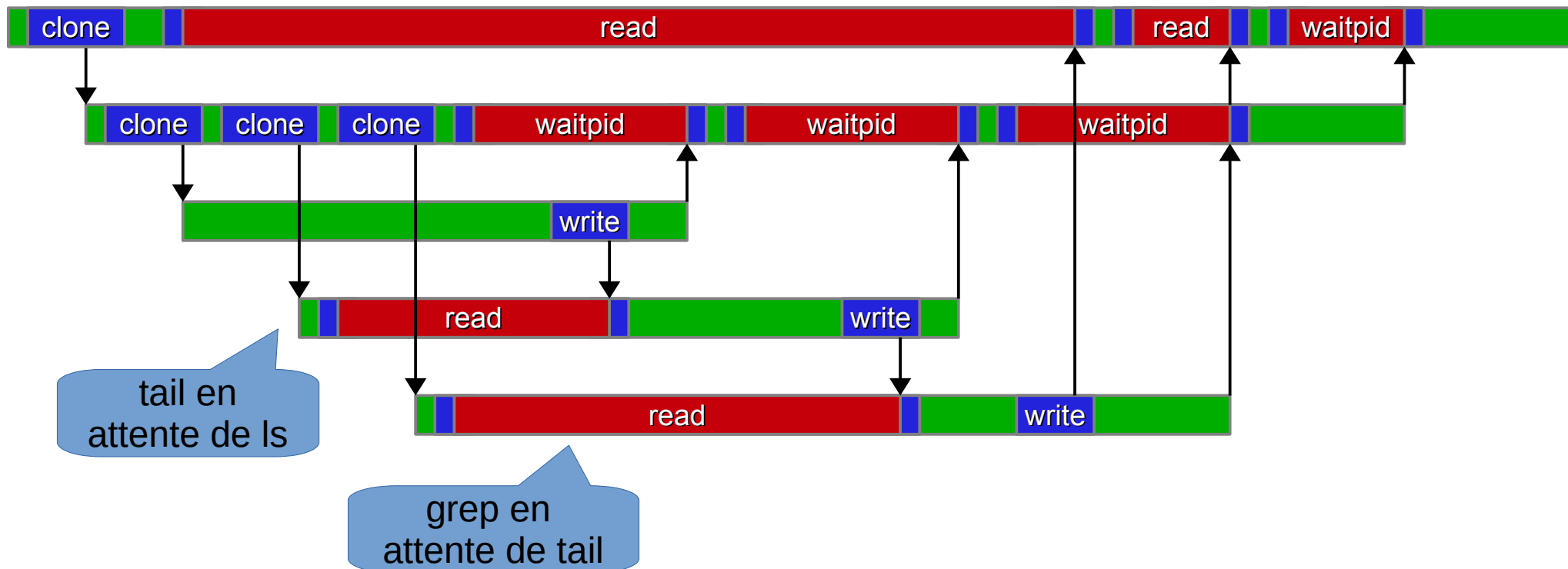


Tubes anonymes

- Liaisons unidirectionnelles de communication
- Se comporte comme une file FIFO
- Accédés par deux descripteurs de fichiers (lecture et écriture)
- Taille limitée (mémoire en mode noyau)
- La lecture dans un tube est destructrice : l'information ne peut être lue qu'une seule fois dans un tube
- Le tube est détruit lorsque tous les descripteurs du tube seront fermés

Exemple d'exécution d'un script shell : ls | tail | grep

- 1 sh
- 2 sh
- 3 ls
- 4 tail
- 5 grep



Les tubes anonymes: création

- Un tube de communication anonyme est créé par l'appel système:
`int pipe(int p[2]).`
- Cet appel système crée deux descripteurs de fichiers. Il retourne, dans p, les descripteurs de fichiers créés :
 - p[0] contient le descripteur réservé aux lectures à partir du tube
 - p[1] contient le descripteur réservé aux écritures dans le tube.
- Les descripteurs créés sont ajoutés à la table des descripteurs de fichiers du processus appelant.
- Seul le processus créateur du tube et ses descendants (ses fils) peuvent accéder au tube (duplication de la table des descripteurs de fichiers).
- Si le système ne peut pas créer de tube pour manque d'espace, l'appel système pipe() retourne la valeur -1, sinon il retourne la valeur 0.
- L'accès au tube se fait via les descripteurs (comme pour les fichiers ordinaires).

