COMMUNICATION ET SYNCHRONISATION ENTRE PROCESSUS UNIX

Michel Riveill – riveill@unice.fr

Le menu du jour

- La vie des processus Unix → rappel
  - Clonage
  - Création par remplacement
- Communication entre processus Unix
  - Les tubes (pipes) → rappel
  - Les tubes nommés
  - Les fichiers couplé en mémoire → cf. TD
  - Les sockets → pas dans ce cours
  - La mémoire partagée
  - Les files de messages
- Synchronisation entre processus UNIX
  - Les signaux → rappel
  - Les verrous
  - Les sémaphores

3

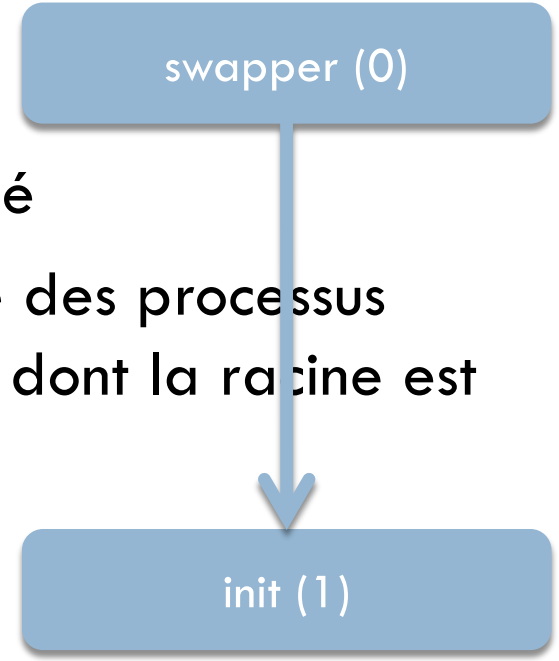
La vie des processus Unix

Clonnage

Remplacement

La vie des processus Unix

- Lors de sa création, tout processus reçoit un numéro unique (entier positif)qui est son identificateur (pid).
- Tout processus est créé par un autre processus, excepté le processus initial, de nom swapper et de pid 0, créé artificiellement au chargement du système
- Le swapper crée alors un processus appelé init, de pid 1, qui initialise le temps-partagé
- Par convention, on considère que l'ensemble des processus existants à un instant donné forme un arbre dont la racine est le processus initial init.



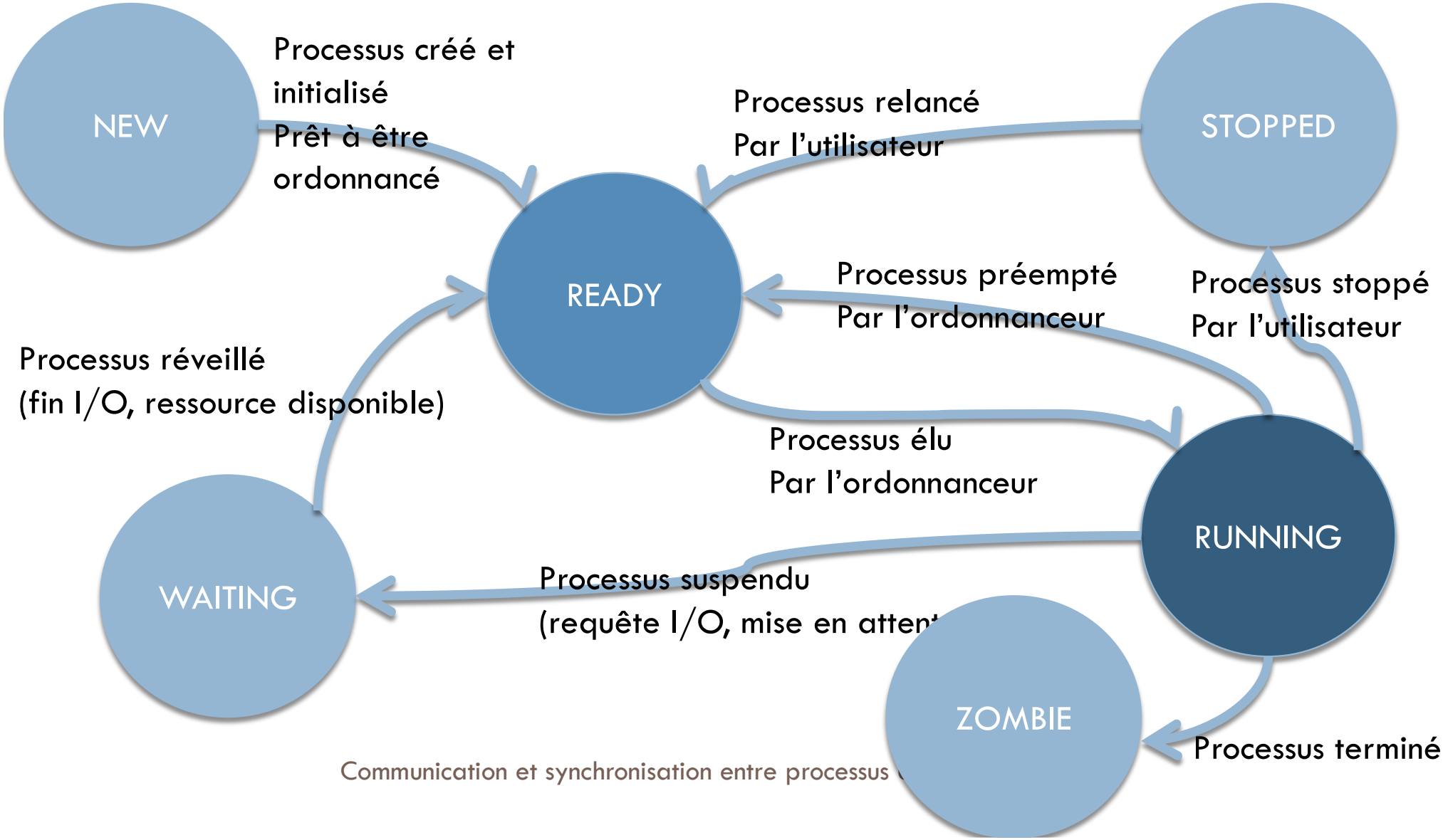
La vie des processus

- Un processus qui s'exécute lâche le processeur de manière
  - Volontaire → multi-tâche coopératif
    - L'ordonnancement dépend de l'application
    - Possibilité de monopoliser le système
  - Forcé → multi-tâche préemptif
    - Protection de l'OS
    - Permet d'assurer un équilibre entre les processus (et donc les utilisateurs)
- Typiquement, un processus est interrompu (préempté)
  - Après un certain temps (time slice)
  - En cas de terminaison d'une entrée/sortie
  - Si un autre processus à une plus haute priorité

