## Réseaux : Programmation multitâche :

### Multi processus et serveur TCP/UDP

Coté serveur

Créer une socket :

| int ds = socket(PF\_INET, SOCK\_DGRAM, 0); |
| --- |

Toujours tester les valeurs de retour et quitter si jamais elle n’est pas satisfaisante :

socket(famille, type, protocole);

* **famille** représente la famille de protocole utilisé (AF\_INET pour TCP/IP utilisant une adresse Internet sur 4 octets.
* **type** indique le type de service (orienté connexion ou non). Dans le cas d'un service orienté connexion (c'est généralement le cas), l'argument type doit prendre la valeur SOCK\_STREAM (communication par flot de données). Dans le cas contraire (protocole UDP) le paramètre type doit alors valoir SOCK\_DGRAM (utilisation de datagrammes, blocs de données)
* **protocole** permet de spécifier un protocole permettant de fournir le service désiré. Dans le cas de la suite TCP/IP il n'est pas utile, on le mettra ainsi toujours à 0

if (ds == -1){

perror("Serveur : pb creation socket :");

exit(1);

}

printf("ds = %i \n",ds);

Nommer la socket du serveur :

| struct sockaddr\_in sockServeur; sockServeur.sin\_family = AF\_INET; sockServeur.sin\_addr.s\_addr = INADDR\_ANY; sockServeur.sin\_port = htons((short)numPortServ); |
| --- |

| struct sockaddr\_in {  uint8\_t sin\_len; /\* longueur totale \*/  sa\_family\_t sin\_family; /\* famille : AF\_INET \*/  in\_port\_t sin\_port; /\* le numéro de port \*/  struct in\_addr sin\_addr; /\* l'adresse internet \*/  unsigned char sin\_zero[8]; /\* un champ de 8 zéros \*/ }; |
| --- |

3 variables :

**sin\_family :** indique la famille d'adresse Ip (toujours AF\_INET pour nous car on utilise les ipV4)

**sin\_addr :** structure de l'adresse Ip (INADDR\_ANY indique que la socket est associée à toutes les adresse Ip de la machine)

**sin\_port :** le port du Serveur

Liaison descripteur de socket et couple (on indique la taille du couple) :

Après création du socket, il s'agit de le lier à un point de communication défini par une adresse et un port, c'est le rôle de la fonction bind() : bind(int descripteur,sockaddr localaddr,int addrlen)

* **descripteur** représente le descripteur du socket nouvellement créé
* **localaddr** est une structure qui spécifie l'adresse locale à travers laquelle le programme doit communiquer

int nomSock = bind(ds, (struct sockaddr\*)&sockServeur, sizeof(sockServeur));

if (nomSock == -1){

perror("Serveur : pb creation liaison :");

exit(1);

}

printf("nomSock = %i \n",nomSock);

**Création de l’interface :**

| SOCKADDR\_IN sin = { 0 }; sin.sin\_addr.s\_addr = htonl(INADDR\_ANY); /\* nous sommes un serveur, nous acceptons n'importe quelle adresse \*/ sin.sin\_family = AF\_INET; sin.sin\_port = htons(PORT); if(bind (sock, (SOCKADDR \*) &sin, sizeof sin) == SOCKET\_ERROR) {  perror("bind()");  exit(errno); } |
| --- |

Recevoir un message :

char msg[100] ;

struct sockaddr\_in sockClient;

/\* \* On initialise la taille de la structure de la socket client

\* si on ne fait pas ça, on aura 0 dans la structure qui est censé contenir la taille de l'adresse source \*/

socklen\_t lgAdr = sizeof(sockClient);

int retourReceive = recvfrom(ds,msg,sizeof(msg),0, (struct sockaddr\*)&sockClient, &lgAdr );

if (retourReceive == -1) {

printf("La réception du message est en erreur \n");

}

printf("Le message reçu par le serveur du client: (%s:%i)",inet\_ntoa(sockClient.sin\_addr), ntohs((short) sockClient.sin\_port));

printf("est : '%s' de taille : %i \n", msg, retourReceive);

Envoyer un message retour au client :

| char \*msgServToClient; size\_t tailleMsg; int res; printf("Saisir le message a envoyer (max 100 char) :"); res = getline(&msgServToClient, &tailleMsg, stdin); if (res == ERROR){  perror("Erreur lors de l'envoi d'un message au serveur");  exit(3); }  /\* \* ici, on supprime le retour charriot \*/ int len = strlen(msgServToClient)-1; // Supprime le \n à la fin de la ligne d'input utilisateur if (msgServToClient[len] == '\n')  msgServToClient[len] = '\0'; }  int retourSendto = sendto(ds,msgServToClient,strlen(msgServToClient)+1,  0, (struct sockaddr\*)&sockClient, sizeof(sockClient)); printf("message = %s \n",msgServToClient); printf("retourSendto = %i \n",retourSendto); if (retourSendto == -1) {  // traitement de l'erreur  perror("L'envoi du message est en erreur \n");} |
| --- |

Fermer la socket :

| int close(int ds); int shutdown(int ds, int SHUT\_RDWR); printf("Serveur : je termine\n"); |
| --- |

**La fonction close()** permet la fermeture d'un socket.

/!\ **IMPORTANT** : cela permet au système d'envoyer les données restantes (pour TCP)

Ici "semble" inutile dans ce TP1 qui utilise UDP

Le close ne ferme pas la socket, elle est partagée par d'autres processus

* La fonction shutdown() permet la fermeture d'un socket dans un des deux sens (pour une connexion full-duplex)
* SHUT\_RD or 0 : ferme les receptions (READ)
* SHUT\_WR or 1 : ferme les envois (WRITE)
* SHUT\_RDWR or 2 : ferme les réceptions (READ) et les envois (WRITE)

Coté client :

**Répéter les premières étapes :**

Nommer la socket du serveur :

| struct sockaddr\_in sockServ; sockServ.sin\_family = AF\_INET ; sockServ.sin\_addr.s\_addr = inet\_addr(argv[1]); //argv[1] adr Ip du Serv sockServ.sin\_port = htons((short) numPortServ); // argv[2] numPortServ  printf("Serveur : désignation de la socket du serveur réussie : \n"). |
| --- |

**Pour envoyer, recevoir et fermer la socket étapes similaires au serveur .**

**La fonction listen()** permet de mettre un socket en attente de connexion. La fonction listen() ne s'utilise qu'en mode connecté (donc avec le protocole TCP) int listen(int socket,int backlog)

* **socket** représente le socket précédemment ouvert
* **backlog** représente le nombre maximal de connexions pouvant être mises en attente

| Sockaddr\_in Appelant; //structure destinée à recueillir les renseignements sur l'appelant Appelantlen = sizeof(from); accept(socket\_local,(struct sockaddr\*)&Appelant, &Appelantlen); |
| --- |

| if(listen(sock, 5) == SOCKET\_ERROR) {  perror("listen()");  exit(errno); } |
| --- |

**La fonction accept()** permet la connexion en acceptant un appel :

int accept(int socket,struct sockaddr \* addr,int \* addrlen)

* **socket** représente le socket précédemment ouvert (le socket local)
* **addr** représente un tampon destiné à stocker l'adresse de l'appelant
* **addrlen** représente la taille de l'adresse de l'appelant

**La fonction connect()** permet d'établir une connexion avec un serveur : int connect(int socket,struct sockaddr \* addr,int \* addrlen)

* **socket** représente le socket précédemment ouvert (le socket à utiliser)
* **addr** représente l'adresse de l'hôte à contacter. Pour établir une connexion, le client ne nécessite pas de faire un bind()
* **addrlen** représente la taille de l'adresse de l'hôte à contacter

**La fonction recv()** permet de lire dans un socket en mode connecté (TCP) :

int recv(int socket,char \* buffer,int len,int flags)

* **socket** représente le socket précédemment ouvert
* **buffer** représente un tampon qui recevra les octets en provenance du client
* **len** indique le nombre d'octets à lire
* **flags** correspond au type de lecture à adopter

**La fonction send()** permet d'écrire dans un socket (envoyer des données) en mode connecté (TCP) : int send(int socket,char \* buffer,int len,int flags)

* **socket** représente le socket précédemment ouvert
* **buffer** représente un tampon contenant les octets à envoyer au client
* **len** indique le nombre d'octets à envoyer
* **flags** correspond au type d'envoi à adopter

**Les fonctions close() et shutdown()**

La fonction close() permet la fermeture d'un socket en permettant au système

d'envoyer les données restantes (pour TCP) :

* int close(int socket)

La fonction shutdown() permet la fermeture d'un socket dans un des deux sens

(pour une connexion full-duplex) :

* int shutdown(int socket,int how)
* Si how est égal à 0, le socket est fermé en réception
* Si how est égal à 1, le socket est fermé en émission
* Si how est égal à 2, le socket est fermé dans les deux sens

**La fonction getsockname() :** renvoie le nom name de la socket indiquée. Le paramètre namelen doit être initialisé pour indiquer la taille de la zone mémoire pointée par name. En retour, il contiendra la taille effective (en octets) du nom renvoyé. Cet appel système renvoie 0 s'il réussit, ou -1 s'il échoue, auquel cas errno est renseignée en conséquence.