Les tubes anonymes: création (3)

- Les tubes anonymes sont, en général, utilisés pour la communication entre un processus père et ses processus fils, avec un processus qui écrit sur le tube, appelé processus écrivain, et un autre qui lit à partir du tube, appelé processus lecteur.
- La séquence d'événements pour une telle communication est comme suit :
 1. Le processus père crée un tube de communication anonyme en utilisant l'appel système `pipe()` ;
 2. Le processus père crée un ou plusieurs fils en utilisant l'appel système `fork()` ;
 3. Le processus écrivain ferme le descripteur de fichier, non utilisé, de lecture du tube ;
 4. De même, le processus lecteur ferme le descripteur de fichier, non utilisé, d'écriture du tube ;
 5. Les processus communiquent en utilisant les appels système:
`read(fd[0], buffer, n)` et `write(fd[1], buffer, n)`;
 1. Chaque processus ferme son fichier lorsqu'il veut mettre fin à la communication via le tube.

```
// macros utiles pour se souvenir que read=0 et write=1
#define R 0
#define W 1

int main() {
    int fd[2];           // tableau de descripteurs de fichiers
    pipe(fd);            // nouveau tube sans nom
    char message[100];   // tampon du message
    int nbocets;
    char* msg = "foo bar";
    if (fork() == 0) {
        /*
         * Producteur
         *
         * L'enfant ferme le descripteur de lecture non utilisé,
         * écrit le message dans le tube et ferme le descripteur
         * d'écriture.
         */
        printf("Écriture du message %s\n", msg);
        close(fd[R]);
        write(fd[W], msg, strlen(msg) + 1);
        close(fd[W]);
    } else {
        /*
         * Consommateur
         *
         * Le parent ferme le descripteur d'écriture non utilisé,
         * copie le message du tube vers le tampon, affiche le message
         * puis ferme le descripteur de lecture
         */
        close(fd[W]);
        nbocets = read(fd[R], message, 100);
        printf("Lecture %d octets : %s\n", nbocets, message);
        close(fd[R]);
    }
    return 0;
}
```

→ descripteur de fichier

\$./620-tube

Écriture du message foo bar
Lecture 8 octets : foo bar

Les tubes anonymes

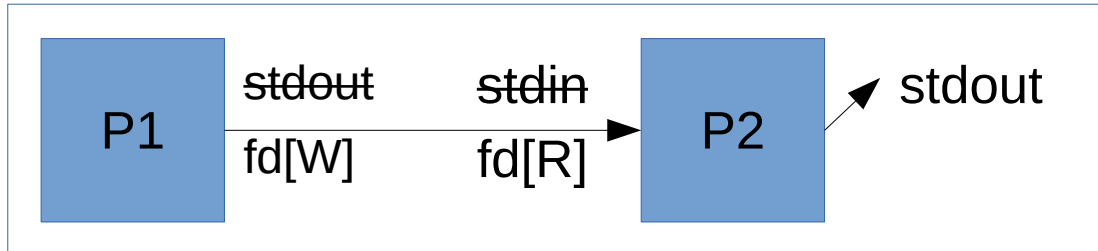
- Chaque tube a un nombre de lecteurs et un nombre d'écrivains
- La fonction `read()` d'un tube retourne 0 (fin de fichier), si le tube est vide et le nombre d'écrivains est 0
- L'oubli de la fermeture de descripteurs peut mener à des situations d'interblocage (attente indéfinie)
- La fonction `write()` dans un tube génère le signal `SIGPIPE`, si le nombre de lecteurs est 0
- Par défaut, les lectures et les écritures sont bloquantes

Redirection de stdin et stdout

- La duplication de descripteur permet à un processus de créer un nouveau descripteur (dans sa table des descripteurs) synonyme d'un descripteur déjà existant.
- `dup()` : duplique un descripteur de fichier
- `dup2()` : duplique un descripteur de fichier avec
- Ces fonctions peuvent être utilisées pour réaliser des redirections des fichiers d'entrées et sorties standards vers les tubes de communication.