La vie des processus – changement de contexte

- Le basculement d'un processus à un autre est géré par le noyau
  - Suspendre le processus P0
  - Sauvegarder le contexte du processus P0
  - Restaurer le contexte du processus P1
  - Reconfigurer l'espace mémoire
    - En particulier reconfiguration du MMU (Memory Management Unit)
    - Rappel : *le MMU gère la correspondance adresse virtuelle / adresse physique*
  - Démarrer le processus P1
- Cette opération est généralement décidée et réalisée par l'ordonnanceur (scheduleur)
  - Opération coûteuse

La vie des processus - état



Création de processus – fork() /exec()

- La création d'un nouveau processus Unix passe par deux mécanismes complémentaires :
  - la duplication d'un processus existant provoqué par la fonction
    - fork ()
  - le remplacement d'un processus par un nouveau code provoqué par la fonction
    - exec ()
- Ces mécanismes sont tels que les processus ainsi créés pourront :
  - se synchroniser : envoi de signaux (appel système kill ()), déroutement des fonctions de traitement des signaux (appel système signal ()), mise en attente (appel système wait ()), ...
- Communiquer entre eux
  - appel système pipe ()

Duplication de processus – fork()

- La fonction système fork() permet de dupliquer un processus en créant dynamiquement un nouveau processus analogue au processus initial
- Le processus créé (processus fils) hérite du processus père de certains de ses attributs :
  - le même code,
  - une copie de la zone de données,
  - une copie de l'environnement,
  - les différents propriétaires,
  - une copie de la table des descripteurs de fichiers,
  - une copie de la table de traitement des signaux, ...
- Le père et le fils ne se distingue que par la valeur de retour de la fonction fork () :
  - 0 dans le processus fils
  - Le pid du fils dans le processus père

Duplication de processus – fork()

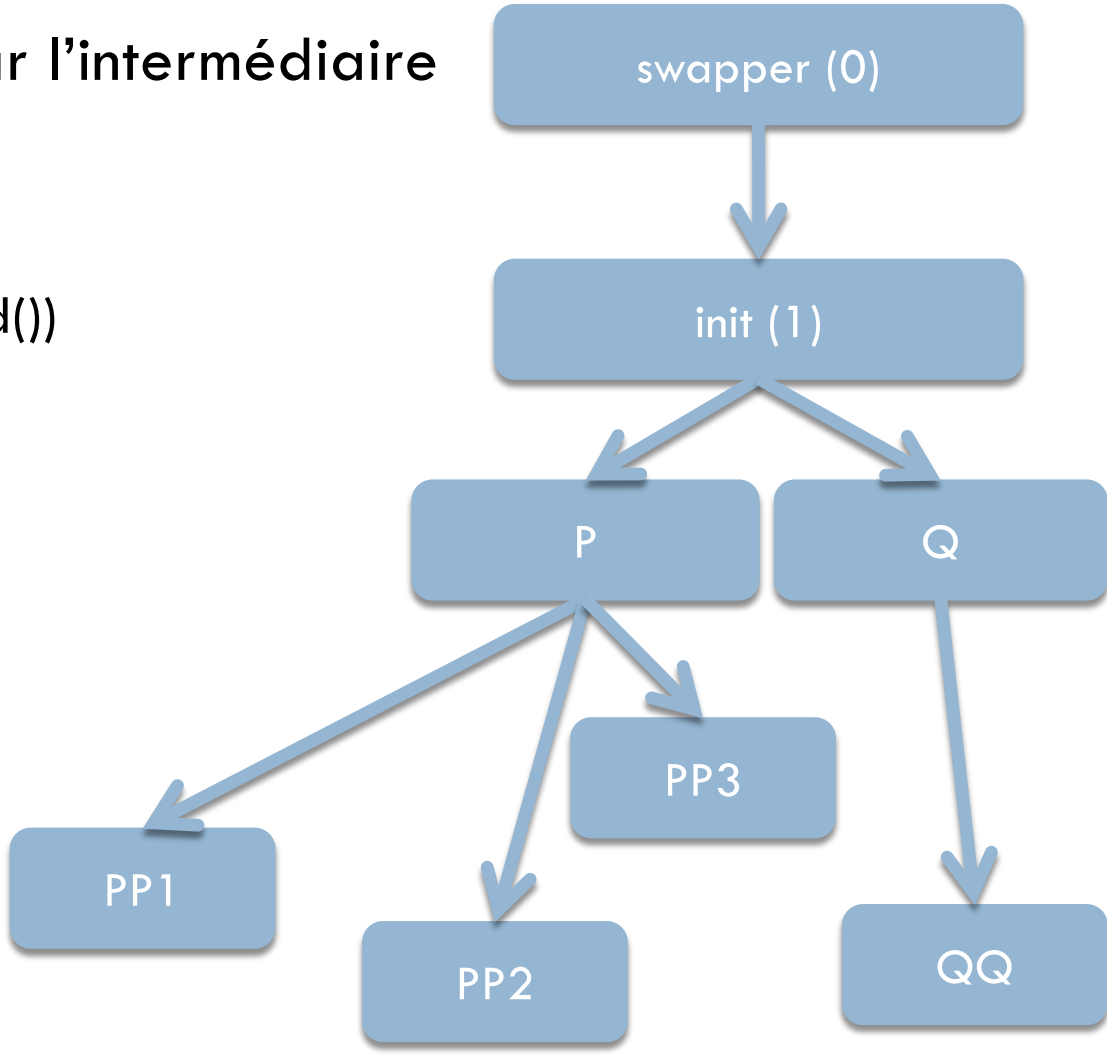
- Le processus fils hérite de beaucoup d'attribut du père comme les descripteurs de fichiers ouverts, mais il n'hérite pas de :
  - L'identité du père
  - Du temps d'exécution (remis à 0)
  - Des signaux pendant du père
  - De la priorité du père (la sienne est initialisée à la valeur standard)
  - Des verrous sur les fichiers détenus par le père
- Pour optimiser la gestion mémoire, l'espace virtuel du père n'est pas dupliqué
  - Si le fils accède en lecture à une donnée héritée du père, c'est la même donnée qui est accédée
  - Si le fils accède en écriture à une donnée héritée du père, alors celle-ci est recopiée dans l'espace virtuel du fils

Duplication de processus – fork()