**Cours :**

La couche transport peut découper un message en unités plus petites (segments) pour les passer à la couche réseau. La couche réseau en forme des paquets, etc. Les paquets peuvent arriver dans le désordre (en fonction du routage). La couche transport effectue le ré-assemblage d’un message à la réception des morceaux et éventuellement la remise dans l’ordre.

Pour communiquer il faut au moins pouvoir identifier une machine dans le réseau : elle aura une adresse IP (Internet Protocol).

**Définition d’un serveur :**

processus qui attend des requêtes provenant de processus clients, réalise ces requêtes et rend (ou pas) les résultats.

**Définition Client :**

processus qui envoie des requêtes au processus dit serveur, attend une réponse (ou pas).

**Définition Requête / réponse :**

suite d’instructions, commandes, simple chaı̂ne de caractères, etc. obéissant à un langage, un accord ou une structure préalables connus des deux entités communicantes (protocole d’application).

Une socket est définie par un domaine d’appartenance, une paire(adresse IP et port) un type et un protocole. Une socket est identifiée **par un descripteur de fichier.**

Une socket est associée **à deux buffers, un de réception** qui contient les données reçues par la couche transport et à lire par la couche application et **un d’émission** qui contient les données que la couche application transmet à la couche transport.

• IPv4 (PF INET) : L’adresse de la socket est une adresse IPv4 et la communication se fait avec un ou des processus distants suivant les protocoles Internet v4.

• IPv6 (PF INET6) : L’adresse de la socket est une adresse IPv6 et la communication se fait avec un ou des processus distants suivant les protocoles Internet v6.

L’association d’une adresse à une socket est appelée : **nommage**.

Deux méthodes de nommage :

• Le programmeur ou l’utilisateur choisi une adresse IP et un numéro de

port. Dans ce cas, le système vérifie leur disponibilité.

• Laisser le système choisir une ou des adresses. Il est possible ensuite de

consulter ces informations pour les connaître.

Les numéros jusqu’à **1024** sont officiellement réservés pour des applications connues et ne peuvent être demandés par une application d’utilisateur non administrateur.

**INADDR ANY** : Attache la socket à toutes les interfaces réseaux locales.

Getsockname (...) : obtenir le numéro de port alloué.

Il est possible de ne pas faire le nommage côté client.

**Connaı̂tre l’adresse de la socket du destinataire.**

• passée en paramètre, saisie, etc.

• autrement : obtenue lors de la réception d’un message (voir plus loin)

**Recevoir un message serveur :** Espace mémoire alloué (statiquement ou dynamiquement) pour stocker le message reçu et l’adresse de l’expéditeur.

**Les types :**

• L’IP et le numéro de port sont stockés au format réseau (network byte order) : XXxx

• Les entiers sont au format hôte (host byte order) : xxXX ou XXxx

• Une conversion est nécessaire : fonctions ntohs(), htons(), ntohl(), htonl()

**Valeurs de retour et gestion des erreurs :**

Création de socket avec : int dSock = **socket**(PF INET, SOCK DGRAM, 0) : (domaine, type protocole) : Le descripteur de la socket créée ou -1 en cas d’erreur.

Nommage de la socket avec : int **bind** (int descripteur, const struct sockaddr \*adresse, socklen t lgAdr) : (descripteur de socket, pointeur vers l’adresse, long de l’adresse) :

0 si le nommage a réussi, -1 sinon (e.g. si le port est déjà utilisé).

Envoi d’un message : ssize t **sendto** (int descripteur, const void \*msg, size t lg, int flags, const struct sockaddr \*adrDest, socklen t lgAdr) : (descripteur, pointeur vers premier octet du message, nbr d’octets du message, option d’envoi 0 par défaut, pointeur vers l’adresse de la socket, longeur de l’adresse) :

Si l’appel réussit, le nombre d’octets effectivement déposés dans le buffer d’envoi de la socket, -1 sinon (avec errno positionné).

Réception d’un message : ssize t **recvfrom** (int descripteur, const void \*msg, size t lg, int flags, const struct sockaddr \*adrExp, socklen t \* lgAdr) : Si l’appel réussit, le nombre d’octets effectivement extraits depuis le buffer de réception de la socket, -1 sinon (avec errno positionné).

Fermer une socket : int **close** (int descripteur) ; int shutdown(int descripteur, int comment) ;

0 si la fermeture a réussi, -1 sinon.

Passer une socket en mode écoute : int **listen**(int descripteur, int nbMaxEnAttente) :

0 en cas de succès, -1 sinon (avec errno positionnée).

Accepter une demande de connexion : int **accept**(int descr, struct sockaddr \*adrClient, socklen\_t lgAdr) : Si demande acceptée, le descripteur (> 0) de la socket créée, -1 sinon (avec errno positionnée).

Recevoir un message TCP : ssize\_t **recv**(int descripteur, const void \*msg, size\_t lg, int flags)

Si l’appel réussit, le nombre d’octets effectivement extraits (> 0 et ≤ lg), 0 si la socket à été fermée, -1 sinon (avec errno positionnée).

Envoi d’un message TCP : ssize\_t **send** (int descripteur, const void \*msg, size t lg, int flags)

Si l’appel réussit, le nombre d’octets (> 0 et ≤ lg) effectivement déposés dans le buffer associé à la socket, 0 si la socket a été fermée, -1 sinon (avec errno positionnée).

• Datagramme (SOCK DGRAM) : un message est expédié en un seul paquet bien délimité et est reçu entièrement en une seule fois (un seul morceau).

• Stream (SOCK STREAM) : un message est considéré par la couche transport comme un flot continu d’octets. Si l’expéditeur envoie plusieurs messages, le destinataire pourra lire ces messages en une ou plusieurs fois. Les limites des messages ne sont pas préservées par la couche transport.

Mode de communication :

• Non connecté : A chaque émission, une adresse de destination doit être spécifiée (analogie : communications par courrier postal). La destination d’un message à émettre via une socket en mode non connectée n’est pas nécessairement la même que celle du message suivant. C’est ce que nous avons appliqué précédemment.

• Connecté : la transmission de messages est précédée par une phase de connexion avec une autre socket. Une socket en mode connecté est donc utilisée pour communiquer de façon exclusive avec une seule autre socket. Nous parlons de circuit ou canal virtuel établi entre les deux sockets (analogie : communications téléphoniques).

**TCP (Transmission Control Protocol)**

• Une socket de type stream (SOCK STREAM) utilise le protocole

* fiable
* ordre garanti
* duplication impossible
* mode connecté

**UDP (User Datagram Protocol)**

• Une socket de type datagramme (SOCK DGRAM) utilise le protocole

* non fiable
* ordre non garanti
* mode non connecté

• Le principe d’utilisation du protocole UDP a été illustré avec un client et un serveur.

• Ce principe est extensible à plusieurs clients et/ou plusieurs serveurs.

• Avant d’envoyer un message vers une socket distante, veiller à ce que le nommage de cette socket soit déjà fait.

**Pré Requis pour recevoir un message** : Une socket correctement créée et connectée à une socket distante. Un espace mémoire alloué (statiquement ou dynamiquement) pour stocker le message reçu.

**Pré Requis pour envoyer un message** : Une socket correctement, créée et connectée ou en attente de connexion. Le message à envoyer correctement construit.