Redirection

Exécution parallèle de deux commandes shell. Un tube connecte stdin de la première vers stdout de la deuxième.



```
// macros utiles
#define R 0
#define W 1

int main(int argc, char* argv[]) {
    int fd[2];
    pipe(fd); // creation d'un tube sans nom

    if (fork() > 0) { // parent

        close(fd[R]); // fermeture du descripteur de lecture non utilisé
        dup2(fd[W], 1); // copie fd[W] en tant que descripteur 1 (stdout)
        close(fd[W]); // fermeture du descripteur d'écriture

        // exécute le producteur
        if (execlp(argv[1], argv[1], NULL) == -1) {
            perror("execlp parent");
        }
    } else { // enfant

        close(fd[W]); // fermeture du descripteur d'écriture non utilisé
        dup2(fd[R], 0); // copie fd[R] dans le descripteur 0
        close(fd[R]); // fermeture du descripteur de lecture

        // exécute le consommateur
        if (execlp(argv[2], argv[2], NULL) == -1) {
            perror("execlp enfant");
        }
    }

    return 0;
}
```

./s09-tube-redirection ls wc

échange 2 descripteur de fichier

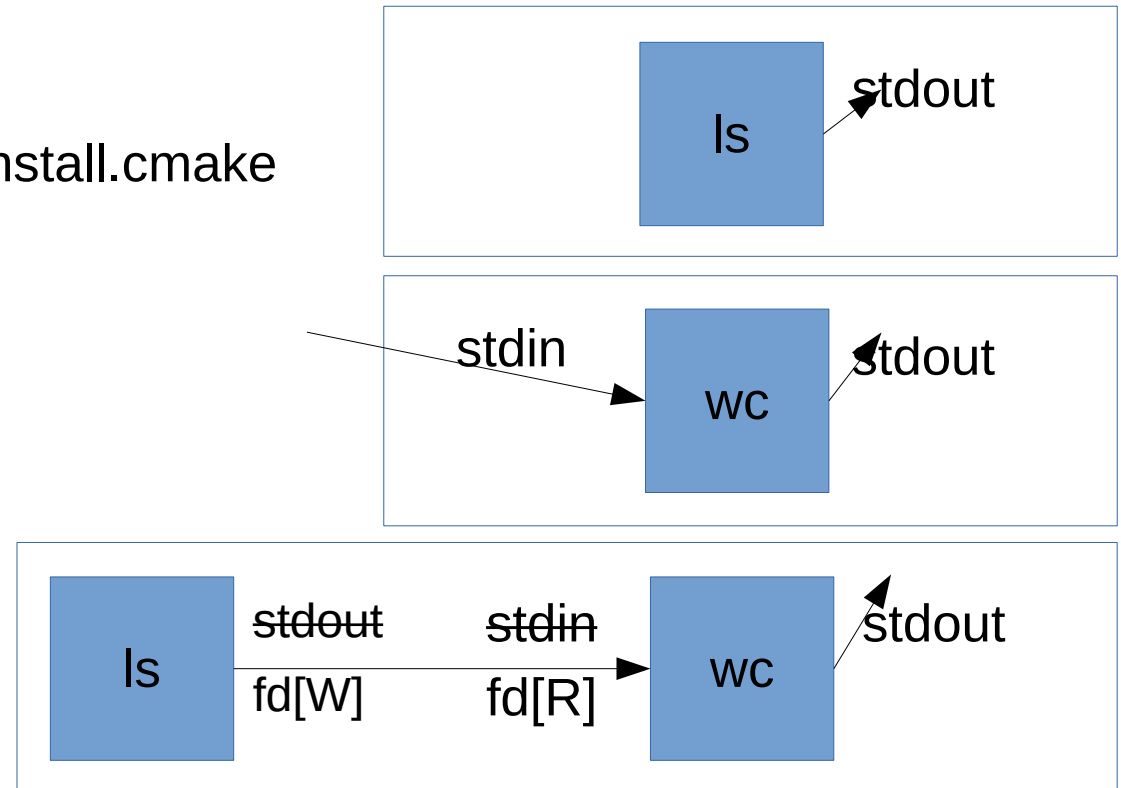
Exemple redirection (2)

\$ ls

621-redirect CMakeFiles cmake_install.cmake
CTestTestfile.cmake Makefile

\$ wc

\$./621-redirect ls wc
3 15 132



Les tubes anonymes: remarques

- Attention aux chaines de caractères
- Deux protocoles habituels :
 - Terminé par zéro '\0'
 - Écrire la taille, puis les données (on peut omettre le caractère nul final)
- La communication bidirectionnelle est possible en utilisant deux tubes (un pour chaque sens de communication).