- En cas de problème lors de la création du processus fils (impossibilité de création en général), la valeur retournée par fork() est -1.
- Le processus père et le processus fils sont concurrents :
  - Ils s'exécutent en parallèle
  - Le processus père et le processus fils peuvent se synchroniser par l'envoi de signaux :
    - le père connaît le pid du fils (valeur de retour de fork ())
    - Le fils peut connaitre le pid du père (fonction getppid())
- Une synchronisation particulière est réalisée à l'aide de la fonction système wait () :
  - Le père est mis en attente jusqu'à la terminaison du fils.
  - waitpid () permet d'attendre la terminaison d'un fils particulier

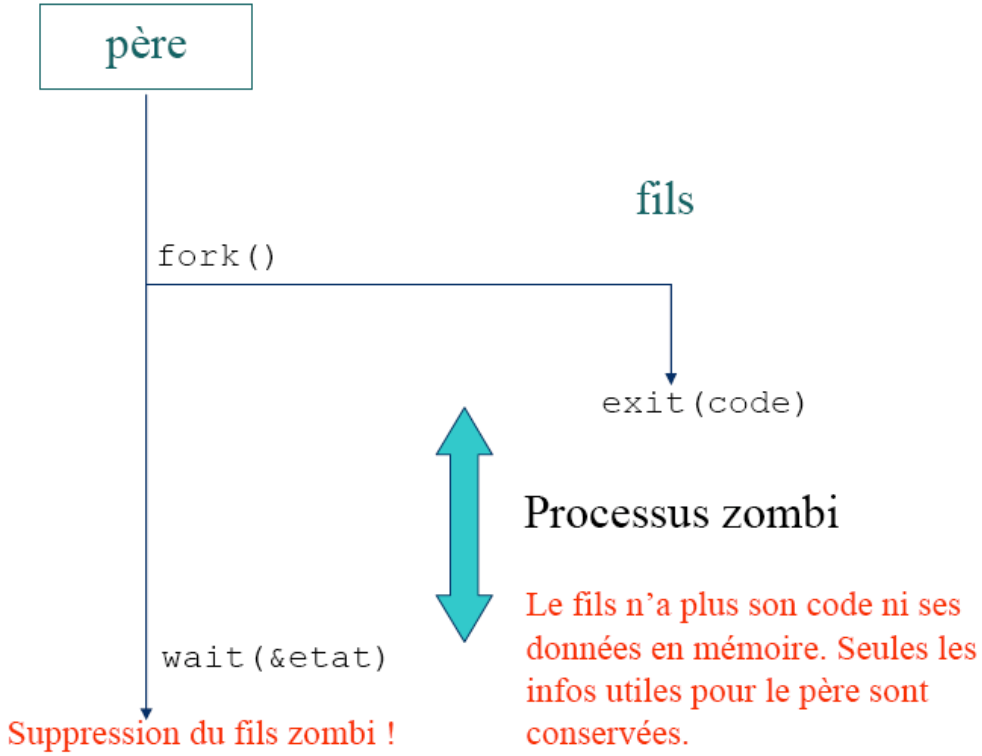
Identité de processus

- Tout processus a accès (par l'intermédiaire de fonctions système) :
  - À son pid (getpid())
  - Au pid de son père (getppid())



Les processus ‘zombis’

- Un processus qui meurt devient un zombi jusqu’au wait () de son père
- Si le père meurt avant le fils, il est adopté par un autre père (généralement init de pid 1)
- Sur certains SE, les zombis ne peuvent pas être supprimés !



Remplacement de processus – exec()

- La primitive système exec[l,v] () de remplacement permet de lancer l'exécution d'un nouveau code.
- Ainsi, il n'y a pas de création de nouveau processus. Dans le cas où le remplacement n'a pu se faire, la fonction exec () retourne -1.
- Le "nouveau" processus possède les mêmes caractéristiques (même contexte) que l'ancien :
  - pid,
  - père,
  - même priorité,
  - même propriétaire,
  - même répertoire de travail,
  - mêmes descripteurs de fichiers ouverts.

15

Communication entre processus

Tubes (pipe)