Un message en TCP est vu comme un flux d’octets et peut être reçu

et envoyée en une ou plusieurs fois.

si un appel à send(...) ou recv(...) retourne un nombre d’octets

inférieur au nombre attendu, il ne s’agit pas d’une erreur ! Ce cas est à

prendre en compte à chaque utilisation de ces fonctions.

Il n’est pas nécessaire d'être en réception pour recevoir un message.

* En TCP, il est aussi possible d’utiliser les fonctions read(...) et write(...) pour recevoir et envoyer des messages, ainsi que sendto(...), recvfrom(...).
* Les appels des fonctions : connect(...), accept(...), send(...), recv(...), read(...), write(...), sendto(...) et recvfrom(...) peuvent être bloquants .

**• Deux types de serveurs :**

* **itératif** : traite un client à la fois et l’un(e) après l’autre. Il y a aussi la possibilité de traiter plusieurs clients en même temps mais une requête après l’autre via le multiplexage des entrées/sorties
* **concurrent** : traite plusieurs clients (TCP) requêtes en parallèle (simultanément).

**Serveur concurrent multiprocesseur :** déléguer le traitement d’un client à un thread est aussi une solution.

**Le serveur concurrent peut gérer plusieurs connexions en parallèle :**

| s = socket (...); bind (...); listen (s, ...); for (;;) {  ns = accept (s, ...);  if (fork() == 0) {  close (s);  executer\_service (ns, ...);  close (ns);}  else {close (ns);} } |
| --- |

<https://broux.developpez.com/articles/c/sockets/>

<https://linux.die.net/man/3/inet_ntoa>

**htons()** host to network short

**htonl()** host to network long

**ntohs()** network to host short

**ntohl()** network to host long

La fonction **inet\_addr()** convertit l'adresse Internet de l'hôte cp depuis la notation numérique pointée IPV4 en une donnée binaire dans l'ordre des octets du réseau. Si l'adresse est invalide, INADDR\_NONE (généralement -1) est renvoyé. L'utilisation de cette fonction pose problème car -1 est une adresse valide (255.255.255.255). Évitez-la au profit de inet\_aton(), inet\_pton(3) ou getaddrinfo(3) qui fournissent un renvoi d'erreur plus propre.

La fonction **inet\_network()** convertit cp, une chaîne en notation numérique pointée IPV4 en un nombre dans l'ordre des octets de l'hôte pertinent pour être utilisé comme adresse de réseau Internet. a string in IPv4 numbers-and-dots notation, Si elle réussit, cette fonction renvoie l'adresse convertie. Si l'adresse est invalide, -1 est renvoyé.

La fonction **inet\_ntoa()** convertit l'adresse Internet de l'hôte in donne dans l'ordre des octets du réseau en une chaîne de caractères dans la notation numérique pointée. La chaîne est renvoyée dans un tampon alloué statiquement, qui est donc écrasé à chaque appel.

La fonction **inet\_lnaof()** renvoie la partie adresse locale extraite d'une adresse Internet in. La valeur est renvoyée dans l'ordre des octets de l'hôte.

La fonction **inet\_netof()** renvoie la partie adresse réseau extraite d'une adresse Internet in. La valeur est renvoyée dans l'ordre des octets de l'hôte.

La fonction **inet\_makeaddr()** est la réciproque des fonctions **inet\_netof()** et **inet\_lnaof()**. Elle renvoie une adresse d'hôte Internet dans l'ordre des octets du réseau, en combinant le numéro du réseau net et l'adresse locale host, chacun d'eux étant donné dans l'ordre des octets de l'hôte.

La structure in\_addr utilisée dans **inet\_ntoa(), inet\_makeaddr(), inet\_lnaof() et inet\_netof()** est définie ainsi dans <netinet/in.h> :

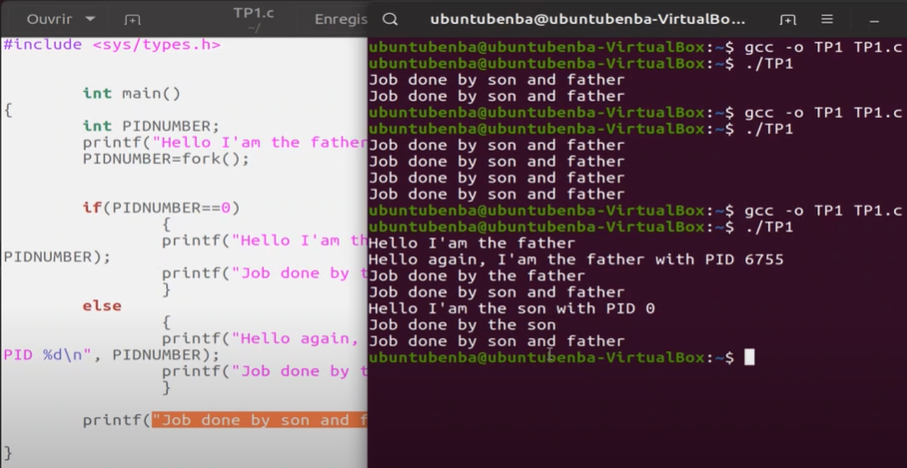
**Fork :**

int pidfils;

pidfils = fork();

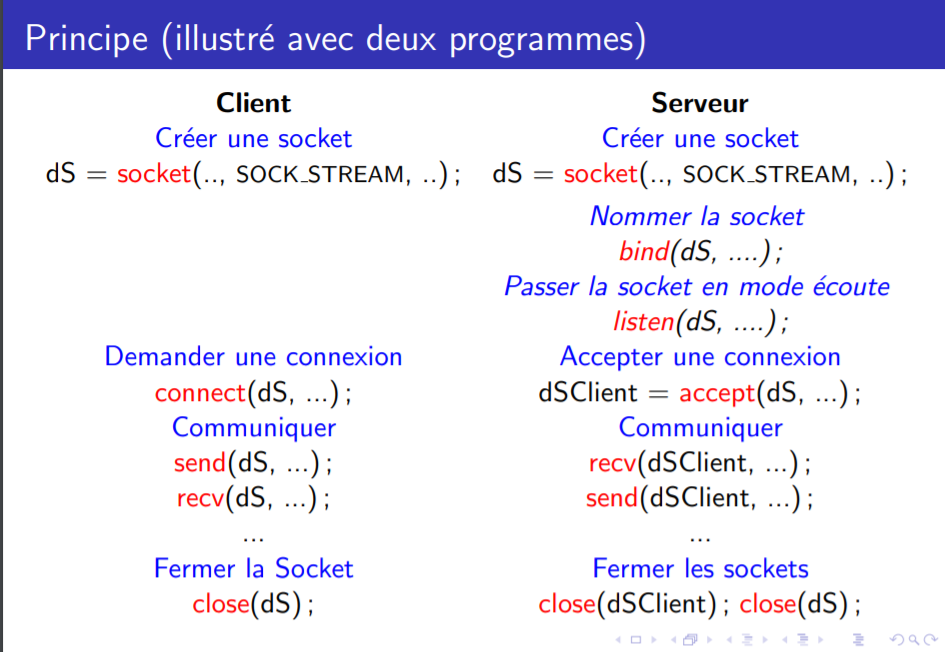
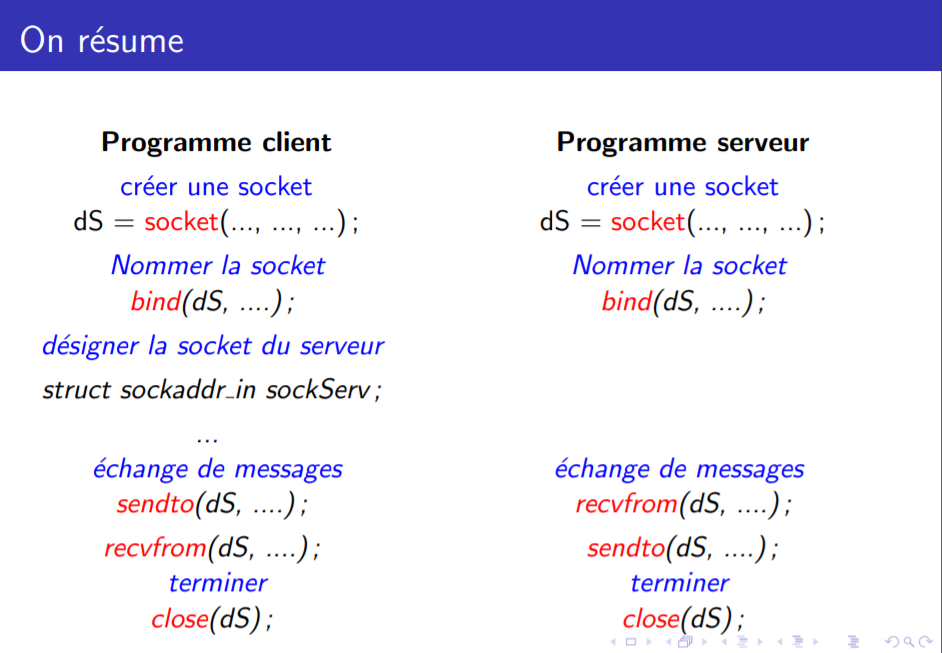
si pidfils == 0 alors on est dans le fils sinon on est dans le père.