Les tubes nommés

- Les tubes de communication nommés fonctionnent aussi comme des files de discipline FIFO (first in first out)
- Ils sont plus polyvalents que les tubes anonymes car ils offrent, en plus, les avantages suivants :
 - Ils ont chacun un nom qui existe dans le système de fichiers
 - Ils peuvent être utilisés par des processus indépendants (pas limité à la relation parent-enfant)
 - Ils existeront jusqu'à ce qu'ils soient supprimés explicitement
- Créés par la commande « mkfifo » ou « mknod » ou par l'appel système `mknod()` ou `mkfifo()`

Socket

- Connexion locale ou réseau
- Permet d'échanger des messages entre deux processus sur des ordinateurs différents (qui ne partagent pas de mémoire)
- Canal **bidirectionnel**
 - Contrairement aux tubes qui sont unidirectionnels
 - Permet `read()` et `write()` sur le même descripteur
- Abstraction de la complexité de la communication
 - Division en paquets, retransmission, etc.

Serveur

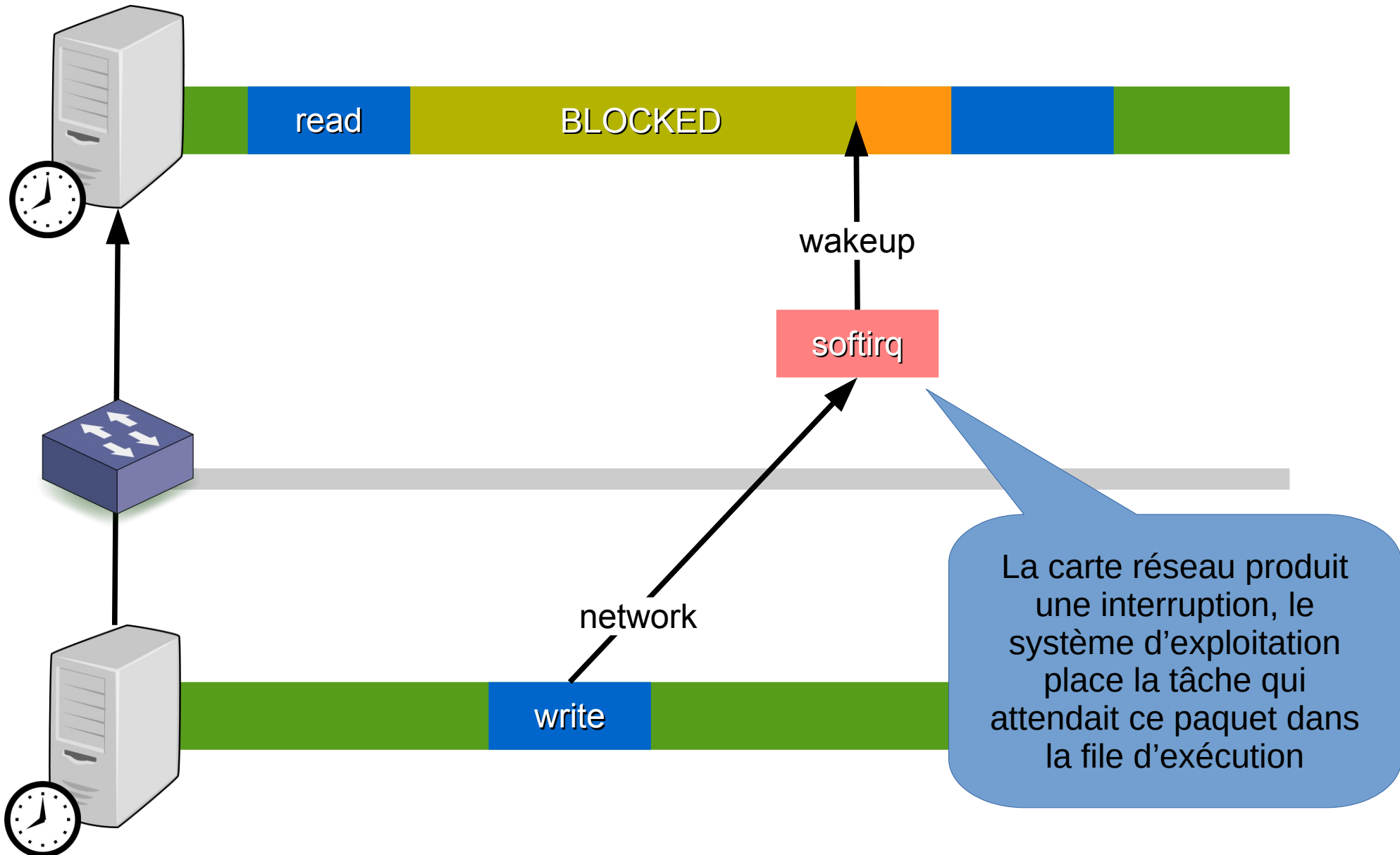
- `socket()` : création du socket
- `bind()` : utilisation du port
- `listen()` : accepter des connexions
- `accept()` : attend une connexion entrante