Tubes nommés

Fichiers couplés

Sockets

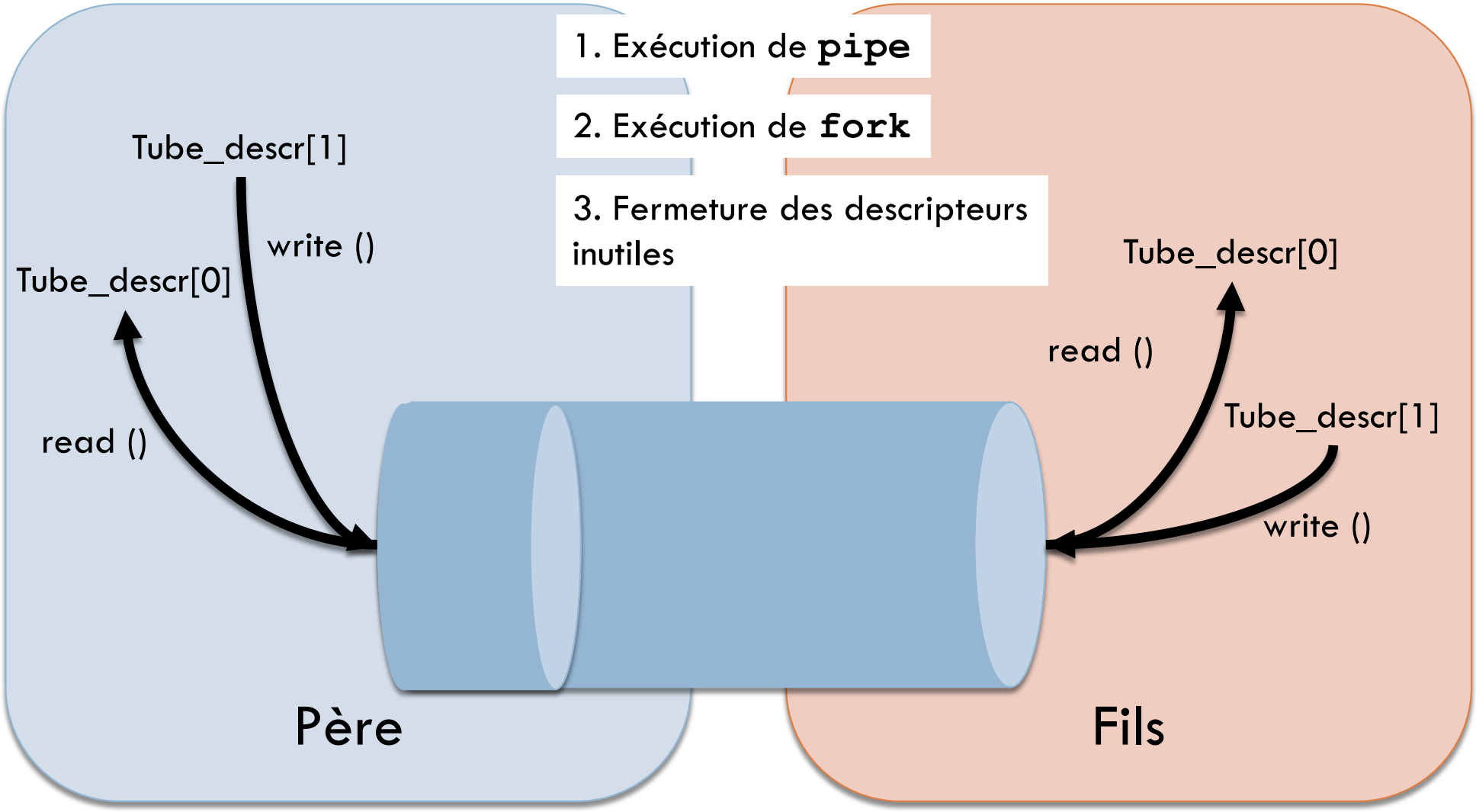
Mémoire partagée

File de messages

Communication entre processus Unix – pipe ()

- La fonction système pipe() crée un tube de communication pour permettre à des processus affiliés de communiquer entre eux
  - int pipe (int fd[2]);
- Cet appel système crée un “tuyau” de communication et renvoie dans le tableau fd un couple de descripteurs.
  - deux nouvelles entrées dans la table de descripteurs de fichiers ouverts sont initialisées (table des descripteurs du processus ayant réalisé cette ouverture de tube).
  - la valeur de retour d’un appel réeussi est 0 et -1 sinon.
  - Par convention le descripteur
    - fd[1] permettra à un premier (ou plusieurs) processus d’écrire à l’entrée du tube
    - fd[0] permettra à un autre processus (en général) de lire à la sortie

Communication entre processus – tubes / tubes nommés



Par convention : 0 pour lire et 1 pour écrire

Communication entre processus Unix – pipe ()

- Tous les processus “voulant” communiquer ainsi doivent “avoir accès” à ces descripteurs.
- Note : la table des descripteurs de fichiers ouverts d’un processus est dupliquée lors de la duplication d’un processus par fork () ou conservée lors de son remplacement par exec ().
- Remarque : la valeur de retour de pipe() est 0 si le tube a été créé et -1 autrement.
- Attention :
  - Une fin de fichier (caractère EOF) est envoyée dans un tube lorsque tous les processus ayant accès en écriture à ce tube (descripteur fd[1]) ont fermé ce descripteur.
  - La fonction read () retourne la valeur 0 à la lecture du caractère ‘EOF’.

Petit exercice de compréhension

- Un processus “père” ouvre un tube de communication pour permettre à ses deux fils, fils1 et fils2 créés par la suite, de communiquer.
- Le processus fils1 lit au clavier des caractères et n’envoie au processus fils2 que des caractères alphabétiques après les avoir capitalisés (filtre).
- Le processus fils2 lit ces caractères dans le tube jusqu’à ce qu’il n’y en ait plus à lire : caractère EOF.
- Le processus père attend que ses fils aient terminé de communiquer.

Alors, c'est compris ? (le père)


```
int p[2];
main () {
    int i, s;
    if (pipe(p) != 0) {fprintf (stderr, ''pb ouverture pipe\n''); exit (1); }
    if (fork () == 0) { fils1 (); }
    if (fork () == 0) { fils2 (); }
    close (p[0]); close (p[1]);
    fprintf (stderr, ''début attente\n'');
    i = wait (&s); i = wait (&s);
    fprintf (stderr, ''fin attente\n'');
    exit (0);
}
```

Alors, c'est compris ? (les fils)


```
fils1 () {
    char c;
    close (p[0]);
    fprintf (stderr, ''début fils 1\n'');
    fprintf (stderr, ''taper 0 pour
    terminer\n'');
    while ((read (0, &c, 1) && (c != '0')) {
            if (('a'<= c && (c<='z')) c-=32;
            write (p[1], &c, 1);
    }
    close (p[1]);
    fprintf (stderr, ''fin fils 1'');
    exit (0);
}
```

```
fils2 () {
    char c;
    close (p[1]);
    fprintf (stderr, ''début fils 2'');
    while ((read (p[0], &c, 1) > 0)
            write (1, &c, 1);
    close (p[0]);
    fprintf (stderr, ''fin fils 2'');
    exit (0);
}
```

Communication entre processus – dup ()

- La fonction système dup () permet de dupliquer un descripteur de fichier, en utilisant le plus petit numéro de descripteur disponible (première entrée libre dans la table de descripteurs de fichiers).
  - Utile pour réaliser des communications ‘double flux’ entre père et fils.

Communication en processus – mkfifo ()

- En complément des tubes qui permettent une communication entre processus ayant une filiation, il existe les 'tubes nommés' :
  - Ces tubes ont une entrée dans le système de fichier
  - Généralement les données restent en mémoire
  - Les primitives sont
    - mknod ()
    - mkfifo ()
    - unlink ()
  - Une fois le tube ouvert, on y accède par read / write
    - Si pas d'écrivain : la fin de fichier est atteinte et read () renvoie 0
    - Si au moins un écrivain :
      - Si le read () est bloquant (par défaut), le processus est réveillé lors d'une écriture
      - Si le read () n'est pas bloquant, si aucune donnée n'est présente read () retourne -1