Le fork créer le fils et garde le père donc si on fait un if else, on aura affichage des instructions pour le père et affichage des instructions du fils.



| **Questions Wooclap :**  Le nommage de la socket du client est-il nécessaire ?   * Non, cette information n'est pas nécessaire au bon fonctionnement du système client/serveur.   Le client envoyait une chaîne de caractères, et maintenant il envoie un tableau d'entiers. - - Que faut-il modifier côté serveur pour que l'application fonctionne ?   * Le type du buffer passé à la fonction recvfrom.   Que se passe-t-il si le client est lancé avant le serveur ?   * Le client s'exécute jusqu'à l'envoi du premier message. * Selon le protocole utilisé (TCP/UDP), l'exécution du client est différente.   Que se passe-t-il si le client envoie un message juste avant le nommage côté serveur ?   * Le client s'exécute mais la socket du serveur n'existe pas (n'est pas atteignable).   Que se passe-t-il si le client envoie un message après le nommage côté serveur mais avant l'appel à recvfrom(...) par ce serveur ?   * Le serveur reçoit le message sur sa socket.   Que se passe-t-il si le serveur attend un message de taille plus petite que celle du message envoyé par le client ?   * Le serveur reçoit uniquement les données comprises dans la taille spécifiée.   Que se passe-t-il si vous lancez deux serveurs avec le même numéro de port ?   * Le deuxième serveur renvoie une erreur au moment du bind. Le système d'exploitation ne permet pas d'associer deux sockets à un même port.   Quelles assertions à propos de la fonction bind sont correctes parmi les suivantes ?   * L'appel bind(ds, (struct sockaddr \*)&sock, lgSock) permet de nommer le descripteur ds avec les informations contenues dans la structure sock. * L'appel bind(ds, (struct sockaddr \*)&sock, lgSock) renvoie une erreur si le port spécifié dans sock est déjà attribué. * L'appel bind(ds, (struct sockaddr \*)&sock, lgSock) renvoie une erreur si ds ne correspond pas à un descripteur de socket.   Que se passe-t-il si un programme client envoie un message juste avant le nommage de la socket côté serveur ?   * Le client s'exécute mais la socket du serveur n'existe pas (n'est pas atteignable).   Quels appels sont valides pour qu'un programme client puisse recevoir un message d'un programme serveur ? Le message sera reçu dans buf de taille len depuis le descripteur de socket sockfd.   * recvfrom(sockfd, buf, len, flags, NULL, NULL) * recv(sockfd, buf, len, flags); * recvfrom(sockfd, buf, len, flags, &sockAddr, &lgSockAddr)   Côté serveur, la fonction accept(ds, (struct sockaddr\*)&sockAddr, &lgAdr) retourne   * Un descripteur de fichier qui permettra au serveur de communiquer avec un client.   Quelles assertions sont vraies à propos de la fonction connect(ds, (struct sockaddr\*)&sockAddr, lgAdr) ?   * Elle retourne 0 en cas de succès. * Elle est bloquante si la liste d'écoute du serveur est saturée.   Quelles assertions à propos des fonctions send/recv sont correctes ?   * Elles renvoient le nombre d'octets envoyés/lus en cas de succès. * Un message envoyé/reçu via send/recv peut être tronqué lors de son transit sur un réseau.   Comment un serveur peut s'assurer d'avoir reçu l'intégralité d'un message provenant d'un client ? Quel protocole d'échange doit être mis en place ?   * Protocole 1: Le client envoie au serveur la taille du message, puis le message. |
| --- |

#### UDP : TCP :



La fonction bzero() met à 0 (octets contenant « \0 ») les n premiers octets du bloc pointé par s.

### Réseaux : Multi threads :

Idée : permettre l'exécution parallèle de plusieurs traitements.

Un processus constitue une seule unité d'exécution qui s'exécute séquentiellement sur un seul processeur. Le moyen de faire du parallélisme consiste à créer plusieurs processus.

Un thread est un fil d'exécution. Ils permettent de dérouler plusieurs suites d’instructions en parallèle, à l'intérieur d’un même processus.

* Chaque thread a sa propre pile et des variables locales.
* Un thread peut partager des données en mémoire avec d’autres threads.
* La fonction s'exécute de façon asynchrone.
* Dans un processus nous avons un thread principal exécutant la fonction main, et des threads secondaires.
* Sur une machine multiprocesseur, chaque thread peut s’exécuter sur un processeur, indépendamment d’un autre.
* Le système gère la commutation de contexte entre threads.

Rappel : deux fonctions, lImpair() et lPair(), pouvant s'exécuter en parallèle. Un schéma possible de traitement de l’image avec des threads :

| int main(){ ... // d´efinition de la matrice (image) ... pthread create(lImpair,...); pthread create(lpair,...); ... } |
| --- |

#### **Prototypes Multi-Thread:**

Création :

| int pthread\_create( // résultat 0 si réussite, 6= 0 sinon pthread t \* idThread, // identité obtenue en résultat pthread attr t \*attributs, // NULL pour commencer void \* (\*fonction)(void \*), // fonction démarrer  void \* param) ; // paramètre(s) à passer à la fonction |
| --- |

Cette fonction démarre l’exécution d’un nouveau thread, en **parallèle avec celui qui l’a appelé**, (thread principal ou secondaire), mais dans le même processus. Le dernier argument permet de passer un paramètre à la fonction, ou plusieurs regroupés dans une structure.

Abandon / fin :

| void pthread\_exit(void \* retour); |
| --- |

Le paramètre retour est un résultat (valeur de retour), pouvant être consulté par un autre thread du même processus, attendant la fin de celui qui vient d’abandonner.

Identification :

| pthread\_t pthread\_self(void); |
| --- |

renvoie l'identité du thread appelant.

Egalité :

| int pthread\_equal(pthread t idT1, pthread t idT2); |
| --- |

Renvoie une valeur non nulle si réussite, 0 si échec.

Un thread quelconque, principal ou secondaire, peut attendre la fin

d’un autre qu’il connaît :

| int pthread\_join(pthread t idT, void \*\*retourCible); |
| --- |

Elle permet au thread appelant d’attendre la fin de celui issu du même processus, identifié par idT. Résultat 0 si réussite != 0 sinon. L’appelant est bloqué si le thread idT n’est pas terminé. Il sera débloqué lorsque le thread idT aura fait pthread exit().

Création variable conditionnelle :

| **int pthread\_cond\_init ( // résultat 0 si réussite, != 0 sinon pthread\_cond\_t \*cond , // variable conditionnelle à créer pthread\_condattr\_t \*attr) ; // NULL par défaut** |
| --- |

Permet de créer une variable conditionnelle et de l’initialiser.

Note : pour des raisons de portabilité, utiliser l’initialisation par défaut.

Une initialisation plus simple peut être effectuée par la déclaration :

| pthread\_cond\_t uneCond = PHTREAD COND INITIALIZER; |
| --- |

Prototype destruction

| int pthread cond destroy(pthread cond \*cond); |
| --- |

Détruit la variable conditionnelle pointée.

Attente non bornée :

| int pthread\_cond\_wait ( // résultat 0 si réussite, != 0 sinon pthread\_cond\_t \*vcond, // variable conditionnelle associée´ // a l'événement attendu pthread\_mutex\_t \*verrou) ; // verrou d'accès à la donnée commune |
| --- |

Cette primitive **réalise un appel bloquant**, qui de façon atomique déverrouille le verrou et attend que la condition vcond soit annoncée, forcément par un autre thread (voir ci-après pour l’annonce). Note : La primitive suppose que le thread appelant a obtenu précédemment le verrouillage de verrou. Attention : plusieurs threads peuvent être débloqués. Il est donc utile de tester à nouveau le prédicat après réveil (à la sortie de l’attente !)