Client

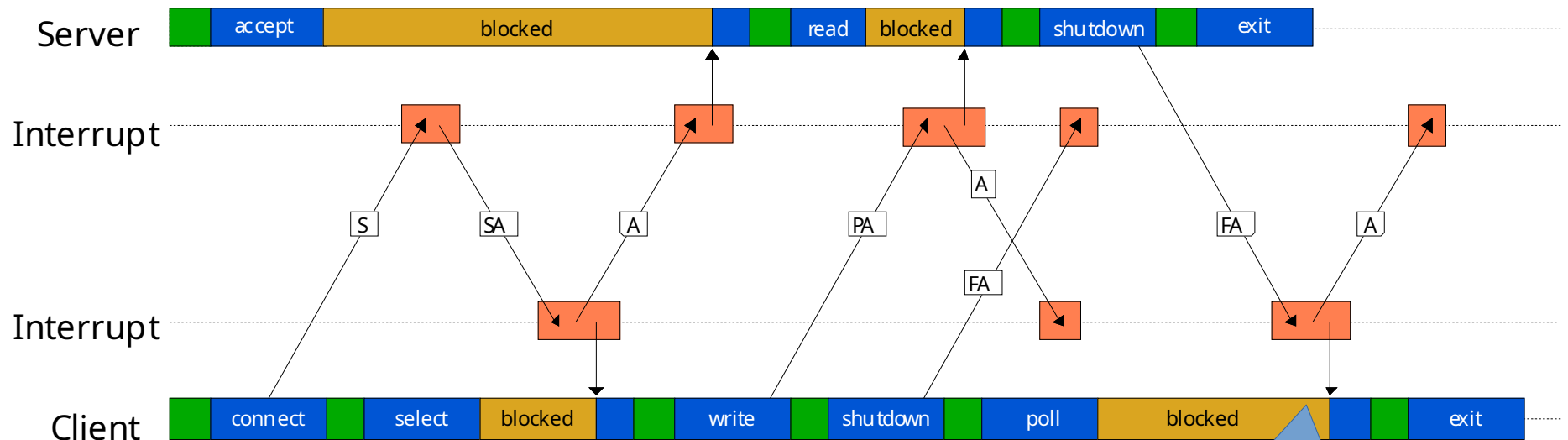
- `socket()` : création du socket
- `connect()` : établissement de la connexion

`read()/write()`
`send(),recv()`

Réveil par un programme distant



Communication TCP



Le système d'exploitation gère le détail de la connexion pour le programme

Exemple socket : serveur Web

Client

```
fd = socket();  
connect(fd, "server.com");  
  
write(fd, "GET index.html");  
read(fd, html);  
  
close(fd);
```

Serveur

```
socket();  
bind(port);  
listen();  
while(1) {  
    fd = accept();  
  
    read(fd, url);  
    write(fd, html);  
  
    close(fd);  
}
```

Sockets : lectures courtes

- À cause d'un délai réseau, il se peut que la quantité à lire soit inférieure à celle attendue.
- Il faut s'attendre à recevoir l'information par fragment : c-à-d lire dans une boucle
- Tant que la quantité demandée n'est pas atteinte, alors on fait une réception
- Voir exemple s09-shortread-client.c

Connect → il prend
un de ret