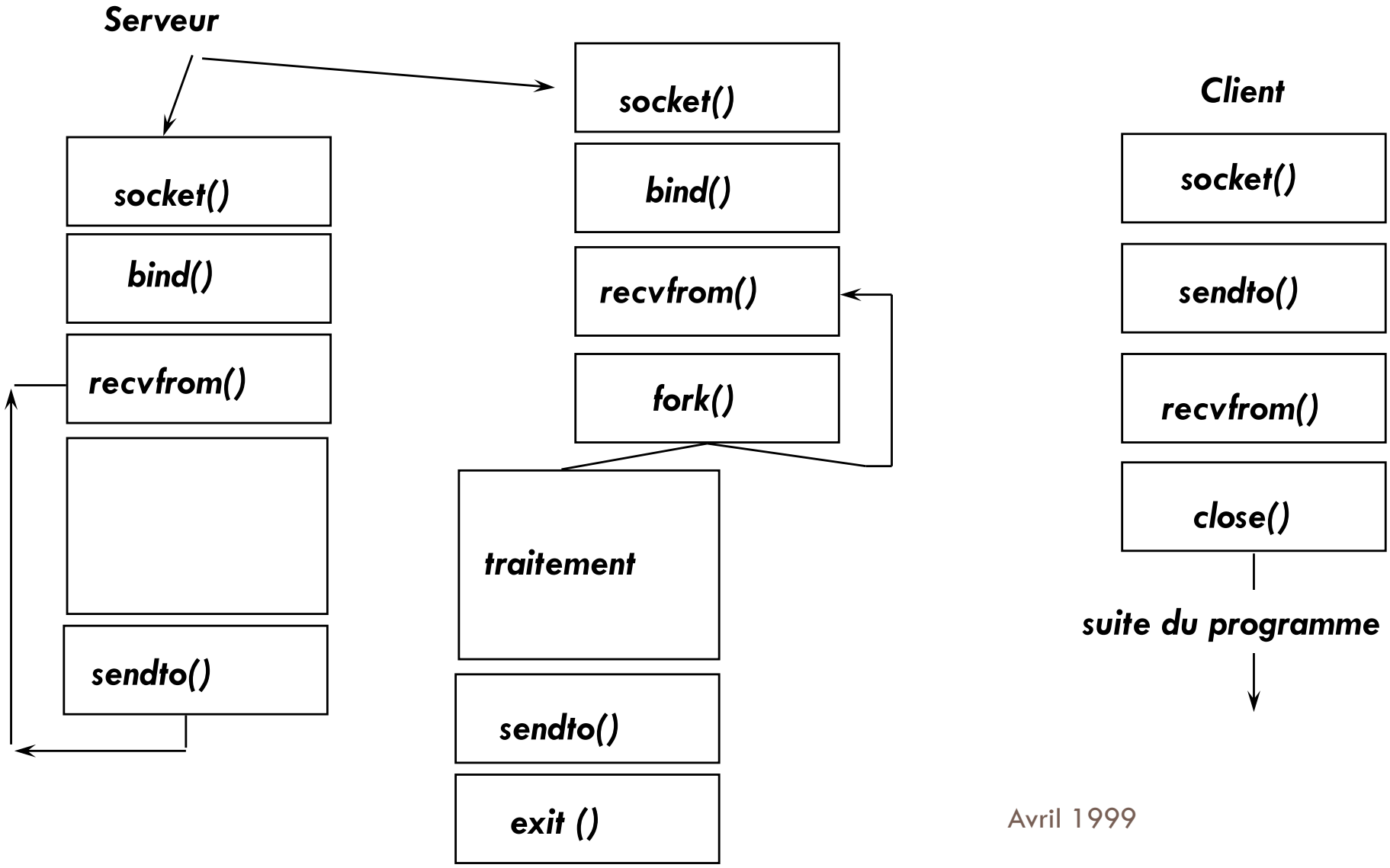
Communication entre processus – mmap ()

- L'appel système mmap () permet de projeter le contenu d'un fichier en mémoire
  - Le contenu de la mémoire est synchronisé automatiquement avec le contenu du fichier (et vice-versa).
    - L'option MAP_SHARED est requise pour garantir la synchronisation.
  - Un fichier couplé en mémoire (mappé) peut être partagé par plusieurs processus
    - Addr = mmap (NULL, size, PROT_READ | PROT_WRITE, MAP_SHARED, fr, 0)
    - NULL : le fichier est coupler à une adresse choisie par le système
    - size : taille de la zone mémoire
    - La zone mémoire peut être lue et écrite (autre protection possible : exécutée)
    - Les modifications sont partagées
    - Descripteur du fichier utilisé
    - Offset