Réveiller un thread :

| int pthread cond signal(pthread cond t \*cond); |
| --- |

Provoque le réveil d’un seul thread.

| int pthread cond broadcast(pthread cond t \*cond); |
| --- |

provoque le réveil de tous les threads attendant la variable conditionnelle cond. Important : lors du réveil, les threads réveillés vont obtenir tour à tour automatiquement le verrouillage du verrou.

#### **Synchronisation :**

Trois formes :

* **Attendre la fin** d'exécution d’un thread
* **Exclusion mutuelle**, via la notion de verrous binaires (à deux états, verrouillé ou non).
* **Attendre l'occurrence d’un événement** (pour importer un ordre par exemple), via la notion de variables conditionnelles.

**Attendre la fin d'exécution :**

Comme vu plus haut un thread quelconque, principal ou secondaire, peut attendre la fin d’un autre qu’il connaît.

**L’exclusion mutuelle :**

* Un verrou est un sémaphore ayant deux états possibles : Libre ou occupé (verrouillé).
* Un thread **peut demander de le verrouiller**. Il ne peut obtenir ce verrouillage que si le **verrou est libre et un seul thread peut obtenir le verrouillage**.
* Lorsqu’un thread a **obtenu le verrouillage**, un autre thread qui demande le **verrouillage du même verrou sera bloqué**.
* Seul le thread qui a obtenu le verrouillage peut déverrouiller un verrou.
* Le verrouillage et déverrouillage sont des opérations atomiques.
* Un **verrou est appelé mutex**. Il est de **type pthread\_mutex\_t.**

**Fonctions :**

| pthread mutex init(...) verrou crée etat libre pthread mutex lock(...) verrouillage pthread mutex trylock(...) //verrouillage si etat libre sinon, erreur sans blocage pthread mutex unlock(...) //déverrouillage etat libre pthread mutex destroy(...) //destruction |
| --- |

Autre façon d’utiliser un mutex :

| pthread\_mutex\_t verrou = PHTREAD MUTEX INITIALIZER; |
| --- |

* Tout accès à une variable accessible en lecture par un thread et en écriture par un autre doit ​être protégé.
* **Un seul mutex peut protéger plusieurs variables, mais pas​ l'inverse.**
* Les opérations ...lock() et ...unlock() sont atomiques, mais pas la portion de code qui se trouve entre les deux.
* Cette portion de code est appelée section critique
* **Si un thread est dans une section critique, il doit être garanti qu’aucun autre thread n’y soit simultanément**.

**Remarque** : plusieurs threads peuvent attendre la satisfaction d’un même prédicat et/ou le rendre insatisfait.

* Un thread ayant réussi le verrouillage d’un verrou, pourra le relâcher et passer à l'état bloqué pour attendre un évènement, ceci de façon atomique. Il demandera aussi à être réveillé en retrouvant l'état verrouillé.
* Le réveil sera réalisé par un autre thread. Ce dernier peut activer un ou plusieurs threads en attente par l'intermédiaire d’une variable conditionnelle.
* Si plusieurs threads sont réveillés, un seul obtiendra le verrou, les autres ne le trouveront que lorsqu’il sera déverrouillé.

**Une variable conditionnelle est une donnée commune à plusieurs threads symbolisant l'occurrence d’un événement.** Cet **événement annonce la satisfaction d’un ou de plusieurs prédicats**. Lorsque vous souhaitez mettre un thread en veille, une variable de condition peut être utilisée. En C sous Linux, il existe une fonction pthread\_cond\_wait() pour attendre ou dormir. D’autre part, il existe une fonction pthread\_cond\_signal() pour réveiller un thread en sommeil ou en attente. Les threads peuvent attendre une variable de condition.

**Fonction :**

| pthread\_cond\_init(&c) //crée la variable conditionnelle c. pthread\_cond\_wait(&c,&v) //bloque l'appelant et rend le verrou de façon atomique pthread\_cond\_timedwait (&c, &v, &délai) //.. wait() avec attente temporelle  pthread\_cond\_broadcast(&c) //libérer tous les threads bloqués pthread\_cond\_signal(&c) //libère un seul thread pthread\_cond\_destroy(&c) //destruction |
| --- |

Création de variable conditionnelle :

| int pthread\_cond\_init ( // résultat 0 si réussite, != 0 sinon pthread\_cond\_t \*cond , // variable conditionnelle à créer pthread\_condattr\_t \*attr) ; // NULL par défaut |
| --- |

Attente non bornée :

| int pthread\_cond\_wait ( // résultat 0 si réussite, != 0 sinon pthread\_cond\_t \*vcond, // variable conditionnelle associée´ // a l'événement attendu pthread\_mutex\_t \*verrou) ; // verrou d'accès à la donnée commune |
| --- |

Cette primitive réalise un appel bloquant, qui de façon atomique déverrouille verrou et attend que la condition vcond soit annoncée, forcément par un autre thread (voir ci-après pour l’annonce).

Remarques :

* **L’attente provoque le déverrouillage de verrou**, donc un autre thread peut le verrouiller.
* **Tous les threads attendant sur la même variable conditionnelle doivent spécifier le même verrou dans l’attente** : une variable conditionnelle est associée à un seul et même verrou (mutex), mais un verrou peut être associé à un nombre quelconque de conditions.
* **La variable conditionnelle condi est utilisée comme un avertisseur** (un drapeau) : lorsqu’on est averti, il s’est passé un événement sur lequel on a demandé un réveil.

Deux façons pour réveiller des threads en attente sur une condition :

* réveiller tous les threads en attente
* réveiller un seul thread, mais ce sera un parmi ceux qui attendent, sans pouvoir choisir.

| int pthread cond signal(pthread cond t \*cond); |
| --- |

Provoque le réveil d’un seul thread.

| int pthread cond broadcast(pthread cond t \*cond); |
| --- |

provoque le réveil de tous les threads attendant la variable conditionnelle cond. Important : lors du réveil, les threads réveillés vont obtenir tour à tour automatiquement le verrouillage du verrou.

**Signal ou broadcast :**

Utiliser signal seulement si on est certain que n’importe quelle tâche en attente peut faire le travail requis et qu’il est indispensable de réveiller une seule tâche. Lorsqu’une variable conditionnelle est utilisée pour plusieurs prédicats, signal est à prohiber. En cas de doute, utiliser broadcast.

**Déverrouiller avant ou après :**

Après : engendre qu’un thread réveillé ne pourra pas obtenir le verrouillage immédiatement car le verrou est toujours indisponible.

Le choix dépendra du contexte.

### 

| **Questions Wooclap :**  L’assertion suivante est vraie :   * Après un appel à pthread\_create, le système d'exploitation choisit le fil d'exécution qui prend la main.   Un programme crée un ensemble de n threads. Que se passe-t-il si un des threads fait un appel à exit()?   * Le fil d’exécution principal et tous les fils secondaires s’arrêtent.   Dans un problème classique de rendez-vous entre tâches, chaque tâche lancée effectue un premier travail puis attend que toutes les autres aient aussi terminé leur premier travail (point de rendez-vous) avant de poursuivre leur exécution pour effectuer un second travail. Quel paradigme de synchronisation utiliserez-vous pour mettre en place ce rendez-vous? Pourquoi?   * Attente d’un événement. |
| --- |

### Réseaux : IPC :

**Besoins** : On veut des moyens de communication entre processus (lourds) s’exécutant sur une même machine (et même OS) qui n’ont pas forcément un lien de parenté (fork) ni le même utilisateur propriétaire.

* **Permettant l’échange de messages, le partage d’espace mémoire, la synchronisation**.
* Il existe plusieurs possibilités, tubes, IPC, sockets …

Communication IPC - SV :

* Files de messages : envoi/réception de **messages entre plusieurs processus.**
* Ensembles de sémaphores : outils et opérations évolués pour **résoudre des conflits d’accès et des problèmes d’ordre d’exécution des sous-ensembles d’instructions** (synchronisation).
* Mémoires partagées : mémoire **commune accessible à plusieurs processus** (en dehors de l’espace d’adressage de chacun).