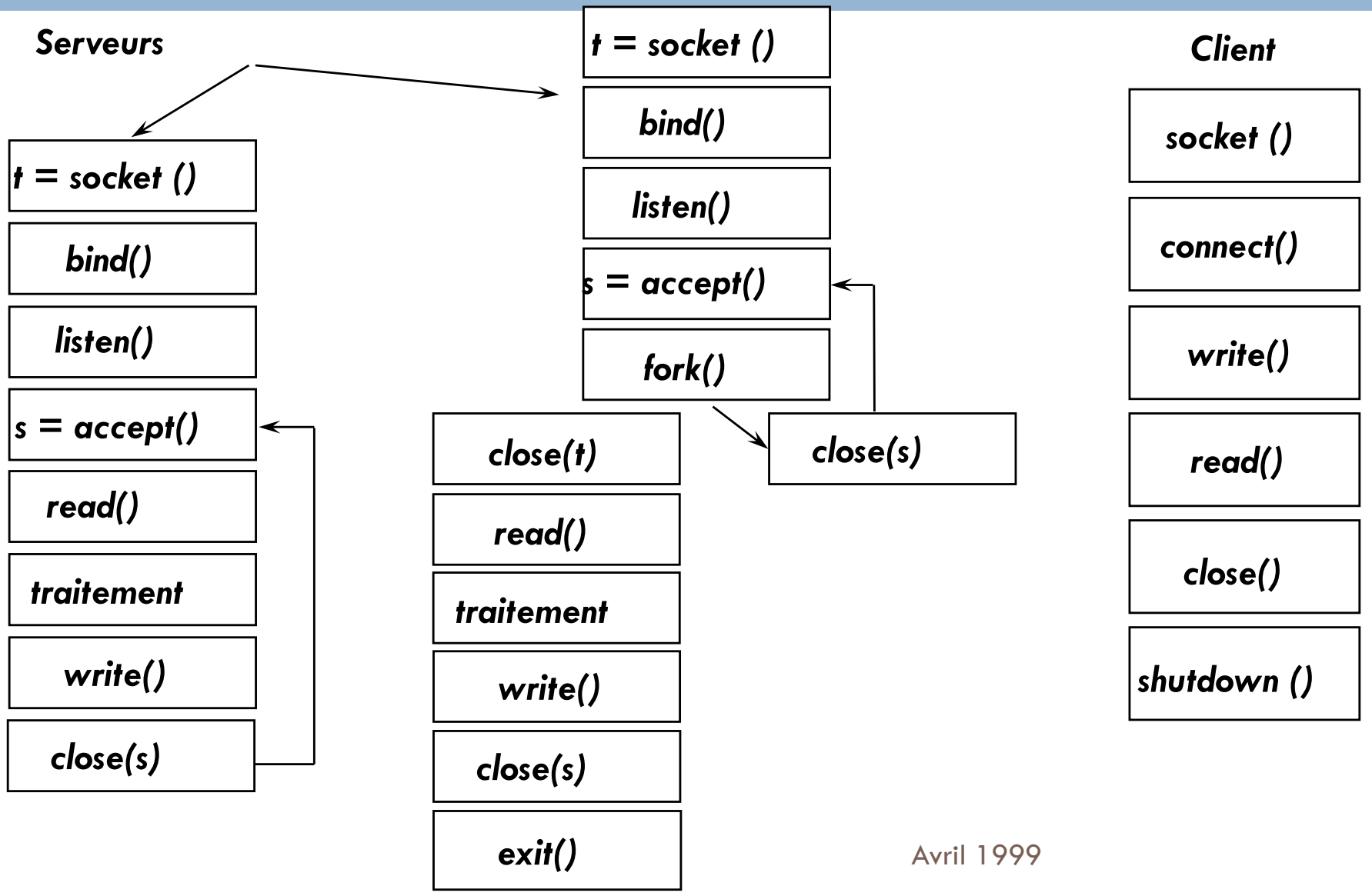
Communication entre processus – socket

- Architecture client-serveur
- Appels systèmes pour les sockets
  - Accept () pour se mettre en attente d'une connexion client
  - Connect () pour se connecter à un serveur
  - Send () / Write () pour envoyer un flux de données
    - Appel non bloquant dans que la capacité ne dépasse pas celle du buffer interne
  - Recv () / Read () pour recevoir un flux
    - Appel bloquant jusqu'à ce qu'il y ait quelque chose de disponible

Communication entre processus – socket
Socket en mode non connecté



Communication entre processus – socket
Socket en mode connecté



Communication entre processus – IPC

- Mécanismes IPC (Inter Process Communication) regroupent
  - Mémoire partagée
  - File de messages
  - Sémaphore
- Tous ces mécanismes survivent au processus qui l'a créés
  - Il faut donc les détruire explicitement
- Pour tous ces mécanismes
  - Les droits d'accès sont définis à la création
  - La commande ipcs liste les ressources IPC allouées
  - La commande ipcrm pemet de libérer des ressources

Communication entre processus – les clés IPC

- Tous les mécanismes IPC utilisent une clé permettant d'identifier la famille de processus pouvant utiliser le même segment mémoire partagé, le même sémaphore ou la même file de message
- Les procédures suivantes permettent de trouver une clé dynamiquement
  - key_t ftok (char *pathname, char project);
  - Exemple
    - cle =ftok ("/", 'A'); // clé absolue
    - cle =ftok (".", 'A'); // clé dépendant du répertoire du processus
- Autres solutions : définir statiquement la clé
  - Exemple
    - cle = 123

Communication entre processus – mémoire partagée

- Permet à plusieurs processus processus tout à fait quelconques (pas nécessairement affilié) de partager des segments en mémoire. Il s'agit d'un partage de mémoire qui n'induit aucune recopie de données ...
- Les segments mémoires peuvent être couplés à des adresses différentes
- Principales opérations
  - shmget () : création
  - shmat () : couplage du segment mémoire dans l'espace d'adressage virtuel du processus
  - shmctl () : modification des droits et du propriétaire, destruction
  - shmdt () : détachement du segment mémoire

Mémoire partagée – création (1ère étape)

- #include <sys/shm.h>
- Création d'une mémoire partagée
  - shmid = shmget (key t key, int size, int shmflg);
- shmget retourne l'identificateur (int shmid) du segment de mémoire partagée ayant la clé key.
- Un nouveau segment de taille size octets est créé si :
  - Indicateur de shmflg contient IPC_CREAT;
  - Par exemple IPC_CREAT|IPC_EXCL indiquent que le segment ne doit pas exister au préalable.
- Les bits de poids faible de shmflg indiquent les droits d'accès (rwxrwxrwx).
- Exemple :
  - id = shmget (cle, sizeof(int), IPC_CREAT|0666);

Mémoire partagée – liaison/couplage (2de étape)

- Lier un segment à un processus lui permet l'accès aux données contenues dans ce segment à l'aide d'un pointeur :
  - mem addr = shmat (int shmid, char *shmaddr, int shmflg);
- retourne l'adresse (char *mem addr) où le segment identifié par (shmid) a été placé en mémoire :
  - Placement automatique (et conseillé) si shmaddr = NULL
  - Si shmaddr != NULL, le segment est couplé à l'adresse indiqué (si c'est possible)
- shmflg spécifie quels sont les droits d'accès du processus au segment : SHM_R, SHM_W, ...
- Exemple :
  - addr = shmat (id, 0, 0);

Mémoire partagée – déliaison/découplage

- Délier un segment de mémoire partagée d'un processus
  - ret = shmdt (char *mem addr);
- Détache le segment du processus et retourne (-1) en cas d'erreur (0 sinon).
- Exemple :
  - If (shmdt (addr) == -1) {
    fprintf (stderr ,"segment indétachable\n"); exit(-1);
    }

Mémoire partagée - contrôle

- int shmctl (int shmid, int cmd, struct asmid ds *buf);

- permet diverses opérations
  - Destruction du segment (IPC_RMID)
    - A priori : le segment est détruit quand plus aucun processus ne le lie (ce n'est pas toujours le cas)
  - Verrouillage en mémoire (SHM_LOCK)
    - Le segment n'est plus swappé
- Exemples :
  - shmctl (shmid, SHM_LOCK, NULL); // verrouille mémoire partagées
  - shmctl (shmid, IPC_RMID, NULL); // détruit une mémoire partagée
  - If (shmctl (id, IPC_RMID, NULL)==-1) {
    fprintf (stderr, "segment indestructible\n"); exit(-1);
    }

Un exemple d'utilisation : producteur-consommateur

- Un processus producteur lit une valeur au clavier puis l'incrémente à la valeur d'une variable commune
  - Si la valeur lut au clavier est 1, le producteur s'arrête
- Un processus consommateur lit la valeur de la variable commune puis l'affiche
- Il n'y a pas de synchronisation…

Le producteur


```
void abandon(char message[]) { perror(message); exit(EXIT_FAILURE); }
int main(void) {
    key_t cle;
    int id; *seg_part; reponse;
    if (cle = ftok (getenv("HOME"), 'A') == -1) abandon("ftok");
    if (id = shmget (cle, sizeof(int), IPC_CREAT | IPC_EXCL | 0666) == -1)
            if (errno == EEXIST) abandon ("Note: le segment existe déjà\n")
            else abandon ("shmget");
    if (seg_part = (int *) shmat(id, NULL, SHM_R | SHM_W)  == NULL) abandon ("shmat");
    *seg_part = 0;
    while (scanf("%d", &reponse) != 1) *seg_part += reponse;
    if (shmdt ((char *) seg_part) == -1) abandon ("shmdt");
    if (shmctl (id, IPC_RMID, NULL) == -1) abandon ("shmctl(remove)");
    return EXIT_SUCCESS;
}
```