**Comment plusieurs processus accèdent à un même objet IPC :**

* Par **mécanisme de clef qui permet d’obtenir un identifiant** (un entier) ou bien en obtenant **directement l’identifiant.**
* Seule l’identification par clé est vue dans cette UE.
* **Une clef, pour un utilisateur, est une paire (chemin d’un fichier existant, entier)**.
* Une clef, pour le système, est une **suite binaire permettant d’obtenir l’identifiant d’un objet IPC.**
* La même clef utilisée par différents processus et pour un type d'objet IPC, permet l'accès à un seul et unique objet IPC, donc de l’identifier.
* Donc, il faut obtenir la **clef numérique à partir de la paire (chemin d’un fichier, entier)** pour pouvoir **ensuite obtenir l’identifiant.**

Pour obtenir la clef unique :

| key\_t uneClef=ftok(const char \* chemin, int entier) |
| --- |

Ou chemin est le chemin d’un fichier existant, et entier est un entier quelconque.

Si tous les processus utilisateurs s’exécutent :

| key\_t sesame = ftok("./readme.txt", 10) |
| --- |

Ils peuvent obtenir l’identifiant d’un objet IPC via une fonction ayant la forme :

| int id\_obj = ?get(sesame, ...) |
| --- |

#### **Files de messages :**

Une **file de messages** est une structure en mémoire permettant la communication entre processus, l’unité d’échange étant un message.

Un **message** est une structure de données quelconque portant une étiquette.

Actions possibles :

* créer une file et lui affecter des droits d’accès,
* utiliser une file existante (si possible),
* déposer un message,
* **extraire un message de plusieurs façons : le premier disponible ou le premier portant une étiquette spécifique ; par exemple le premier message portant une étiquette rouge,**
* détruire une file, consulter ou gérer ses paramètres.
* A la création d’une file de messages, des droits de lecture/ écriture sont donnés aux utilisateurs (même principe que pour les fichiers).
* La gestion des accès (lectures / écritures) concurrents à une file est prise en charge par le système (exclusion mutuelle).
* Un message extrait disparaît de la file.
* La durée de vie d’une file va de sa création jusqu'à sa destruction : elle ne dépend pas de la vie des processus accédant.

**Création et identification d’une file** : Soit on crée une file et on obtient son identifiant, soit on obtient l’identifiant d’une file existante.

| int msgget (key t uneClef, int droits) |
| --- |

int = identifiant

key\_t uneclef = clef attachée à la file

int droits : droits attachés à la file ou accès demandé + options

Les droits s’énoncent comme pour la création de fichiers :

| int f id = msgget(cle, IPC\_CREAT|0666) |
| --- |

renvoie l’identifiant d’une file existante, sinon crée une nouvelle file avec les droits de lecture et d’écriture à tous les processus.

| int f id = msgget(cle, O RDONLY) |
| --- |

est une demande d’accès en lecture seule à une fille existante.

**Un message est une structure** contenant une étiquette suivie du contenu du message (la donnée à envoyer ou à recevoir).

La structure du message est décrite ainsi :

| struct msgbuf { long mtype; /\* message type, must be>0 \*/ char mtext[1]; /\* message data \*/ }; |
| --- |

Accès - extraction :

| extraire(idFile, tamponRécupération, uneEtiquette) |
| --- |

Concrètement, l’appel système est msgrcv(), de syntaxe :

| ssize t msgrcv( int identifiant, // résultat de msgget() struct msgbuf \*ptrmsg, // pointeur tampon réception size t lgmsg, // longueur max acceptée long etiquette, // quelle étiquette  int flags) // 0 pour l'instant |
| --- |

* Le résultat est le nombre d’octets lus hors étiquette.
* **Etiquette > 0** : lecture du premier message disponible avec l’étiquette e = étiquette.
* **Etiquette = 0** : lecture du premier message disponible.
* **Etiquette < 0** : lecture premier message disponible avec la plus petite étiquette e<|étiquette|.

Exemple :

| struct sMsg {long etiq ; char mot[12];} vmsg; int ret = msgrcv(f id, &vMsg, (size t)sizeof(vMsg.mot), (long) monPid, 0); |
| --- |

demande à extraire de la file ayant l’identifiant f\_id, le premier message portant l'étiquette de valeur monPid, et de copier ce message dans vMsg.

Remarque : le message disparaît de la file après son extraction.

**Accès - dépôt :**

| déposer(idFile, message avec son Etiquette) |
| --- |

Concrètement l’appel système msgsnd(), de syntaxe :

| int msgsnd( int identifiant, // résultat de msgget() struct msgbuf \*ptrmsg, // pointeur sur le message a d'exposer (avec étiquette) size t lgmsg, // longueur du message int flags) // 0 pour l'instant |
| --- |

* L’étiquette n’est par un paramètre : elle **fait partie de la structure msgbuf du message déposé avec cette étiquette**
* Le résultat indique l’opération a réussi ou échoué

La **suppression d’une file** peut se faire par la commande **ipcrm**, ou par l’appel système :

| int msgctl(int f id, int op,  .../\*struct msqid ds \*entreeTable\*/). |
| --- |

L’appel système est en fait très général et permet de gérer tous les paramètres de la file. On se contente ici de donner la forme permettant la suppression seule :

| int res = msgctl( identifiant, // résultat de msgget() IPC RMID, // constante pour la destruction NULL) // pointeur si gestion de paramètres |
| --- |

#### **Les sémaphores :**

L’idée est d'utiliser un compteur de déblocage, un mécanisme de synchronisation de processus concurrents. Il s’agit d’une structure de données qui comprend :

* Un **entier S** non négatif qui **désigne par exemple le nombre d’autorisation d’accès** à une **section critique ou un nombre de ressources partagées disponibles.**
* Une **file d’attente de processus.**

Elles sont manipulés uniquement au travers de trois opérations :

* **Init** (sémaphore sem, int nb)
* **P** (sémaphore sem, int nb) : bloque l’appelant si nb <= la valeur de sem, sinon décrémente cette valeur de nb.
* **V** (sémaphore sem, int nb) : incrémente la valeur de sem de nb et peut provoquer le déblocage de processus en attente.

Résoudre des problèmes tel que la demande d’un sous-ensemble de ressources de différents types efficacement signifie : sans interblocages, sans famine, sans attente inutile.

**Le Système V propose la notion de tableau de sémaphores**, permettant d'exécuter une **combinaison d’opérations sur un sous-ensemble des sémaphores du tableau**. Cette combinaison s'exécute de manière atomique (sans interruption).

* **Pn(Si)** qui bloque l’appelant si la valeur du sémaphore à l'indice i est inférieure à |n|
* **Vn(Si)** qui incrémente la valeur du sémaphore à l’indice i de |n| et débloque les attentes
* **Z(Si)** qui attend que le sémaphore à l'indice i soit nul (exemple : pour réaliser des rendez-vous)

**La création ressemble à celle des files de messages et de mémoires partagées.**

| int semget (key t uneClef, int nbSem, int opt) |
| --- |

Permet de créer un tableau de nbSem sémaphores ou de récupérer l’identifiant d’un tableau existant. Attention, l’objet IPC ici est le tableau. Il s’agit d’un “vrai” tableau C.

Exemple :

| int idSem = semget(cleSem, 1, IPC\_CREAT|0666); |
| --- |

**créer ou récupère l’identifiant d’un tableau à un seul sémaphore, associé à cleSem.**

La primitive semctl() est utilisée pour l’initialisation d’un ensemble de sémaphores et pour d’autres actions.

| int semctl(int semid, int semnum, int cmd, ...); |
| --- |

Revoir la structure **semun** : on peut initialiser de façon atomique un tableau de sémaphores (semnum devient le nombre d'éléments), **obtenir des valeurs courantes ou encore gérer des caractéristiques relatives à l'ensemble de sémaphores.**

| union semun { int val ; /\* cmd = SETVAL \*/ struct semid ds \*buf ; /\* cmd = IPC STAT ou IPC SET \*/ unsigned short \*array ; /\* cmd = GETALL ou SETALL \*/ struct seminfo \* buf ; /\* cmd = IPC INFO (sous Linux) \*/ } ; |
| --- |

En supposant qu’on a déclaré dans le programme une union semun comme celle décrite, on **peut initialiser un sémaphore a 1 comme suit :**

| semun egCtrl; egCtrl.val=1; if(semctl(idSem, 0, SETVAL, egCtrl) == -1){ perror(“problème init”); //suite } |
| --- |

On souhaite réaliser une combinaison **d'opérations P, V et Z sur un (sous-)ensemble de sémaphores.**

Concrètement, on utilise la fonction :

| int semop( int idSem, // résultat de semget() struct sembuf \*tabOp, // ensemble d'opérations à réaliser int nbOp) // nombre d'opérations dans ce tableau |
| --- |

Le résultat est 0 (réussite) ou −1 (échec).