Le consommateur


```
void abandon(char message[]) { perror(message); exit(EXIT_FAILURE); }
int main(void) {
    key_t cle;
    int id, *commun;
    struct sigaction a;
    if (cle = ftok (getenv("HOME"), 'A') == -1) abandon("ftok");
    if (id = shmget (cle, sizeof(struct donnees), 0) == -1)
            if (errno == ENOENT) abandon ("pas de segment\n")
            else abandon ("shmget");
    if (commun = (int *) shmat (id, NULL, SHM_R) == NULL) abandon("shmat");
    while (*commun < 1000) { sleep(2); printf("sous-total %d\n", *commun;}
    if (shmdt((char *) commun) == -1) abandon("shmdt");
    return EXIT_SUCCESS;
}
```

Communication entre processus – file de message

- Fait parti des mécanismes IPC (Inter Process Communication)
- Implantation du concept de boîte aux lettres qui permet l'échange de messages structurés entre processus
- Fonctionnement très proche de celui de la mémoire partagée
  - msgget // création
  - msgrcv, msgsnd // réception, envoie
  - msgctl // contrôle - destruction

Synchronisation entre processus

Signaux

Verrous

Sémaphore

Files de messages

Synchronisation entre processus Unix – Les signaux

- cf. man -k signal ou man 7 signal
- Le traitement réalisé par un processus peut être interrompu par divers mécanismes d'interruptions.
- Par exemple, l'utilisateur peut "agir" sur le processus actif attaché au terminal
  - Emission des signaux à l'aide du clavier
    - CtrI-C (SIGINT)
    - CtrI-Z (SIGTSTP)
    - CtrI-\ (SIGQUIT).
- La liste des signaux disponibles sur le système peut être obtenue par la commande UNIX : kill -l

La liste des signaux

1) SIGHUP	2) SIGINT	3) SIGQUIT	4) SIGILL
5) SIGTRAP	6) SIGABRT	7) SIGBUS	8) SIGFPE
9) SIGKILL	10) SIGUSR1	11) SIGSEGV	12) SIGUSR2
13) SIGPIPE	14) SIGALRM	15) SIGTERM	16) SIGSTKFLT
17) SIGCHLD	18) SIGCONT	19) SIGSTOP	20) SIGTSTP
21) SIGTTIN	22) SIGTTOU	23) SIGURG	24) SIGXCPU
25) SIGXFSZ	26) SIGVTALRM	27) SIGPROF	28) SIGWINCH
29) SIGIO	30) SIGPWR	31) SIGSYS	34) SIGRTMIN
35) SIGRTMIN+1	36) SIGRTMIN+2	37) SIGRTMIN+3	38) SIGRTMIN+4
39) SIGRTMIN+5	40) SIGRTMIN+6	41) SIGRTMIN+7	42) SIGRTMIN+8
43) SIGRTMIN+9	44) SIGRTMIN+10	45) SIGRTMIN+11	46) SIGRTMIN+12
47) SIGRTMIN+13	48) SIGRTMIN+14	49) SIGRTMIN+15	50) SIGRTMAX-14
51) SIGRTMAX-13	52) SIGRTMAX-12	53) SIGRTMAX-11	54) SIGRTMAX-10
55) SIGRTMAX-9	56) SIGRTMAX-8	57) SIGRTMAX-7	58) SIGRTMAX-6
59) SIGRTMAX-5	60) SIGRTMAX-4	61) SIGRTMAX-3	62) SIGRTMAX-2
63) SIGRTMAX-1	64) SIGRTMAX		

Les signaux – principe de fonctionnement

- Lorsqu'un processus est chargé en mémoire, le système initialise sa table de traitement des signaux
- A chaque signal correspond un élément de la Table de Traitement des Signaux
- Par la suite, lorsque le processus recevra un signal, le traitement qu'il réalisait sera interrompu, et il exécutera la fonction associée au signal reçu.
- Pour la plupart des signaux, la fonction de traitement associée a pour effet de "terminer" le processus, excepté pour des signaux tels que SIGSTOP, SIGTSTP, SIGCONT.

Utilisation des signaux par programme

- La fonction système kill() permet à un processus d'envoyer un signal à un autre processus :
  - int kill (pid_t pid, int signum)
- La fonction système signal() permet à un processus de changer la fonction de traitement d'un signal :
  - typedef void (*sighandler_t) (int)
  - sighandler_t signal (int signum, sighandler_t handler)
- Ainsi, dans la table de traitement des signaux, la fonction associée au signal signum est remplacée par la fonction handler().
- Deux fonctions ont un rôle particulier :
  - SIG_IGN : permet d'ignorer un signal,
  - SIG_DFL : permet de repositionner la fonction de traitement d'un signal à la fonction par défaut.
  - signal(signum, SIG_IGN) ou signal(signum, SIG_DFL)

Utilisation des signaux par programme

- La fonction initiale de traitement de certains signaux (fonction par défaut) ne peut être modifiée ou ignorée : c'est notamment le cas des signaux SIGSTOP et SIGKILL
- ATTENTION : selon les signaux et les systèmes UNIX il est possible que lorsqu'un processus reçoit un signal, le système repositionne la fonction de traitement du signal par défaut ...

Petit exercice de compréhension - signaux

- Ecrivez un programme qui
  - compte le nombre de Ctrl-C (SIGINT)
  - ignore les Ctrl-Z (SIGSTP)
  - affiche “Au revoir” si l’utilisateur tape Ctrl-\ (SIGQUIT)

C’est compris ?


```
int cmpt = ‘1’;
void arret (int k) {
        println (‘’au revoir’’);
        signal (SIGQUIT, SIG_DFL);
        exit (0);
}
void interruption (int k) {
        signal (SIGINT, interruption);
        cmpt++;
}
main () {
        signal (SIGINT, interruption);
        signal (SIGQUIT, arret);
        signal (SIGTSTP, SIG_IGN);
        for (;;) {println (‘’cmpt %d’’, cmpt); sleep (1); }
}
```

Exercice de compréhension - signaux et fork

- Ecrire un programme qui se duplique
- Le fils envoie un signal (SIGUSR1) à son père toutes les secondes
- Le père compte les signaux (SIGUSR1) jusqu'à ce que l'utilisateur tape Ctrl-C

Avez-vous compris ?


```
int nb_reçu = 0;
void hand (int sig) {
    if (sig==SIGSUSR1) {signal (SIGSUSR1, hand); nb_recu++; printf (“.”); fflush (stdout); }
    else { printf (“reçu %d\n”, nb_recu); exit (0); }
}
main () {
    signal (SIGSURS1, hand), signal (SIGINT, hand);
    if (fork () == 0) {
            for (int i=0; i<10; i++) { kill (getppid(), SIGSUSR1); sleep (1); }
            printf(“fin du fils\nVous pouvez taper Ctrl-C\n”);
            exit (0);
    }
    while (1) pause ();
}
```

Synchronisation entre processus Unix – verrouillage de fichier

- La commande système flock () permet de mettre des verrous partagés ou exclusifs sur des fichiers
  - Verrou partagé (**LOCK_SH**) : autorise l'accès simultanée à plusieurs processus
  - Verrou exclusif (**LOCK_EX**) : un seul accès simultanée
  - Libération d'un verrou précédemment acquis (**LOCK_UN**)
  - On peut passer d'un verrou partagé à un verrou exclusif
  - Les verrous portent sur le fichier (pas son descripteur)
    - Si on duplique le descripteur, le verrou concerne toujours le même fichier
- Exemple d'utilisation

```
fp = fopen ("/tmp/lock.txt", "w+");
if (flock (fp, LOCK_EX)) {    // pose un verrou exclusif
    fwrite (fp, "Écrire dans un fichier\n");
    flock (fp, LOCK_UN);                    // libère le verrou
} else {  printf ( "Impossible de verrouiller le fichier !\n";  }
fclose (fp);
```

Synchronisation entre processus Unix – wait ()

- La primitive wait () provoque la suspension (mise en attente) du processus jusqu'à ce que l'un de ses processus fils se termine.
  - int wait (int *status)
  - La fonction wait () retourne le pid du fils qui s'est terminé (et qui a donc provoqué le réveil du père); dans le cas où il n'y a pas de fils, wait () retourne -1.
  - Le paramètre passé par adresse (int *status) permet d'obtenir des informations sur la façon dont s'est termiée le processus fils. Cette information de 16 bits doit être interprétée de la manière suivante :
    - si le processus se termine normalement par un exit (k), alors l'octet de poids faible est mis à 0 et l'octet de poids fort reçoit la valeur k,
    - si le processus se termine anormalement (signal), les deux octets permettent d'obtenir le numéero de ce signal (cf. man wait)...

Synchronisation entre processus - sémaphore

- Fait parti des mécanismes IPC (Inter Process Communication)
- Permet de résoudre le problème des accès concurrents à une même ressource telle que, par exemple, un segment de mémoire partagé entre plusieurs processus
- Les sémaphores IPC sont gérés sous forme d'un tableau, on effectue les opérations équivalente à P () et V() sur les éléments du tableau
  - Création du tableau : semget ()
  - Manipulation du tableau : semctl ()
  - Opération Down () et Up () : réservation ou libération de N unité de ressources

Utilisation des sémaphores Unix à la Dijsktra


```
typedef int semaphore;
void abandon(char message[]) { perror(message); exit(EXIT_FAILURE); }
semaphore creer_sem (key_t key, int val_init) {
    /* création d'un tableau de 1 sémaphore initialisé à val_init */
    semaphore sem;
    int r;
    if (sem = semget (key, 1, IPC_CREAT | 0666) < 0) abandon ("creer_sem");

    if (r = semctl (sem, 0, SETVAL, val_init) < 0)
            abandon ("initialisation sémaphore");
    return sem;
}
void detruire_sem(semaphore sem) { if (semctl (sem, 0, IPC_RMID, 0) != 0)
    abandon("detruire_sem"); }
```

Utilisation des sémaphores Unix à la Dijsktra


```
void changer_sem(semaphore sem, int val) {
    struct sembuf sb[1];
    sb[0].sem_num = 0;
    sb[0].sem_op = val;
    sb[0].sem_flg = 0;
    if (semop (sem, sb, 1) != 0) abandon("changer_sem");
}
void down(semaphore sem) { changer_sem(sem, -1); }
void up(semaphore sem) { changer_sem(sem, 1); }
```

Contrôle d'une section critique avec des sémaphores


```
// mutex
// → sémaphore
   initialisé à 1

down (mutex)
/* je suis
   * en
   * section
   * critique
   */
up (mutex)
```

```
semaphore sem;
key_t cle;

if (cle = ftok(getenv("HOME"), 'A')
                    == -1)
    abandon("ftok");
sem = creer_sem (cle, 1);
down (sem);
/* je suis en section critique */
up (sem);
detruire_sem (sem);
```

Mise en place d'une barrière avec des sémaphores

- Chaque processus i se bloque sur un sémaphore attendre initialisé à 0
- Le nombre de processus à attendre est N
- Pour protéger l'utilisation de la variable n initialisée à N qui compte le nombre de processus arrivée à la barrière, on utilise un sémaphore mutex

```
down(mutex)
n = n-1
Si (n > 0)
Alors
   up (mutex)
   down(attendre)
   down (mutex)
Finsi
n = n+1
Si (n < N)
   up(attendre)
Finsi
up(mutex)
```

Rendez-vous avec des sémaphores
var := expr


```
// variables partagées
écrit : Semaphore = 0;
lut : Semaphore = 0;
canal : aType;

// Processus 1 : (canal ? var)
P(écrit);
var := canal;
V(lut);

// Processus 2 : (canal ! expr)
canal := expr
V(écrit);
P(lut);
```

Rendez-vous avec des sémaphores
échange de valeur


```
// variables partagées
écrit : Semaphore = 0;
lut : Semaphore = 0;
canal1, canal2 : aType;
```

```
// Processus 1
canal2 := expr1;
V(écrit2);
P(écrit1);
var1 := canal1;
V(lut1);
P(lut2);
```

```
// Processus 2
canal1 := expr2
V(écrit1);
P(écrit2);
var2 := canal2;
V(lut2);
P(lut1);
```

Nous avons vu ou revu

- La vie des processus Unix
  - Clonage
  - Création par remplacement
- Communication entre processus Unix
  - Les tubes (pipes)
  - Les tubes nommés
  - Les fichiers couplé en mémoire
  - Les sockets
  - La mémoire partagée
  - Les files de messages
- Synchronisation entre processus UNIX
  - Les signaux
  - Les verrous
  - Les sémaphores

TD : ‘autour du parking’

Utilisation :

- Mémoire partagée
- Sémaphores