Ou, toute opération (P, V ou Z) sur un sémaphore est décrite par une structure sembuf et est propre à ce sémaphore. L’ensemble des nbOp opérations sera réalisé de façon atomique.

| struct sembuf{ unsigned short sem num ; /\* Numéro du sémaphore \*/ ´ short sem op ; /\* Opération sur le sémaphore \*/ ´ short sem flg ; /\* Options par exemple SEM UNDO \*/ } ; |
| --- |

* **Les valeurs commencent à 0.**
* La valeur de n de sem\_op détermine l’opération
* Si **n<0 l’opération est P avec comme valeur |n| : tentative de décrémenter le sémaphore numéro sem\_num de |n|.**
* Si **n>0 l’opération est V : incrémente de n avec réveils des processus en attente.**
* Si **n=0 l’opération est Z : attente que la valeur du sémaphore soit 0 (voir rendez-vous).**

**Une opération P :**

struct sembuf opp;

opp.sem num=0;

opp.sem op=-1;

opp.sem flg=0;

semop(idSem,&opp,1);

**Une opération V :**

struct sembuf opv;

opv.sem num=0;

opv.sem op=+1;

opv.sem flg=0;

semop(idSem,&opv,1);

La **destruction** d’un ensemble se fera classiquement avec l’appel :

| semctl(idSem, 0, IPC\_RMID) |
| --- |

**Elle réveillera tous les processus en attente, s’il en existe.**

#### 

#### **Mémoire partagées :**

Il consiste à **disposer d’un espace de mémoire, accessible à plusieurs processus**. Chaque processus **pourra y travailler**. Il est possible de mettre des restrictions.

L’espace alloué (on parlera de segment) sera persistant : son existence sera indépendante des processus qui y accèdent.

* **Un espace, ou segment, de mémoire partagée sera créé par un processus.**
* Chaque processus **voulant y accéder demandera à s'attacher l’espace** ; après vérification des droits, **il disposera d’un pointeur vers cet espace.**
* Les processus **accédant devront gérer la synchronisation** : exclusion et protection. Classiquement, ils utiliseront des sémaphores.
* La **destruction de l’espace devra être faite par un processus ayant le droit de destruction**. En cas d'arrêt du système, l’espace sera perdu : fonctionnement identique à celui des files de message.

Actions sur un segment de mémoire partagée :

* création d’un segment ;
* demande d’attachement (obtention d’un pointeur) ;
* détachement (abandon d'accès) ;
* contrôle des paramètres dont suppression (comme pour les files de messages).

Création et identification d’un segment :

L’appel système shmget(), permet de créer un segment ou uniquement d’obtenir son identifiant.

| int shmget (key t uneClef, size t taille, int optEtDroits) |
| --- |

* **Le principe d’obtention de la clef est celui déjà vu avec ftok().**
* Les droits s'énoncent comme pour la création de fichiers.
* Lorsque le segment existe, on demande une taille inférieure ou égale à la taille du segment (0 est une bonne solution)

Exemple :

| struct uneChaine{ char c ; int x, y ; struct uneChaine \*suiv; };  int sh\_id=shmget( sesame, size t(30\*sizeof(uneChaine)), IPC CREAT|0666); |
| --- |

permet de créer un segment avec les droits d’accès de lecture et écriture à tout processus.

Le segment contient une liste chaînée.

| int sh\_id=shmget(sesame,size t(0),O RDONLY); |
| --- |

**Demande d’accès attachement :**

Pour accéder à un espace de mémoire partagée, un processus demande l’attachement de cet espace; il consiste à **obtenir un pointeur dans son espace propre**, vers cet espace extérieur.

| void \* shmat (int idMem, const void \* adrForce, int options) |
| --- |

Le type d'accès par défaut est en lecture et écriture. options permet de le modifier, par exemple de **demander l'accès en lecture seule avec SHM RDONLY.**

**On abandonne l'accès en détachant l’espace commun :**

| int shmdt (const void \* adrAtt) |
| --- |

0 : reussite, −1 : échec

**Suppression**

La suppression est similaire à celle des files de messages :

| int shmctl( int identifiant, // résultat de shmget() IPC RMID, // constante pour la destruction NULL) // pointeur si gestion de paramètres |
| --- |

| **Derniers conseils :**   * S’assurer qu’un tableau de sémaphores est initialisé avant de l'utiliser (un seul processus sera en charge de l’initialisation(logiquement le créateur de l’objet IPC). * Il est possible d’identifier le processus responsable de la création d’un objet IPC via l’option IPC\_EXCL. * Terminaison ”propre” : libération de l’espace mémoire alloué, nettoyage des tables IPC, terminaison des processus etc. * Traitement des retours d’une fonction et gestion des erreurs. * Faire attention aux problèmes liés à la synchronisation, en particulier les situations d’interblocage. Exemple : ne jamais effectuer un blocage dans une section critique sans libérer la section critique. |
| --- |

**Différence MUTEX-Sémaphores :**

| **Mutex :** Un mutex autorise le contrôle à seulement un thread il est utilisé pour l’exclusion mutuelle, forçant les autres threads qui veulent contrôler la même section à attendre la fin du premier. Exemple : 1 toilette et une clé pour 4 personnes, dès qu’une à fini, elle passe la clé à la personne suivante, mais les autres attendent de recevoir la clé. Un sémaphore avec N=1 simule à peu près un Mutex. Le mutex est un booléen, le sémaphore est un entier. |
| --- |

| **Sémaphores** : 4 toilettes et 4 clés identiques, Une personne va venir prendre une clé et aller aux toilettes (ceux libres). Quand il n’y a plus de clés, la personne doit attendre. La sémaphores restreint le nombre d’utilisateurs simultanés pour une ressource partagées avec un nombre maximum. Les threads peuvent demander à accéder aux ressources(en décrémentant les sémaphores), et peuvent signaler qu’ils ont fini d’utiliser la ressource (en incrémentant la sémaphore). Variable accessible à l’aide des opérations **P**(tester la valeur du sémaphore, peut bloquer un processus), **V**(incrémenter la valeur du sémaphore, jamais bloquant)  **Schéma :**    Si S = 1 : (un processeur et plusieurs processus concourants)   * Un premier processus P1 arrive : * P(S) test et voit S = 1 et retire 1 donc S = 0, comme S n’est pas inférieur strictement à 0, alors P1 a accès au processeur et S = S-1 = 0 * Un deuxième processus P2 arrive : * P(S) voit 0 et retire 1 donc S-1 = -1, cette fois -1<0, donc P2 ne peut pas accéder au processeur, donc la valeur S reste à 0. * Si P1 quitte le processeur, V(S) incrémente la sémaphore et donc S = S +1 donc S = 1, P2 va encore tester avec P(S) et voit S = 1 donc il décrémente et   S = S -1 = 0, P2 peut donc accéder au processeur (ressource critique)  **int sem\_init**(sem\_t ∗sem , int pshared , unsigned int value) ;  **int sem destroy** ( sem\_t ∗sem ) ;  **int sem\_wait** ( sem\_t ∗sem ) ;  **int sem\_trywait** ( sem\_t ∗sem ) ;  **int sem\_timedwait** ( sem\_t ∗sem, construct timespec ∗ abs\_timeout ) ;  **int sem\_post**( sem\_t ∗sem ) ;  Les variables conditionnelles permettent de mettre en attente dans une queue un thread en débloquant (atomiquement) un mutex. Une variable de condition fonctionne toujours de pair avec un mutex.  **signal(c)** : débloque le premier thread de la queue de c.  **broadcast(c)** : débloque tous les threads de la queue de c.    L’attente provoque le déverrouillage de verrou, donc un autre thread peut le verrouiller. |
| --- |

| Conclusion :  **MUTEX** : Permet à un seul thread d'accéder à la ressource, objet créé par les threads.  **SÉMAPHORES** : Mécanisme de signalement, autorise un certain nombre de threads à accéder à la ressource. |
| --- |

**Vidéos :**

| **Files de messages :**  Liste de messages stockés dans un espace du kernel. Créer ou ouvrir une liste existante avec **msgget**(). Ajouter un message à la fin de la queue avec msgsnd(). Recevoir un message de la queue avec **msgrcv**(). Contrôler les propriétés de la queue avec **msgctl**().  Le message est contenu dans une structure, qui contient une étiquette suivie du contenu du message. |
| --- |

| **Sémaphores :** On crée les sémaphores avec **semget()** qui créer un tableau de sémaphore identifié par une clé (la clé interne). Dans la fonction nous avons la clé externe à introduire, le nombre de sémaphores dans le groupe et les autorisations (lecture etc).  Création de la structure sembuf ou on indique l’opération qui sera affectée à un sémaphore, en tant que premier paramètre, semop est la décrémentation ou l’incrémentation. Après avoir créé le tableau de sémaphores et la structure qui contient les opérations, la prochaine étape est de réaliser les opérations, avec **semop()**. Il réalise les opérations du tableau d'opération. Prend en paramètre la clé interne, le tableau de structure des opérations pour les sémaphores, et le nombre d”opérations totales à appliquer. Pour accéder à la ressource, le thread doit tester le sémaphore, regarder si la valeur est > 0 et la décrémenter, et attendre sinon. Une fois fini, incrémentation de la sémaphore. On va donc obtenir la sémaphore avec semget, la contrôler avec semctl et faire des opérations avec semop.  **Semctl()** sert à appliquer certaines commandes sur des sémaphores comme SETVAL pour initialiser la valeur du sémaphore, ou RMID pour supprimer.  **Int Num = semget(1000, 2, 0666 | IPC\_CREAT);**  créer un tableau de 2 sémaphores numérotés 0 et 1, de clé externe 1000 identifiant le groupe et de droit d’accès 0666. La clé interne est fournie dans Num.  **struct sembuf tabop[2]{ {0,1,0}, {1,-1,0}};**  indique que le sémaphore 0 doit être incrémenté de 1 (opération V) et que le sémaphore 1 doit être décrémenté de 1 (opération P).  **semop(NUm, tabop, 2);**  réalise les deux opérations de tabop de manière indivisible (tout ou rien).  **semctl(Num, N, SETVAL, V);**  Initialise le sémaphore N du groupe Num à la valeur V.  **semctl(Num, 0, IPC\_RMID, 0);**  Détruit le groupe de sémaphores Num. |
| --- |

Exemple du carrefour routier à faire : <https://www.youtube.com/watch?v=9OZofG500Js&list=WL&index=2> 7:17

| **Mémoire partagée** : Moyen de communication IPC, il faut une clé externe et une fois le segment de mémoire partagée créé, on aura une clé interne. 4 appels système pour la gestion des segments de mémoire partagées :  **shmget(share memory get)** : ouvre un segment de mémoire partagée de clé externe key, et fournit en retour la clé interne. Avec size la taille de la mémoire partagée et shmflg les droits d’accès et si il faut créer la mémoire partagée ou seulement la référencer.  **shmat(share memory attach)** : attache la mémoire partagée de clé interne shmid à une adresse virtuelle du processus. Si shmaddr vaut NULL, le système cherche une adresse virtuelle non utilisée. shmflg indique un accès en lecture seule ou lecture écriture.  **shmctl(share memory control)** : pour consulter ou modifier les caractéristiques ou détruire le segment de mémoire partagée :   * cmd = IPC\_STAT pour obtenir des informations dans BUF * cmd = IPC\_SET pour que le système prenne en compte certaines modifications faites dans buf. * cmd = IPC\_RMID pour détruire le segment   shmdt(share memory detach) : Le segment est détaché du processus (il est accessible pour le processus, mais il continue à exister).  Les appels aux fonctions :  **int shmget** (key\_t key, int size, int shmflg);  **char \* shmat**(int shmid, char\* shmaddr, int shmflg);  **int shmctl**(int shmid, int cmd, struct shmid\_ds\* buf);  **int shmdt**(char\* shmaddr);  **int Num = shmget(1000, 4, 0666 | IPC\_CREAT);**  créer une mémoire partagée de clé externe 1000, de clé interne Num, de taille 4 octets et de droits d’accès 0666.  **int \*N = (int \*) shmat(Num, NULL, 0);**  N est un pointeur (d’entier) sur le début de la zone de mémoire partagée de numéro interne Num.  **shmctl(Num, IPC\_RMID, NULL);**  destruction du segment de clé interne Num.  Chaque processus à son espace de d’adressage privé, et accède à la mémoire physique via son espace d’adressage |
| --- |

<https://fr.gadget-info.com/difference-between-semaphore>

| **Base de comparaison** | **Sémaphore** | **Mutex** |
| --- | --- | --- |
| **De base** | Le sémaphore est un mécanisme de signalisation. | Mutex est un mécanisme de verrouillage. |
| **Existence** | Le sémaphore est une variable entière. | Mutex est un objet. |
| **Une fonction** | Sémaphore permet à plusieurs threads de programme d'accéder à une instance finie de ressources. | Mutex permet à plusieurs threads de programme d'accéder à une seule ressource mais pas simultanément. |
| **La possession** | La valeur de sémaphore peut être modifiée par tout processus d’acquisition ou de libération de la ressource. | Le verrou d’objet Mutex n’est libéré que par le processus qui a acquis le verrou dessus. |
| **Classer par catégories** | Le sémaphore peut être catégorisé en comptage de sémaphore et binaire. | Mutex n'est pas catégorisé davantage. |
| **Opération** | La valeur du sémaphore est modifiée à l’aide des opérations wait () et signal (). | L'objet Mutex est verrouillé ou déverrouillé par le processus qui demande ou libère la ressource. |
| **Ressources occupées** | Si toutes les ressources sont utilisées, le processus demandant une ressource exécute l'opération wait () et se bloque jusqu'à ce que le nombre de sémaphores soit supérieur à un. | Si un objet mutex est déjà verrouillé, le processus de demande de ressources attend et est mis en file d'attente par le système jusqu'à ce que le verrou soit libéré. |

**Annales : Sujet 2020**

Si plusieurs messages de différentes étiquettes sont présents dans une file de messages, un processus peut extraire un message de n'importe quelle étiquette, le premier message qui sera extrait est le premier déposé ayant l’étiquette demandée.

L'étiquette dans la structure d’un message transmis via une file de messages permet d’identifier le type de messages à recevoir.

Si un ensemble de processus réalisent une opération Z sur un sémaphore S au moment ou la valeur de S est 10 et sachant que ces processus ont les droits de lecture et d’écritures sur S, quand la valeur de S sera égale à 0, tous les processus de l’ensemble seront réveillés, tous les processus de l’ensemble sont mis en attente.

Dans une application un ensemble de processus partage un tableau de sémaphores en lecture et écriture. Ce tableau contient au moins deux sémaphores. Chaque processus peut demander à exécuter de manière non atomique plusieurs opérations sur différents sémaphores du tableau, ou peut demander à exécuter ensemble et de manière atomique plusieurs opérations sur différents sémaphores du tableau.

Si on utilise des threads pour réaliser un application réelle, il n’est pas indispensable que le nombre de threads pouvant s'exécuter en parallèle, ne dépasse pas le nombre de cœurs du processeur utilisé pour l’exécution.

Si un serveur en mode connectée (itératif ou concurrent) exécute listen(Br, 10), il peut dialoguer avec un nombre quelconque de clients.

En mode TCP si la couche transport décide de fermer la socket et si l’application locale est en attente d’un message sur cette socket, la fonction recv renvoie 0.

En mode UDP, un serveur peut traiter simultanément plusieurs clients dans un seul processus et sans avoir à créer des threads.

En mode UDP, un processus ne doit pas forcément connaître la taille exacte d’un message pour pouvoir le recevoir.

Pour réduire le coût des communications, la file de messages peut permettre à un serveur de transmettre un message à plusieurs clients.

Pour réduire le coût des communications, la file de messages est la meilleure option quand on doit d’échange implique uniquement des échanges de type requête-réponse.